DISSERTATION

ANALYSIS AND MODELING OF CELLS, CELL BEHAVIOR, AND HELICAL BIOLOGICAL MOLECULES

Submitted by

Steven Benoit

Department of Mathematics

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring, 2011

Doctoral Committee:

Advisor: Vakhtang Putkaradze

Patrick D. Shipman
Donald J. Estep
Mario C. Marconi
Stuart A. Tobet

ABSTRACT

ANALYSIS AND MODELING OF CELLS, CELL BEHAVIOR, AND HELICAL BIOLOGICAL MOLECULES

Mathematical models of biological systems have evolved over time and through the introduction and growth of computer simulation and analysis. Models have increased in sophistication and power through the combination of multi-scale approaches, molecular and granular dynamics simulations, and advances in parallelization and processing speed. However, current cell models cannot accurately predict behaviors at the whole-cell scale, nor can molecular models predict accurately the complex shape assumed by large biological molecules including proteins, although significant progress is being made toward this goal. The present work introduces new models in three domains within biological systems modeling. We first discuss a phenomenological model of observed cell motions in developing tissue that characterizes cells according to a best-fit generalized diffusion model and combines this data with Voronoi diagrams to effectively visualize patterns of cell behavior in tissue. Next, we present a series of component models for cells and cell structure that support simulations involving tens to hundreds of cells in a way that captures behaviors ignored by existing models, including pseudopod formation, membrane mechanics, cytoskeletal polymerization / depolymerization, and chemical signal transduction. The resulting

models exhibit many of the behaviors of real-world cells including polarization and chemotaxis. Finally, we present a method for analysis of biological molecules that form helical conformations that includes long-range electrostatic interactions as well as short-range interactions to prevent self-intersections. We consider the stability of molecules with repeating monomers that include off-axis charge concentrations and derive energy landscapes to identify stable conformations, then analyze helical stability using geometric methods.

ACKNOWLEDGMENTS

I am grateful to V. Putkaradze for longstanding support and encouragement, and to the Department of Mathematics for supporting my rather lengthy doctoral program. I thank S. Tobet and the members of his lab, most notably M. Stratton, B. Searcy, and K. Frahm who provided patient help as I learned some measure of cell biology. I also acknowledge the able assistance of D. D. Holm with the work on molecular simulation and geometric methods, and of P. Ziherl and A. Hocevar with whom I had fruitful discussions on modeling and analysis of cell motions. Finally, I am grateful to T. Chen, M. de Miranda, G. Majdic, D. Strle, and my cofellows Z. Cashero, M Easterly, C. Hartshorn, M. Duwe, A. Chen, R. Hoppal, and C. Bishop for including me in their interdisciplinary program and giving me the opportunity to experience such a multi-faceted project.

This work has been supported by a National Science Foundation grant number GDE-0841259; Colorado State University, Thomas Chen, Principal Investigator, Michael A. de Miranda and Stuart Tobet Co-Principal Investigators.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION: CELLS, BIOLOGICAL MOLECULES

The study and analysis of the composition, movement, and self-organization of cells into functional tissues spans the boundaries of biology, rigid body and continuum mechanics, fluid dynamics, chemistry, thermodynamics, and mathematics. In the present work, we explore three related aspects of this extremely broad field of study, each of which providing a window to understanding some aspect of the behavior of this and other complex biological systems.

In a way, this is the quintessential multi-scale problem. Behaviors at the molecular scale drive the construction of specialized proteins and polymers that dictate both the mechanical properties as well as the chemical responses of cells and facilitate cell communication. At a scale several orders of magnitude larger, cellular structures including cell membrane, cytoskeleton, and organelles drive bulk behavior including migration and adhesion, but simulation at this scale is well out of the reach of molecular methods. At even larger scales, the formation of tissue-scale structures from thousands of cells create a topography through which migrating cells travel to their destinations following chemical and mechanical cues, with the end result being a macro-scale organism that functions properly only because each cell migrates from

its place of birth to its ultimate location correctly. An understanding of this process covers ten orders of magnitude, from Angstroms to meters, and represents one of the most challenging modeling problems known [1, 2].

1.1 Outline of the Dissertation

Observation and analysis

The essential first step in construction of models of a complex system is observation and analysis of the system behavior at a scale appropriate to the modeling effort. In Chapter 2, we present an automated system for analysis of cell movements as observed using fluorescence video microscopy. This analysis generates a useful classification of behaviors for groups of cells in living tissue, and also provides a basis for comparison with simulated collections of cells under similar conditions.

This analysis is based on an observation protocol developed at Colorado State University that allows *in vitro* visualization of cell movements in embryonic tissues for up to three days [3]. The product of this imaging protocol is a series of multi-plane images, where the focus is swept through three planes, at a separation of 10 microns, in an attempt to capture images of cells through a roughly 40 micron thickness of tissue.

The automated tool processes the raw images produced from the microscope into a series of cell trajectories, then analyzes the diffusive properties of those trajectories, and correlates motion parameters with physical position in the sample, providing both numerical and visual representations of the results. Examples from the paraven-

tricular region of hypothalamus the are used to demonstrate the technique.

Multi-scale modeling

There are a number of existing models of cells, cell components, and multi-cellular systems at various scales, ranging from molecular dynamics simulations of cell membranes and motor proteins through continuum mechanical models of large-scale tissue structures and models of cell dynamics and motion based on differential equations or stochastic simulation. If we consider the set of models that begin with first-principle representations of the molecules, liquids and bond that constitute a cell as *bottom up* models, the most elaborate and robust of these are, at best, capable of simulating a small subset of a cell (say, the plasma membrane), for a very short time scale (on the order of a few milliseconds). These models can generate predictions of physical properties such as elastic or bulk moduli, viscosity, or tensile strength, which can be tested against observation, resulting in increased understanding of those component systems, but this class of model cannot hope to simulate whole-cell systems, much less systems of interacting cells or tissue.

The other approach, which we call *top down*, consists of models based on phenomenology of large aggregates of cells or tissue, and relies either continuum mechanical approaches that discard the notion of individual cells, or stochastic methods, which model cells as discrete points. While these approaches have achieved some success in modeling and predicting real-world behavior of cell aggregates, they do not provide insight into the cell-level processes that generate their predicted behaviors.

We see a need for a modeling "middle ground", or *mesoscale* models of cell aggre-

gates. These models would capture the behaviors of cells that lead to migration and tissue formation, while stopping short of attempting to model at the molecular level. In Chapter 3, we present a class of mesoscale models of cells and their interactions, and describe a method by which these models can be integrated into a simulation environment that can support new models of individual cell components as they are developed.

Biological molecules

At its lowest level, models of biological systems must consider individual molecules and their behavior and characteristics. For simple molecules or ions, this is a straightforward exercise in molecular dynamics. For more complex biomolecules, however, the mechanical properties of long-chain systems require a more efficient mechanism of study due to the large number of constituent parts. Many of these molecules, including the primary structural proteins of the cellular cytoskeleton (actin, tubulin, keratin, vimentin, etc.), form polymers with uniform structure along their length, opening these molecules to forms of analysis that take advantage of their intrinsic helical symmetries.

In Chapter 4, we present a method for quickly analyzing and classifying the stable conformations of helical molecules based on a geometric analysis that leverages the inherent symmetries in the molecules to identify helical stable shapes as well as explore the levels of stability of those shapes. This method can be applied to molecules that have arbitrarily complex monomers with arbitrary charge distributions, where charges need not be on the helical axis of the molecule. Long-range electrostatic forces are included in the analysis, and it is the interplay of these forces with elastic forces that result in stable helical conformations.

Implementations

Finally, we provide, in a series of appendixes, a complete implementation of the three systems described in this work: the automated analysis tool to extract and analyze cell trajectories from sequences of multi-plane microscopic images; the system

of meso-scale models of cell components and the simulation environment in which they can be integrated; and software to generate the energy landscapes and identify stable helical conformations of biological molecules consisting of repeating monomer chains with generalized off-axis charge distributions. Each of these implementations are presented in the Java language, and are also available for download from the author's web site [4].

Remark Because of the diverse nature of the material covered in this dissertation, we choose to include a separate References section at the end of each chapter. It is hoped that this makes resources easier for the reader to locate.

REFERENCES

[1] Masaru Tomita. Whole-cell simulation: a grand challenge of the 21st century. *Trends in Biotechnology*, 19(6):205–210, June 2001.

[2] David Harel. A grand challenge for computing: Towards full reactive modeling of a multi-cellular animal. *Lecture Notes in Computer Science*, 2937:39–60, 2003.

[3] Stuart A. Tobet, Heather J. Walker, Marianne L. Seney, and Kwok W. Yu. Viewing cell movements in the developing neuroendocrine brain. *Integrative and Comparative Biology*, 43(6):794–801, 2003.

[4] Steven Benoit. Personal web site. http://lamar.colostate.edu/∼sbenoit/.

CHAPTER 2

IN-VITRO ANALYSIS OF CELL BEHAVIOR

2.1 Introduction

An essential prerequisite to any modeling effort is a phenomenological study of the system under consideration. Observations can guide model development and provide a set of benchmarks against which model performance can be measured. The value of the model is then gaged by its ability to reproduce meaningful behaviors observed in the real-world system.

Video fluorescence microscopy is an essential tool for the study of cells and cell migration, and especially in the study of *in vitro* tissue slices. Of particular interest is the ability to prepare slices for imaging, then introduce reagents during the imaging cycle to allow detection of alterations in cell behavior that can be attributed to the reagents [1]. The output of these microscopy cycles consists of a series of images representing discrete time points (typically several minutes apart), and a set of distinct focal planes within the sample for each time point.

The analysis of the data that comes from video microscopy traditionally involves manual adjustment of image intensities, manual alignment of each successive image to

compensate for motion of the sample on the stage during imaging, manual identification of points of interest (cells) in each frame, then assembly of those point locations into paths that can be subsequently evaluated to characterize cell behavior.

Outline of the chapter

We first describe an analysis procedure that can be applied to a series of images generated by video fluorescence microscopy. Section 2.2 describes the steps of an analysis of cell trajectories that begins with the raw output images produced by a microscope under the control of MetaMorph® Microscopy Automation and Image Analysis Software[1], and proceeds through the calculation of diffusion parameters for cell trajectories and the generation of a set of visualizations of the resulting cell motion characterizations. Examples of the results of the analysis when applied to the motion of cells near the paraventricular nucleus of the embryonic hypothalamus in mice are provided for illustration. Section 2.3 then describes a generalized framework for video analysis, consisting of a specification for *filters* and a data *pipe* from which the filters receive data and to which they deliver filtered data.

The Java source code of the author's implementation of the automated analysis tool is available in the Appendix and on the author's web site [2].

2.2 Analysis of cell trajectories

Throughout this section, we will denote an individual image frame with the letter **F** (an abbreviation for *frame*). We use subscripts to index these image frames – an

[1]MetaMorph is a registered trademark of Molecular Devices, Inc.

image frame set $\{\mathbf{F}_{z,t}\}$, $1 \leq z \leq m$, $1 \leq t \leq n$, is a set of image frames $\mathbf{F}_{z,t}$, where the index t represents the time, and z represents the focal plane, with respect to an arbitrary reference focal plane. A typical image frame set that we consider might contain thirty to forty time points, and three z values, separated from each other by some fixed distance (typical values of this separation would be 10 microns, or the order of the size of a cell).

During the analysis of a set of images, we will often generate new image frames from existing frames, and so we use parenthesized superscripts to indicate the generation of the frame. For example, $\mathbf{F}_{z,t}^{(0)}$ might represent the raw image generated by a camera attached to the microscope, while $\mathbf{F}_{z,t}^{(1)}$ represents the results of applying some filter to that raw image, $\mathbf{F}_{z,t}^{(2)}$ represents the output of the second stage of image processing, and so forth.

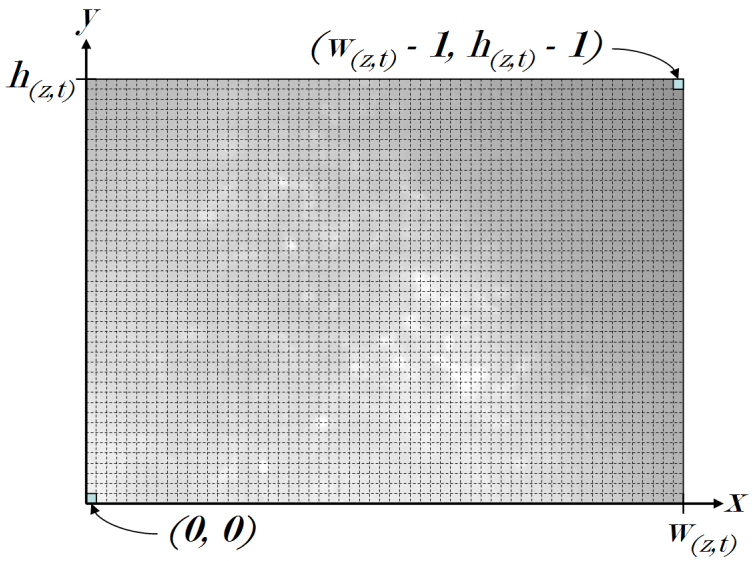

Figure 2.1: Cartesian coordinate system applied to image frames.

Within an image frame, we specify a Cartesian coordinate system, as shown in

Figure 2.1, with the origin in the lower left corner of the image, x axis increasing to the right, and y axis increasing upward, scaled so one unit on each axis represents one pixel in the image. We denote the width and height of image frame $\mathbf{F}_{z,t}^{(i)}$ by $w_{z,t}^{(i)}$ and $h_{z,t}^{(i)}$, respectively. For example, an 800×600 pixel image frame has $w_{z,t}^{(i)} = 800$, and $h_{z,t}^{(i)} = 600$. We refer to a pixel by the coordinates at its lower left corner – the pixel at the bottom left corner of the image frame would be referred to by coordinates $(0, 0)$, while the pixel in the upper right corner of an image frame is assigned coordinates $(w_{z,t}^{(i)} - 1, h_{z,t}^{(i)} - 1)$. We assume throughout that pixels have a 1:1 aspect ratio.

Remark This coordinate system is chosen to match the mathematician's preferred frame, not the computer scientist's preferred frame. Computer imaging systems typically place the $(0, 0)$ coordinate in the upper left corner and allow y to increase downward. Making this choice, however, subjects the interpretation angle measures and functions like sine and cosine to confusion, and so we adopt the more traditional system.

2.2.1 Equalization of pixel intensities

The original raw image frames $\{\mathbf{F}_{z,t}^{(0)}\}$ are collected with a digital camera which generates intensity values on a scale from 0 (no incident light) to some upper limit (typically either 255 or 65,535, corresponding to 8- or 16-bit unsigned binary values, respectively) which represents the maximum measurable light intensity, and presumably any intensities beyond this limit. In practice, however, the actual range of intensities measured is a very small subset of this range, concentrated near the lower limit. The result is that if the images are presented visually on a computer monitor,

they appear as featureless black. Moreover, there is no guarantee that images taken at different time points or different Z planes have the same baseline intensity scale (the ambient light level may have changed, for example).

To do meaningful analysis, therefore, the intensity λ of the pixels in each image are first scaled so they fill the range $0 \leq \lambda \leq 255$, and image frames are further intensity-scaled such that all image frames have the same mean intensity. Letting $\lambda^{(0)}_{x,y,z,t}$ be the intensity of the pixel at coordinates (x, y) in image frame $\mathbf{F}^{(0)}_{z,t}$, and letting $w^{(0)}_{z,t}$ and $h^{(0)}_{z,t}$ be the width and height of $\mathbf{F}^{(0)}_{z,t}$ (in pixels), respectively, we first compute the average image intensity of each image as

$$\bar{\lambda}^{(0)}_{z,t} = \frac{1}{w^{(0)}_{z,t}\, h^{(0)}_{z,t}} \sum_{x=1}^{w^{(0)}_{z,t}} \sum_{y=1}^{h^{(0)}_{z,t}} \lambda^{(0)}_{x,y,z,t}\,,$$

and the average intensity $\bar{\Lambda}^{(0)}$ over all image frames as

$$\bar{\Lambda}^{(0)} = \frac{1}{n\,m} \sum_{t=1}^{n} \sum_{z=1}^{m} \bar{\lambda}^{(0)}_{z,t}\,.$$

The maximum and minimum overall intensity is given by

$$\lambda^{(0)}_{max} = \max\{\lambda^{(0)}_{x,y,z,t} : 1 \leq t \leq n, 1 \leq z \leq m, 0 \leq x \leq w_{z,t}, 0 \leq y \leq h_{z,t}\}\,,$$

$$\lambda^{(0)}_{min} = \min\{\lambda^{(0)}_{x,y,z,t} : 1 \leq t \leq n, 1 \leq z \leq m, 0 \leq x \leq w_{z,t}, 0 \leq y \leq h_{z,t}\}\,.$$

We define a scale factor $\alpha_{z,t}$ for each image frame as

$$\alpha_{z,t} = \frac{255\bar{\Lambda}^{(0)}}{\bar{\lambda}^{(0)}_{z,t}(\lambda^{(0)}_{max} - \lambda^{(0)}_{min})}\,,$$

then generate by intensity-leveled images $\mathbf{F}^{(1)}_{z,t}$ as image frames with $w^{(1)}_{z,t} = w^{(0)}_{z,t}$, $h^{(1)}_{z,t} = h^{(0)}_{z,t}$, and

$$\lambda^{(1)}_{x,y,z,t} = \max\left((\lambda^{(1)}_{x,y,z,t} - \lambda^{(0)}_{min})\alpha_{z,t}, 255\right)\,.$$

An example of a raw and intensity-leveled image frame are shown in Figure 2.2.

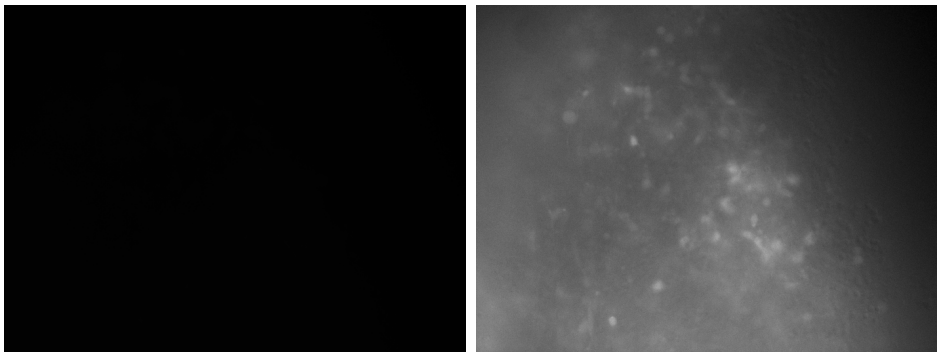

Figure 2.2: A raw image frame from microscope, and the resulting intensity-leveled image.

2.2.2 Merging focal planes

The reason that multiple focal planes are gathered during image acquisition is to increase the probability that a given cell in the sample is within reasonable focus in at least one of the planes. The focal planes are roughly one cell body diameter apart, meaning that any cell within a section of tissue whose depth is equal to three cell diameters should be within half of its body diameter of the actual focal plane in one of the images. In-focus cells will appear as intensity maxima given the nature of fluorescence microscopy, while out-of-focus cells appear as a vague brightening of a broad region of the image.

Since the extents of an image frame in the x or y direction far exceeds the three cell diameters of resolution in the z direction, it makes little sense to attempt to include z coordinates in analyses of cell positions. However, capturing this extra z information may extend the time over which a cell is visible if it is moving transverse to the focal plane. We therefore merge the focal plane images into a single image that captures the intensity maxima revealed in each of the individual images. The

merged image frames $\mathbf{F}_t^{(2)}$ (where we may now omit the z subscript), are created with $w_t^{(2)} = w_{z,t}^{(1)}$, $h_t^{(2)} = h_{z,t}^{(1)}$, and

$$\lambda_{x,y,t}^{(2)} = \max_{1 \le z \le m} \left\{ \lambda_{x,y,z,t}^{(1)} \right\} .$$

Figure 2.3 shows a portion of the result of merging three intensity-leveled image frames representing three different focal planes at a single time point. Realistically, there is a short time interval between the exposures, as the microscope would take some time to adjust its focal plane, but this time is very short (a few seconds) compared to the interval between time points (five or ten minutes, typically), so we consider these images to be simultaneous. In this case, the focal planes were very close and the input images show little variation.

2.2.3 Gaussian smoothing

Before we extract intensity maxima, we next apply a Gaussian smoothing to the images. The reason for this is that without smoothing, the location of an actual intensity maximum may be misinterpreted if noise gives a point near the actual maximum a higher intensity, making that point appear to be the location of the maxima. To avoid this, we generate a Gaussian kernel based on the function

$$G(x, y) = \frac{1}{2\pi} e^{-x^2 - y^2} ,$$

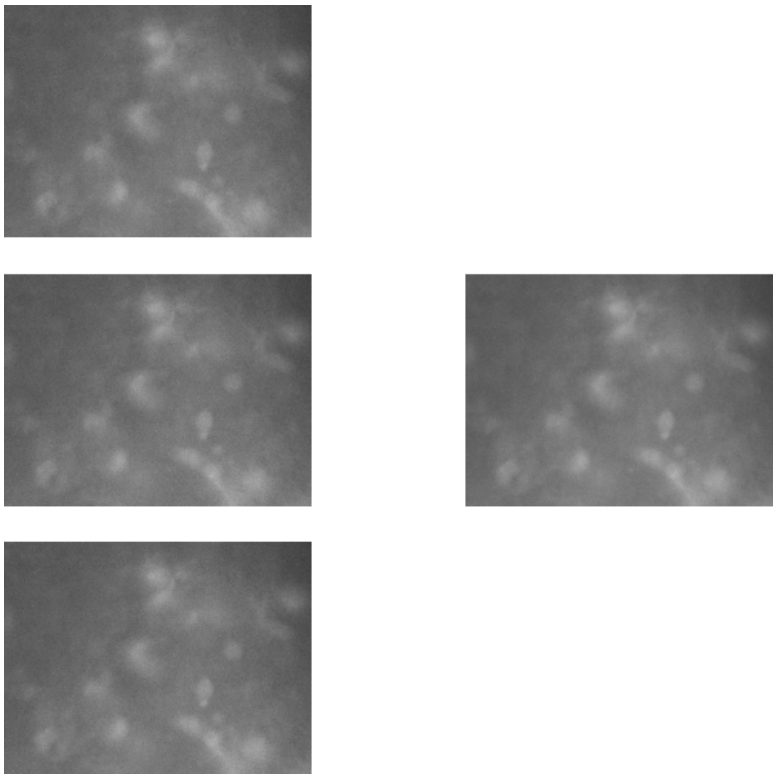

Figure 2.3: A magnified portion of three focal planes for a particular time point, and the resulting merged image.

where we take x and y as integers in the range $-2 \leq x \leq 2$, $-2 \leq yle2$. The result is a five by five kernel with the values,

$$G = \frac{1}{2\pi} \begin{bmatrix} e^{-8} & e^{-5} & e^{-4} & e^{-5} & e^{-8} \\ e^{-5} & e^{-2} & e^{-1} & e^{-2} & e^{-5} \\ e^{-4} & e^{-1} & 1 & e^{-1} & e^{-4} \\ e^{-5} & e^{-2} & e^{-1} & e^{-2} & e^{-5} \\ e^{-8} & e^{-5} & e^{-4} & e^{-5} & e^{-8} \end{bmatrix}.$$

Normalizing this kernel,

$$G' = \frac{1}{2\pi\left(1 + \frac{4}{e} + \frac{4}{e^2} + \frac{4}{e^4} + \frac{8}{e^5} + \frac{4}{e^8}\right)} \begin{bmatrix} e^{-8} & e^{-5} & e^{-4} & e^{-5} & e^{-8} \\ e^{-5} & e^{-2} & e^{-1} & e^{-2} & e^{-5} \\ e^{-4} & e^{-1} & 1 & e^{-1} & e^{-4} \\ e^{-5} & e^{-2} & e^{-1} & e^{-2} & e^{-5} \\ e^{-8} & e^{-5} & e^{-4} & e^{-5} & e^{-8} \end{bmatrix}.$$

we can then convolve it with the region surrounding each pixel in image frame $\mathbf{F}_t^{(2)}$ to obtain $\mathbf{F}_t^{(3)}$, taking any pixel value with coordinates outside the limits of $\mathbf{F}_t^{(2)}$ to have an intensity of zero. The result is a new set of image frames with $w_t^{(3)} = w_t^{(2)}$, $h_t^{(3)} = h_t^{(2)}$, and

$$\lambda_{x,y,t}^{(3)} = \sum_{i=1}^{5}\sum_{j=1}^{5} G'_{i,j}\lambda_{x+i-2,y+j-2,t}^{(2)}\,.$$

Figure 2.4 shows a portion of an image frame before and after Gaussian smoothing was applied.

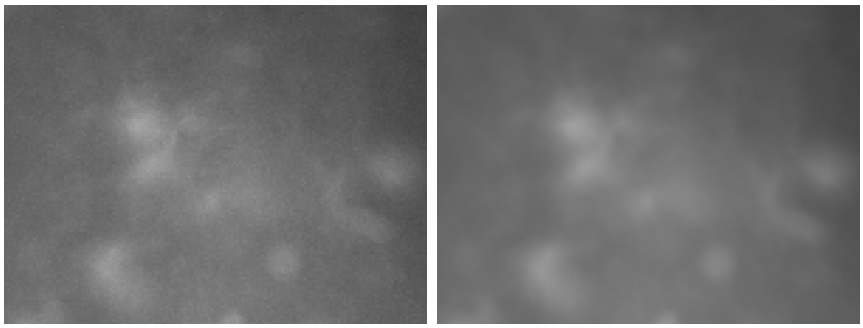

Figure 2.4: The effect of Gaussian smoothing on an image frame.

2.2.4 Global motion compensation

It is an unfortunate fact that during the course of observation, a tissue sample will move on the microscope stage, resulting in a shift of the sample in images. This

movement may be global (vibration or shock moves the slide on the stage), or local (the tissue changes shape, shrinking or expanding during image acquisition). If this motion is not compensated, any cell positions and trajectories will be inaccurate, reducing the validity of results, or making what appear to be high-confidence predictions that are blatantly wrong. For example, tissue shrinkage might result in very clean looking trajectories with a clear directional bias, but which bear no relationship to actual cell movement within its surrounding tissue.

Two factors are at work here - global movements of the sample on the stage, and local changes in the relative size and shape of the tissue. These two effects need different strategies to address them, and at this stage, we focus on compensating for global motion of the sample on the stage. To prevent shrinkage and shape change effects from confounding this process, we restrict our attention to the center of the image, acting to stabilize that portion of the image against motion. We compute the offset vector $(\Delta x, \Delta y)$ for each image frame such that it correlates most strongly with the image frame from the preceding time point, where the correlation is taken over the middle fourth of the image in each axis. In particular, to compute the correlation between the image frame $\mathbf{F}^{(3)}_{t-1}$ and $\mathbf{F}^{(3)}_{t}$ for a particular vector offset $(\Delta x_i, \Delta y_i)$, we set $x_- = \max(\Delta x_i, 0)$, $x_+ = \min(w^{(3)}_t, w^{(3)}_t + \Delta x_i)$, $y_- = \max(\Delta y_i, 0)$, and $y_+ = \min(h^{(3)}_t, h^{(3)}_t + \Delta y_i)$, then the mean squared error (MSE) between the two images is

$$\text{MSE} = \frac{1}{(x_+ - x_-)(y_+ - y_-)} \sum_{x=x_-}^{x_+} \sum_{y=y_-}^{y_+} \left(\lambda^{(3)}_{x,y,t-1} - \lambda^{(3)}_{x-\Delta x_i, y-\Delta y_i, t} \right)^2 .$$

We select the $(\Delta x_i, \Delta y_i)$ for which the MSE is minimized. In actual implementations, we may restrict this search to vectors $(\Delta x_i, \Delta y_i)$ whose length does not exceed some limit, in the interest of efficiency. Once the vector is selected for each

t, $2 \leq t \leq n$, we chain these vectors together into a trajectory, and compute the minimum bounding rectangle R that contains the entire trajectory.

$$R = \left\{ (x,y) : \min_{2 \leq j \leq n} \sum_{i=2}^{j} \Delta x_i < x < \max_{2 \leq j \leq n} \sum_{i=2}^{j} \Delta x_i, \right. \\ \left. \min_{2 \leq j \leq n} \sum_{i=2}^{j} \Delta y_i < y < \max_{2 \leq j \leq n} \sum_{i=2}^{j} \Delta y_i \right\}.$$

We denote the boundaries of this rectangle as $R_l = \min\{x : x \in R\}$, $R_r = \max\{x : x \in R\}$, $R_b = \min\{y : y \in R\}$, and $R_t = \max\{y : y \in R\}$, where the subscripts l, r, b, and t represent left, right, bottom, and top, respectively. We then construct image frames $\{\mathbf{F}_t^{(4)}\}$ with $w_t^{(4)} = w_t^{(3)} + R_r - R_l$, and $h_t^{(4)} = h_t^{(3)} + R_t - R_b$. The first frame is not shifted, but has a border of zero-intensity pixels,

$$\lambda_{x,y,1}^{(4)} = \begin{cases} \lambda_{x+R_l, y+R_b, 1}^{(3)} & \text{if } 0 \leq x + R_l \leq w_t^{(3)} \text{ and } 0 \leq y + R_b \leq h_t^{(3)} \\ 0 & \text{otherwise} \end{cases}.$$

Subsequent frames are shifted by the cumulative resultant of all preceding vectors $(\Delta x_i, \Delta y_i)$. If we let

$$(Dx_i, Dy_i) = \left(\sum_{j=1}^{i} \Delta x_j, \sum_{j=1}^{i} \Delta y_j \right),$$

then for $2 \leq t \leq n$,

$$\lambda_{x,y,t}^{(4)} = \begin{cases} \lambda_{x+R_l+Dx_i, y+R_b+Dy_i, t}^{(3)} & \text{if } 0 \leq x + R_l + Dx_i \leq w_t^{(3)} \\ & \quad \text{and } 0 \leq y + R_b + Dy_i \leq h_t^{(3)} \\ 0 & \text{otherwise} \end{cases}.$$

Figure 2.5 shows three image frames from a globally motion compensated sequence. The black borders where the image was shifted to compensate for global motion are apparant.

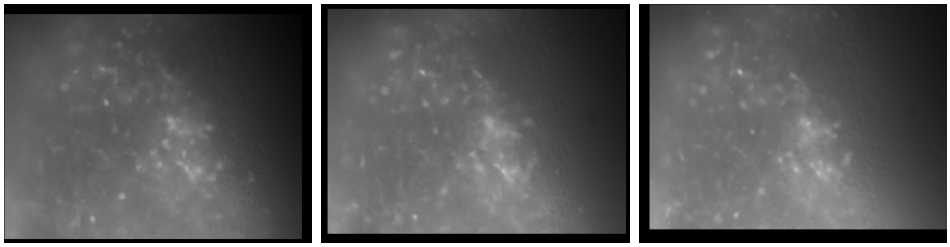

Figure 2.5: Image frames after global motion-compensation has been applied.

2.2.5 Identification of cells as local maxima

With the images stabilized against global motions, we next identify cells in the image frames $\{\mathbf{F}_t^{(4)}\}$. We first define a threshold intensity λ_{min} and consider only pixels with an intensity $\lambda_{x,y,t} >= \lambda_{min}$ (experiment suggests a value of $\lambda_{min} = 80$ is reasonable). We then define a maximum radius r_{max} that we expect a cell to have in our image, and for each pixel $\lambda_{x,y,t}$ in a given image frame $\mathbf{F}_t^{(4)}$, we classify the pixel as a local maximum if

1. the pixel has the highest intensity of any pixel within r_{max},

2. the pixel has no neighbor that is black, and

3. there is at least one pixel $\lambda_{x',y',t}$ within r_{max} with $|\lambda_{x',y',t} - \lambda_{x,y,t}| \geq \frac{\lambda_{min}}{4}$.

Having done this, it is possible that two or more pixels within r_{max} of one another had the same intensity and were both classified as maxima. If this is the case, we combine any such maxima into a single maximum, giving that maximum coordinates that are the average of the coordinates of all maxima identified within the r_{max} neighborhood. The end result will be that no maximum is within r_{max} of any other maximum.

We will denote the set of J maxima in image frame $\mathbf{F}_t^{(4)}$ by $\{(Px_{t,j}, Py_{t,j})\}$, $1 \leq j \leq J$. Figure 2.6 shows an image frame with the set of identified intensity maxima highlighted.

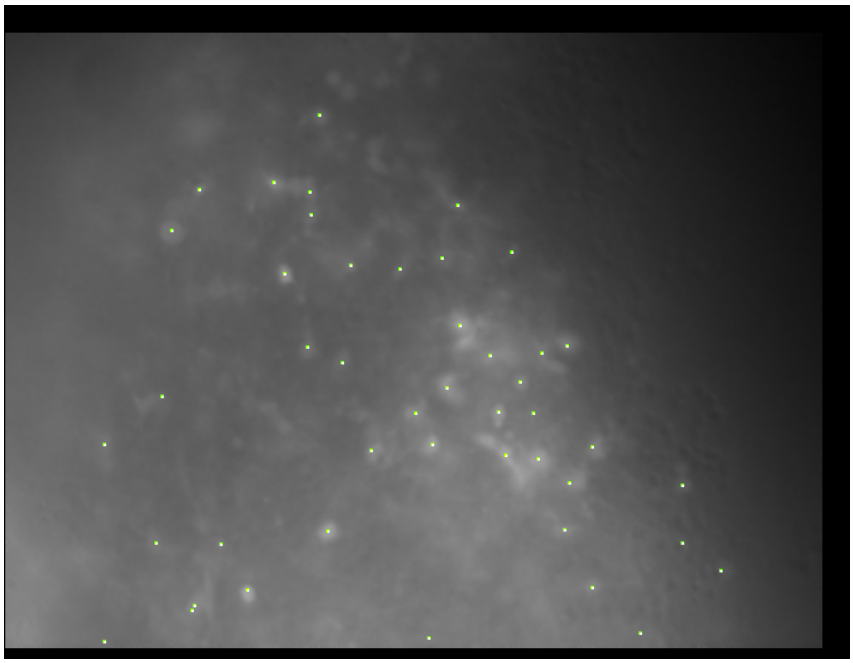

Figure 2.6: The set of local intensity maxima identified in an image frame.

2.2.6 Trajectory extraction

Now that we have a set of identified maxima, we can assemble them into cell trajectories. To do this, we compare the sets of maxima from a given image frame with those in the subsequent image frame, looking for the maximum from the first frame that is nearest a maximum from the second frame, if any maxima in the first frame are within r_{max} of any maxima in the second frame. If such a pairing is found, those two maxima are considered as part of a trajectory, and are removed from consideration. This process is repeated until there are no maxima in the first frame within r_{max} of

any in the second frame.

For a given pair of image frames $\mathbf{F}_t^{(4)}$ and $\mathbf{F}_{t+1}^{(4)}$, denote the set of K_t identified pairings defined above by

$$S_{t,t+1} = \{s_{t,1}, ..., s_{t,K_t}\},$$

where

$$s_{t,k} = [(Px_{t,a_k}, Py_{t,a_k}), (Px_{t+1,b_k}, Py_{t+1,b_k})].$$

Any maxima that do not fall into one of these pairings are discarded.

Trajectories are assembled by chaining these pairings together. If the second maximum of any pairing is the first maximum of a pairing in the subsequent frame, the two pairings are combined into a sequence of three points, and so forth, until all pairings that share a common maximum have been collected into the longest possible sequences of maxima.

The result will be a set of trajectories T_i, each of which consisting of a sequence of maxima that span some range of time points,

$$T_i = \{(Px_{t_i,a_i}, Py_{t_i,a_i}), (Px_{t_i+1,b_i}, Py_{t_i+1,b_i}), ..., (Px_{t_i+l_i,z_i}, Py_{t_i+l_i,z_i})\},$$

where l_i is the length of trajectory T_i, and t_i is the time point where the trajectory began.

At the end of this process, we cull any trajectories that are too small to analyze. The cull criteria can include a minimum length of trajectory needed to obtain valid statistical information on its behavior, or a lower limit on the amount of total motion we require in order to make a trajectory worth analyzing. We found 10 steps to be a reasonable lower limit for trajectory length, and culled any trajectories that never moved more than $r_{max}/3$ from their starting point.

2.2.7 Local motion compensation

Given the set of trajectories, this identifies the local areas within the sequence of image frames that need to be analyzed for local motion (shrinkage or shape changes in the tissue). We wish to analyze cell motion *relative to its surrounding tissue*, and so we need to compensate for this tissue movement without losing the cell motion in the process.

We do this by considering the pairwise steps of a trajectory between frames (the $s_{t,k}$ from which trajectories are assembled above). For each such pairing, we perform exactly the same motion compensation analysis that was done to compensate for global motion, but in this case we use a smaller region (a rectangle centered at the maximum position in each frame, with edge length one twelfth of the total image size in each axis), and we ignore all points within r_{max} of the maxima. That is, we disregard the cell motion when correlating the tissue regions. The vector that gives the lowest MSE represents the motion of the tissue surrounding the cell, independent of the cell motion.

For a given trajectory T_i consisting of l_i steps, this will generate $l_i - 1$ tissue motion vectors $(\delta x_{t_j}, \delta y_{t_j})$, $t_i + 1 \leq j \leq t_i + l_i$. The corrected cell trajectory, $\widehat{T_i}$, is composed of the points

$$\widehat{T_i} = \left\{ \left(Px_{t_i,a_i}, Py_{t_i,a_i} \right), \left(Px_{t_i+1,b_i} - \delta x_{t_i+1}, Py_{t_i+1,b_i} - \delta y_{t_i+1} \right), ..., \right.$$
$$\left. \left(Px_{t_i+l_i,z_i} - \sum_{k=i+1}^{l_i} \delta x_{t_i+k}, Py_{t_i+l_i,z_i} - \sum_{k=i+1}^{l_i} \delta y_{t_i+k} \right) \right\}.$$

This notation is cumbersome at best, so at this point we relabel the points in the corrected trajectories, retaining the starting point and starting time index for

correlation with position and time during the analysis. In what follows, we denote a corrected trajectory as

$$\mathcal{T}_i = \{(x_{i,j}, y_{i,j})\}\,,\ 1 \le j \le l_i \qquad \text{where} \qquad (x_{i,1}, y_{i,1}) = (0,0)\,.$$

We denote the initial point of $\mathcal{T}_i$ as (x_i^*, y_i^*), and the starting time point of the trajectory as t_i^*.

2.2.8 Cell trajectory analysis

For each corrected trajectory $\mathcal{T}_i = \{(x_{i,j}, y_{i,j})\}$, we compute the following values

$$\langle x \rangle_j = \frac{1}{j}\sum_{k=1}^{j}(x_{i,j} - x_i^*) \qquad \text{and} \qquad \langle y \rangle_j = \frac{1}{j}\sum_{k=1}^{j}(y_{i,j} - y_i^*)\,,$$

$$d_j = \sqrt{(x_{i,j} - x_i^*)^2 + (y_{i,j} - y_i^*)^2} \qquad \text{and} \qquad \langle d_j \rangle = \frac{1}{j}\sum_{k=1}^{j} d_j\,,$$

and

$$\left\langle (d_j - \langle d_j \rangle)^2 \right\rangle = \frac{1}{j}\sum_{k=1}^{j} \left(d_j - \langle d_j \rangle\right)^2\,.$$

We then perform a least-squares regression to find the best-fit curve of the form

$$y = 4Kj^{\alpha}$$

to the data series $\{\langle (d_j - \langle d_j \rangle)^2 \rangle\}$, $1 \le j \le l_i$. This function represents a diffusion model of the cell's motion, where K is the diffusion constant (representing how easily the cell can move in the surrounding tissue), and α characterizes the diffusion of the cell, where $\alpha < 1$ represents *subdiffusive* motion, $\alpha = 1$ represents normal *diffusive* motion, and $\alpha > 1$ indicates *superdiffusive* behavior.

By correlating the diffusion constant K, diffusion exponent α, along with trajectory length, average speed, average direction, and total distance covered with cell

position in tissue, and with the presence or absence of treatment reagents, we can extract a substantial amount of information from a series of microscopic image frames.

2.3 Automated analysis tool

The author's implementation of the automated analysis described in the previous section is a Java-based system that uses a general concept of a series of *filters* which take their input data from a *pipe* and which add output data to that pipe for downstream filters to use. A screen image from the automated analysis tool is shown in Figure 2.7.

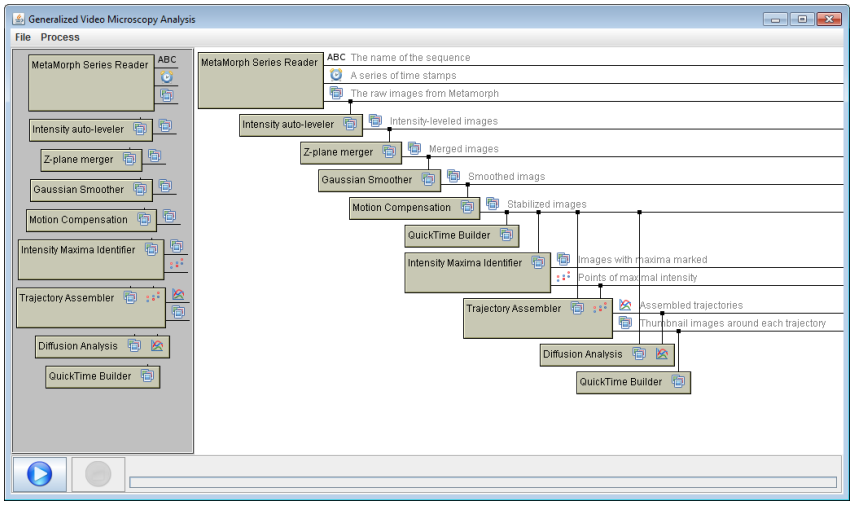

Figure 2.7: A typical screen view of the automated analysis tool. The list of available filters are shown on the left, and the assembled filter tree is on the right. Start and stop buttons at the bottom control the process, and a progress bar indicates status.

2.3.1 The data pipe

By a *pipe*, we mean a collection of named data objects of specified type. A data object can be anything from a simple text string to a complex data structure such as a series of images or a set of coordinate points. When a data object is added to a pipe, it becomes available to any filters that draw from that pipe. By naming each data object, the pipe can contain multiple objects of a given type. Filters retrieve objects from the pipe by name, then verify that the object is of the expected data type.

The pipe object has the ability to store itself to a file system, and to load itself from a previously stored state. After each filter in the analysis is executed, the pipe is stored to the hard drive. In this way, if an automated process is interrupted, it can reload its prior state and resume with the last filter that did not complete, rather than executing all filters again. An executor object handles loading of stored pipes and executing each filter in turn based on a comparison of the data objects that filter requires and the data objects currently available in the pipe.

2.3.2 The filter set

The analysis application described here includes a set of nine filters, used for various stages of processing. Before the first filter is executed, the user is prompted for a working directory in which to load and store files, and any existing pipe information in that directory is loaded to allow interrupted analyses to continue. It should be noted that the filter set listed here is by no means exhaustive, and the application supports the development and integration of new filters as needed to extend the

analysis.

MetaMorph file reader filter This filter is typically the first filter in the filter chain. It searches for MetaMorph image sets (indicated by the presence of files with ".nd" file extensions), and presents a list of the image sets it finds for the user to choose from. It then reads the TIF files generated by the MetaMorph program that form the selected image set. Each TIF file may contain several images, representing the various Z planes captured at a single time point, or may contain several time points in a single file. The outputs of this filter are the name of the selected image set; an array of images, with each column representing a time point, and each row representing a focal plane; and a series of time stamps representing the actual times each time point was captured.

Intensity leveling filter This filter implements the intensity leveling algorithm described in the prior section. It takes as its input the raw images that the MetaMorph file reader filter generates and produces as its output an array of the intensity-leveled images, with the same dimensions as the input array

Z-plane merger filter This filter merges several focal planes into a single image, by selecting the highest intensity value across all focal planes for each pixel, and constructing the resulting image. The input to this filter is an image array of arbitrary size. The output is an image array with a single row and as many columns as the input array.

Gaussian smoother filter This filter smooths the images in an image array so subsequent detection of maxima will identify the locations of objects in the frame accurately. It convolves a Gaussian kernel with the input images to produced smoothed images. The input of the filter is an image array of arbitrary size, and the output is a new image array of the same dimensions with the smoothed images.

Global motion compensation filter Global motion compensation attempts to stabilize the center portion of an image array by correlating each frame with the prior frame, and maximizing that correlation over some transformation of the second frame. In the present implementation, only translation transformations are considered, so the filter can compensate for linear displacements of the sample on the stage during image acquisition. Rotational transformations will be added in the future to correct for rotations of the sample on the stage as well. The input to this filter is an image array of arbitrary dimension, and the output is a new image array of the same dimension but whose images are transformed in such a way as to stabilize the central portion of the images throughout the sequence. The size of the images in the output image array are larger by an appropriate amount so that transforming the input images does not truncate any of the original image data.

Local maxima identification filter This filter identifies local maxima in an image array, producing a point-set array as its output. A point set array is an array of the same dimensions as the input image array, but where each array element is a set of points in the image frame representing the coordinates of he maxima. This filter also examines the ambient region surrounding each point, correlating it with

the same region in the subsequent frame using the same techniques that the global motion compensation filter used. The result is an ambient tissue velocity for the region surrounding each cell that can be used to determine cell motion relative to surrounding tissue, rather than relative to the sample as a whole.

Trajectory assembler filter Trajectory assembly examines the point set array and ambient tissue velocities and attempts to determine which maxima in adjacent frames represent the same object. Having done this, it assembles trajectories across as many frames as possible. The output of this process is a list of trajectories with actual coordinates and compensated coordinates that factor out the ambient tissue motion. This filter also produces a series of images showing the region surrounding a trajectory with the cell's position in the region for each frame. These thumbnail images can then be assembled by the QuickTime filter (described below) to generate movies of individual cells traveling in tissue.

Diffusion analysis filter The diffusion analysis takes assembled trajectories and computes the mean squared displacement of each cell from its starting point over time. It then fits this data to a diffusion model of the form $y = 4Kt^{\alpha}$ using a least-squares algorithm, generating a diffusion constant K and exponent α that characterized cell motion as subdiffusive, diffusive, or superdiffusive. The output of this filter is a set of Excel spreadsheets with the detailed calculations and results, a series of graphs of mean squared distances for each trajectory and the associated best-fit diffusion curve, and a series of visualizations that divide the tissue into a Voronoi diagram around each cell location, then color the regions in this diagram based on parameter

values, including mean cell speed, distance traveled, diffusion constant, and diffusion exponent.

QuickTime movie creation filter The final filter in the set is a QuickTime movie generator. This filter takes an arbitrary image array as its input and produces a QuickTime movie from each row in that image array. This filter can be attached at one or more points in the filter tree to create helpful visualizations of any stage in the analysis process.

REFERENCES

[1] Stuart A. Tobet, Heather J. Walker, Marianne L. Seney, and Kwok W. Yu. Viewing cell movements in the developing neuroendocrine brain. *Integrative and Comparative Biology*, 43(6):794–801, 2003.

[2] S. Benoit. Personal web site. http://lamar.colostate.edu/∼sbenoit/.

CHAPTER 3

MESOSCALE MODELS OF CELLS AND CELL BEHAVIOR

3.1 Introduction

Most cells encountered in living organisms have complex internal structures, enclosed within a membrane. Such *eukaryotic* cells exhibit widely diverse morphology and behavior in living organisms. The ability of cells to move and form larger structures is essential to many biological processes, including reproduction, [1, 2], embryonic cell migration [3–5], immune response [6–9], wound healing [10, 11], axon formation [12–14], and others. Cells can migrate significant distances from their place of origin to their ultimate location in tissue, guided by chemical and mechanical signals [15, 16]. Understanding interactions of large aggregates of cells during tissue growth and organization is challenging due to the complex nature of the system and the difficulty of imaging migrating cells in vivo.

Mathematical models of cell aggregation date back to the works of Patlak (1953) [17] and Keller and Segel (1971) [18]. These works considered cells as isolated, moving, *active* particles that exhibit reaction to the environment by, for example, excret-

ing chemical into the surrounding media other particles can react to. The models then address the continuum description for the motion of these particles, relating cell densities to chemical concentration distributions rather than tracking individual cell positions or trajectories. However, such an idealized view of cells as infinitesimally small objects in space leads to the possibilities of infinitely high concentrations of cells [19–23]. Various ways of regularizing that singularity have been suggested, such as [24, 25], most of those relating to the very intuitive idea that for finite size particles, the density cannot reach values beyond some critical *dense packing*, so that the density evolution equation must be modified to include this property. In addition, the dynamics of intercellular interaction changes under closed packing, which must be addressed as well. More recent work, including [26] and [27] model individual cells with local interactions and reproduce global behavior in reasonable agreement with observation.

The continuum models are powerful tools for the study of motion and aggregations of particles, including motile cells. However, under realistic conditions of organism development, motile cells not only form closely packed aggregates, but also considerably change their shape during the process. The change of shape of individual units in large aggregates under close packing is a phenomenon that, to our knowledge, has not been studied consistently. Of course, deformation of a cell could be addressed using a molecular approach to modeling the cell, but because of extreme complexity of even the most simple cells, such approach cannot describe collective dynamics of many cells. The goal of this chapter is to find a suitable "middle ground" of the course-graining of cellular structure, in order to accurately represent aggregation dynamics. A reasonable number of model elements for a simulation on a modern computer is

on the order of 10^8, if we assume short-range interactions between the elements as well as an efficient neighbor-finding methodology. The desire to study behavior in aggregates on the order of, say 10,000 cells, then, calls for models with around 10,000 model elements per cell that capture with sufficient fidelity cell behaviors critical to migration and guidance. The present work attempts to address this need by developing models for the principal components of cells and describing a framework in which these models can interact to simulate the desired aggregate systems.

Outline of the chapter

We assume the primary factors influencing cell aggregate behavior are cell membrane mechanics, cytoskeleton, adhesion, and signal transduction, and will model each component separately. We begin with some global constraints on component models that will facilitate their integration, then within each functional area, we present a component model with the objective of keeping total model element count over the aggregate within our target range of 10^8. We then describe a framework in which these models can be integrated to produce tissue-scale simulations, while allowing for new models of individual components to be added as they are developed. Finally, we provide results of simulations performed using our suite or component models and compare with observations from developing embryonic tissue. Our simulations were all performed in two dimensions, but we describe three-dimensional models in the text, which we plan to implement in future work.

3.1.1 Component Model Construction and Notation

Component models will be based on collections of interacting model elements. To facilitate efficient computation, only short-range forces will be considered. This removes electrostatic effects from consideration, a restriction that bears further study in the future. Moreover, we will consider all model elements to be points or spheres, and all body interactions in the model to be either contact, Hooke spring, Lennard-Jones, or soft sphere interactions. At first glance, this seems a very restrictive constraint, but as will be shown, a wide variety of structures can be represented with surprising fidelity under such limitations, and this restriction allows model elements from diverse component models to interact with a consistent and efficient mechanism. Within this system, model element are characterized by a center position and radius. We will use $\mathbf{p}_i$ to denote the position of the i^{th} model element, r_i to denote its radius, and (x_i, y_i, z_i) to denote its coordinates with respect to a standard right-handed orthonormal coordinate frame.

Given our stated target capability of modeling 10^4 cells, with roughly 10^4 model elements per cell, we divide this number among the individual systems as shown in Table 3.1 to inform model development in the sequel. An objective in component model design is that increasing element counts should model behavior more accurately, with the asymptotic limit of large element counts being a very good representation of the component structure.

Table 3.1: Approximate per-cell target element counts for component models.

Component model	Target element count
Cell Membrane	2,000
Cytoskeleton	6,000
Signal Transduction	1,000
Extra-Cellular Environment	1,000

3.2 Plasma membrane

The plasma membrane separates the interior (cytoplasmic) domain and exterior (extracellular) domain. It consists of a bilayer of phospholipids in which proteins are embedded at irregular intervals [28], and at concentrations on the order of one protein to fifty lipids [29]. Individual lipids are in an ordered liquid or gel-like state in which a lipid is free to move relative to its neighbors (which it does quite readily) or move to the other layer within the bilayer (which proceeds much more slowly) [30]. Cell membranes act as elastic media, and there are many factors in the composition of the membrane, proteins that span the membrane, and interaction of the membrane with the cytoskeleton that can lead to shape and curvature changes [31].

The cell membrane is not a static surface, but dynamically forms a variety of structures [30,32,33]. These structures are not well understood, but proposed organizational units include shells (rings of cholesterol and sphingolipid surrounding proteins in the membrane, much like a hydration shell around ions in aqueous solution [34]), clusters (groups of a few proteins with their associated shells [29]), caveolae (invagi-

nations of the membrane coated with caveolin [34]), nanodomains and rafts (regions with larger extent than a cluster that maintain stability for longer time periods than the membrane in general [29]), and pores (transient openings in the membrane [33]). In addition, the membrane may be able to buckle and fold in response to local stimuli [33]. A variety of techniques have been used to model lipid membranes, a survey of which can be found in [35–38], but none of which the author is aware provide a model in the range of our stated target model element count.

Typical values of key mechanical properties of plasma membranes are shown in Table 3.2.

Table 3.2: Empirical mechanical properties of plasma membranes.

Property	Typical Value	Citation
Elastic modulus (K_c)	7×10^{-20} J	[39]
Tension (T)	3×10^{-5} N/m	[40]
Area per lipid	5.89×10^{-19} m^2	[41]
Bilayer thickness	4.45×10^{-9} m	[41]

3.2.1 Cells and organelles as domains

We can view a cell, organelle or vesicle as an open simply connected domain $\mathcal{C}$ in $\mathbb{R}^N$ whose boundary $\Omega_{\mathcal{C}}$ is a smooth manifold of dimension $N-1$. Domains of distinct cells may not overlap, nor may domains of distinct organelles or vesicles within a given cell. We expect such domains to have fixed, or at least slowly varying, interior volume, but it has been shown that membrane surface area can change quickly in

response to changes in cell shape, so we do not constrain surface area [42].

For the purpose of designing a model for discrete simulation, we will represent the boundary $\Omega_{\mathcal{C}}$ of $\mathcal{C}$ by a triangulated mesh consisting of faces $\mathcal{F} = \{F_i\}$, $i \in \{1, ..., N_F\}$. We denote the set of vertex points in such a triangulation as $\mathcal{P} = \{\mathbf{p}_j\}$, $j \in \{1, ..., N_P\}$. To simplify what follows, we consider an edge as pair of oppositely directed edges between two vertex points. Attaching an orientation to an edge allows the boundary of a triangle to be specified as an ordered sequence of edges in such a way that its outward-facing side can be determined.

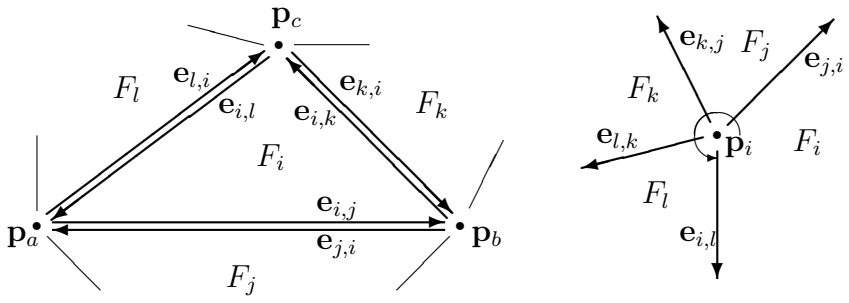

Figure 3.1: The set of faces, edges, and vertex points that make up a triangulated mesh representation of the boundary of a domain, as viewed from the exterior of the domain.

As shown in Figure 3.1, we label an edge $\mathbf{e}_{i,j}$ based on the two faces it connects, with the ordering of the indexes such that the left-hand face is listed first when viewed from the exterior of the domain. Under this convention, if we have a face F_i consisting of edges $(\mathbf{e}_{i,j}, \mathbf{e}_{i,k}, \mathbf{e}_{i,j})$ then the outward pointing normal vector $\hat{\mathbf{n}}_i$ of f_i is given by

$$\hat{\mathbf{n}}_i = \frac{\mathbf{e}_{i,j} \times \mathbf{e}_{j,k}}{|\mathbf{e}_{i,j} \times \mathbf{e}_{j,k}|} \,. \tag{3.1}$$

We denote the list of edges leaving a vertex point $\mathbf{p}_i$, in counter-clockwise order as seen from the exterior of the domain, by $\mathcal{E}_i = (\mathbf{e}_{j,i}, \mathbf{e}_{k,j}, ..., \mathbf{e}_{i,l})$, and we will at times

refer to this list of edges by a single index, as in $\mathcal{E}_i = (\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_m)$, when doing so makes the exposition more clear.

If we restrict the domains that represent cells to those whose surface is a genus 0 manifold, with Euler characteristic 2, then the number of vertex points is related to the number N_F of faces by

$$N_P = 2 + \frac{N_F}{2} .$$

Given a target model element (vertex point) count N_P for a cell or organelle membrane, then, we compute the number of faces in the triangulation as $N_F = 2(N_P - 2)$. In practice, we algorithmically construct a domain in some initial state using this fixed number of faces. The number of faces in the model may change as the system evolves, however, as explained below.

We will make use of the area A_i of face F_i, which is computed using any two of the edges that make up the boundary of the face, say $\mathbf{e}_{i,j}$ and $\mathbf{e}_{k,i}$, as

$$A_i = \frac{1}{2} \left| \mathbf{e}_{i,j} \times \mathbf{e}_{k,i} \right| . \tag{3.2}$$

3.2.2 Maintaining validity of the triangulated mesh

During simulation of membrane dynamics, vertices move in response to forces, which causes changes in edge lengths and the shape and size of faces. The mesh representation of the membrane may require adjustment after each iteration in the evolution process to ensure that the mesh remains valid. Moreover, the structure of the mesh may provide an upper limit on the movement that vertices may undergo in each time step to prevent degeneracies in the triangulation. Appendix B describes an algorithm that can be applied after each iteration to ensure the model remains valid, and which

produces an upper limit on movement that can be used in the subsequent evolution step to prevent invalidating the mesh.

3.2.3 Energy Functional

Internal energy of a cell

Once we have defined the outer shape of the cell, let us turn our attention to the energy of the cell in a given configuration. The energy functional E, the gradient of which will drive evolution of the membrane, is given by

$$E = E^{(elem)} + E^{(pres)} + E^{(tens)} + E^{(curv)} + E^{(si)}, \tag{3.3}$$

whose terms represent, respectively, model element interaction energy, compression of the interior volume, surface tension, membrane curvature, and a term to prevent membrane self-intersection.

To model the interaction of the membrane with a set of model elements with positions $\{\mathbf{p}_j\}$ and radii r_j, for $j \in \{1, ..., M\}$, we use a Lennard-Jones potential

$$E^{(elem)} = K_{LJ} \sum_{i=1}^{N_p} \sum_{j=1}^{M} \left(\xi_{i,j}^{12} - 2\,\xi_{i,j}^{6}\right) \quad \text{where} \quad \xi_{i,j} = \frac{\varepsilon + r_j}{|\mathbf{p}_i - \mathbf{q}_j|}\,, \tag{3.4}$$

where the inner sum is taken over the set of model elements that are not part of the membrane, whose positions we denote by $\{\mathbf{q}_j\}$, $1 \le j \le M$. Note that this energy only describes inter-cellular interaction under close contact, and *does not* describe the long-range interaction between the cells due to, for example, electrostatic interactions.

The pressure term is based on the compression of the cell volume V (relative to its equilibrium volume V_0), and on the bulk modulus of cytoplasm, which we estimate as $K_b = 10^5$ Pa. Using a modulus closer to that of water, say 2×10^9 Pa, while more

realistic, generates very large forces in response to small movements in membrane elements. Our lower estimate preserves the essential incompressibility of the cell while bringing the magnitude of the pressure forces into alignment with other forces in the model.

$$E^{(pres)} = \frac{K_b}{2V_0}\left(V - V_0\right)^2. \tag{3.5}$$

We compute the cell volume of a given triangulation of the cell surface using the following lemma, whose proof we present in Appendix A,

Lemma 3.2.1. *Given a triangulation $\mathcal{F}$ of $\Omega_{\mathcal{C}}$ where $\mathcal{C}$ is a compact 2-manifold embedded in $\mathbb{R}^3$ whose vertices are $\{\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_n\}$, and where $\mathbf{q}_i = (x, y, z)$ with respect to some orthonormal basis $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$, if each triangle $f_i = (\mathbf{p}_{i_1}, \mathbf{p}_{i_2}, \mathbf{p}_{i_3})$ has outward-pointing normal vector $\hat{\mathbf{n}}_i$, then the volume V of the interior of $\mathcal{F}$ is given by*

$$V = \frac{1}{6}\sum_{f_i \in \mathcal{F}} \left|\left(\mathbf{p}_{i_2} - \mathbf{p}_{i_1}\right) \times \left(\mathbf{p}_{i_3} - \mathbf{p}_{i_1}\right)\right| \mathbf{p}_{i_1} \cdot \hat{\mathbf{n}}_i. \tag{3.6}$$

We formulate surface tension energy based on surface area, where the force acts to minimize surface area. Therefore,

$$E^{(tens)} = \frac{T}{2}\sum_{i=1}^{N_f} A_i\,, \tag{3.7}$$

where A_i is given by (3.2).

To compute curvature energy, if we denote the angle between face normals $\hat{\mathbf{n}}_i$ and $\hat{\mathbf{n}}_j$ by $\alpha_{i,j}$, which we can compute directly using $\alpha_{i,j} = \cos^{-1}\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_j$, then the energy contribution of edge $\mathbf{e}_{i,j}$ can be estimated by

$$E_{i,j}^{(curv)} = \frac{4K_c|\mathbf{e}_{i,j}|}{\varepsilon}\tan^2\frac{\alpha_{i,j}}{2} = \frac{4K_c|\mathbf{e}_{i,j}|}{\varepsilon}\frac{1 - \hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_j}{1 + \hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_j}\,,$$

where K_c is the elastic modulus of the membrane. Summing over all edges associated with each vertex point, and dividing by two since this counts each edge twice,

$$E^{(curv)} = \frac{2K_c}{\varepsilon} \sum_{i=1}^{N_p} \sum_{\mathbf{e}_{j,k} \in \mathcal{E}_i} |\mathbf{e}_{j,k}| \frac{1 - \hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_k}{1 + \hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_k} \,. \tag{3.8}$$

Self-intersection in the membrane is prevented by placing a soft sphere of radius $\varepsilon/2$ at each vertex in the membrane model, but ignoring the interaction between spheres that lie on vertices connected by an edge. That is, an edge can shorten to accommodate membrane shape changes, but "distant" vertices will be prevented from getting close to one another. The energy term is given by,

$$E^{(si)} = K_{SS} \sum_{i=1}^{N_p} \sum_{j} \begin{cases} \left(\frac{\varepsilon}{|\mathbf{p}_j - \mathbf{p}_i|}\right)^{12} - 1 & |\mathbf{p}_j - \mathbf{p}_i| < \varepsilon \\ 0 & |\mathbf{p}_j - \mathbf{p}_i| \geq \varepsilon \end{cases} , \tag{3.9}$$

where K_{SS} is the soft-sphere interaction strength and the inner sum is taken over the set $\mathcal{V}$ of vertices excluding vertex $\mathbf{p}_i$ and any vertices that share an edge with $\mathbf{p}_i$.

Remark In practice, an efficient neighbor-finding algorithm will result in very few terms in the inner sums for both E_{elem} and E_{si}, since we can make ε a global upper limit for the radius of elements that interact via Lennard-Jones or soft sphere potentials, assuming we truncate our Lennard-Jones potentials at, say, $r = 2.5\varepsilon$.

3.2.4 Force Field

The force $\mathbf{F}_i$ on vertex $\mathbf{p}_i$ is given by $\mathbf{F}_i = -\nabla_i E$ where $\nabla_i = (\partial/\partial x_i, \partial/\partial y_i, \partial/\partial z_i)$. The force due to interactions with other (non-membrane) model components $\mathbf{q}_j$, $1 \leq j \leq M$ with radii r_j is given by

$$\mathbf{F}_i^{(elem)} = -12K_{LJ} \sum_{j=1}^{M} \left(\xi_{i,j}^{12} - \xi_{i,j}^{6}\right) \frac{\mathbf{q}_j - \mathbf{p}_i}{|\mathbf{q}_j - \mathbf{p}_i|^2} \quad \text{where} \quad \xi_{i,j} = \frac{\varepsilon + r_j}{|\mathbf{q}_j - \mathbf{p}_i|} \,. \tag{3.10}$$

When we consider the force due to pressure experienced by a single vertex point, recall that in (3.6), we are free to choose any starting edge and reference vertex to compute the contribution of a face to the volume. Therefore, we will assume that for all faces that a particular vertex point participates in, we use the vertex of interest as $\mathbf{p}_{i_1}$ in (3.6). If we index the edges in $\mathcal{E}_i$ as $(\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n)$, and make the identification $\mathbf{e}_{n+1} \equiv \mathbf{e}_1$, and denote the points that edge $\mathbf{e}_j$ connects by $(\mathbf{p}_i, \mathbf{q}_j)$, then

$$\mathbf{F}_i^{(pres)} = -\frac{K_b}{2V_0}\nabla_i \left(V - V_0\right)^2 \quad = -\frac{K_b}{6V_0}\left(V - V_0\right)\sum_{j=1}^{n}(\mathbf{q}_j \times \mathbf{q}_{j+1})\,. \tag{3.11}$$

The tension force on a vertex point $\mathbf{p}_i$ includes contributions from each face the vertex participates in. As with the pressure term, we are free to choose which edges we use to compute the area in (3.2), and so we choose those that meet at $\mathbf{p}_i$.

$$\mathbf{F}_i^{(tens)} = -\frac{T}{4}\sum_{j=1}^{n}\nabla_i|\mathbf{e}_j \times \mathbf{e}_{j+1}| \quad = -\frac{T}{4}\sum_{j=1}^{n}\hat{\mathbf{n}}_j \times (\mathbf{q}_{j+1} - \mathbf{q}_j)\,, \tag{3.12}$$

where $\hat{\mathbf{n}}_j$ is the outward-pointing unit normal vector to the face with edges $\mathbf{e}_j$ and $\mathbf{e}_{j+1}$.

When computing the curvature force on vertex $\mathbf{p}_i$, we must consider that the vertex position affects not only the edges directly connected to the vertex, but also the edges opposite the vertex in each face containing the vertex. The edge opposite the vertex in the face with edges $\mathbf{e}_j$ and $\mathbf{e}_{j+1}$ is given by $\mathbf{e}_{j+1} - \mathbf{e}_j$, and if we denote

the normal vector of the face that shares this edge by $\hat{\boldsymbol{\eta}}_j$, then

$$\begin{aligned}\mathbf{F}_i^{(curv)} &= -\frac{4K_c}{\varepsilon}\sum_{j=1}^{n}\nabla_i\left(|\mathbf{e}_{j+1}|\,\frac{1-\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}{1+\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}+|\mathbf{e}_{j+1}-\mathbf{e}_j|\,\frac{1-\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j}{1+\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j}\right)\\ &= -\frac{4K_c}{\varepsilon}\sum_{j=1}^{n}\left(\frac{1-\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}{1+\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}\nabla_i|\mathbf{e}_{j+1}|+\frac{1-\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j}{1+\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j}\nabla_i|\mathbf{e}_{j+1}-\mathbf{e}_j|\right.\\ &\qquad\left.-\frac{2\,|\mathbf{e}_{j+1}|}{(1+\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})^2}\nabla_i(\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})-\frac{2\,|\mathbf{e}_{j+1}-\mathbf{e}_j|}{(1+\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j)^2}\nabla_i(\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j)\right).\end{aligned} \tag{3.13}$$

Using $\nabla_i|\mathbf{e}_{j+1}| = -\mathbf{e}_{j+1}/|\mathbf{e}_{j+1}|$, $\nabla_i|\mathbf{e}_{j+1}-\mathbf{e}j| = 0$, this simplifies to

$$\begin{aligned}\mathbf{F}_i^{(curv)} = \frac{4K_c}{\varepsilon}\sum_{j=1}^{n}\left(\frac{1-\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}{1+\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}\frac{\mathbf{e}_{j+1}}{|\mathbf{e}_{j+1}|}+\frac{2\,|\mathbf{e}_{j+1}|}{(1+\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})^2}\nabla_i(\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})\right.\\ \left.+\frac{2\,|\mathbf{e}_{j+1}-\mathbf{e}_j|}{(1+\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j)^2}\nabla_i(\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j)\right).\end{aligned} \tag{3.14}$$

Now recall that

$$\begin{aligned}\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1} &= \frac{\mathbf{e}_j\times\mathbf{e}_{j+1}}{|\mathbf{e}_j\times\mathbf{e}_{j+1}|}\cdot\frac{\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}}{|\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|}\\ &= \frac{(\mathbf{e}_j\cdot\mathbf{e}_{j+1})(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+2})-(\mathbf{e}_j\cdot\mathbf{e}_{j+2})(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+1})}{|\mathbf{e}_j\times\mathbf{e}_{j+1}||\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|},\end{aligned}$$

and so

$$\begin{aligned}\nabla_i(\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}) &= \nabla_i\frac{(\mathbf{e}_j\cdot\mathbf{e}_{j+1})(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+2})-(\mathbf{e}_j\cdot\mathbf{e}_{j+2})(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+1})}{|\mathbf{e}_j\times\mathbf{e}_{j+1}||\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|}\\ &= \frac{(\mathbf{e}_j\cdot\mathbf{e}_{j+1})\nabla_i(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+2})+(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+2})\nabla_i(\mathbf{e}_j\cdot\mathbf{e}_{j+1})}{|\mathbf{e}_j\times\mathbf{e}_{j+1}||\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|}\\ &\quad-\frac{(\mathbf{e}_j\cdot\mathbf{e}_{j+2})\nabla_i(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+1})+(\mathbf{e}_{j+1}\cdot\mathbf{e}_{j+1})\nabla_i(\mathbf{e}_j\cdot\mathbf{e}_{j+2})}{|\mathbf{e}_j\times\mathbf{e}_{j+1}||\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|}\\ &\quad-(\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})\left(\frac{\nabla_i|\mathbf{e}_j\times\mathbf{e}_{j+1}|}{|\mathbf{e}_j\times\mathbf{e}_{j+1}|}+\frac{\nabla_i|\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|}{|\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|}\right).\end{aligned}$$

To simplify this, we use the facts that

$$\nabla_i|\mathbf{e}_j\times\mathbf{e}_{j+1}| = \frac{\mathbf{e}_j\times\mathbf{e}_{j+1}}{|\mathbf{e}_j\times\mathbf{e}_{j+1}|}\times(\mathbf{e}_{j+1}-\mathbf{e}_j) = \hat{\mathbf{n}}_j\times(\mathbf{e}_{j+1}-\mathbf{e}_j),$$

and

$$\nabla_i(\mathbf{e}_j\cdot\mathbf{e}_{j+1}) = -(\mathbf{e}_j+\mathbf{e}_{j+1}).$$

The result is

$$\nabla_i(\hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1}) = (\hat{\mathbf{n}}_{j+1} - (\hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1})\hat{\mathbf{n}}_j) \times \frac{\mathbf{e}_{j+1} - \mathbf{e}_j}{|\mathbf{e}_j \times \mathbf{e}_{j+1}|} + (\hat{\mathbf{n}}_j - (\hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1})\hat{\mathbf{n}}_{j+1}) \times \frac{\mathbf{e}_{j+2} - \mathbf{e}_{j+1}}{|\mathbf{e}_{j+1} \times \mathbf{e}_{j+2}|}\,.$$

Then using

$$\nabla_i(\hat{\mathbf{n}}_j \cdot \hat{\boldsymbol{\eta}}_j) = \nabla_i \left[\frac{\mathbf{e}_j \times \mathbf{e}_{j+1}}{|\mathbf{e}_j \times \mathbf{e}_{j+1}|} \cdot \hat{\boldsymbol{\eta}}_j \right] = \left(\hat{\boldsymbol{\eta}}_j - (\hat{\mathbf{n}}_j \cdot \hat{\boldsymbol{\eta}}_j)\hat{\mathbf{n}}_j\right) \times \frac{\mathbf{e}_{j+1} - \mathbf{e}_j}{|\mathbf{e}_j \times \mathbf{e}_{j+1}|}\,,$$

we are left with

$$\begin{aligned}\mathbf{F}_i^{(curv)} = \frac{4K_c}{\varepsilon} \sum_{j=1}^{n} \Bigg(& \frac{1 - \hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1}}{1 + \hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1}} \frac{\mathbf{e}_{j+1}}{|\mathbf{e}_{j+1}|} \\ & + \frac{2\,|\mathbf{e}_{j+1}|}{(1 + \hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1})^2} \Bigg[(\hat{\mathbf{n}}_{j+1} - (\hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1})\hat{\mathbf{n}}_j) \times \frac{\mathbf{e}_{j+1} - \mathbf{e}_j}{|\mathbf{e}_j \times \mathbf{e}_{j+1}|} \\ & \qquad + (\hat{\mathbf{n}}_j - (\hat{\mathbf{n}}_j \cdot \hat{\mathbf{n}}_{j+1})\hat{\mathbf{n}}_{j+1}) \times \frac{\mathbf{e}_{j+2} - \mathbf{e}_{j+1}}{|\mathbf{e}_{j+1} \times \mathbf{e}_{j+2}|} \Bigg] \\ & + \frac{2\,|\mathbf{e}_{j+1} - \mathbf{e}_j|}{(1 + \hat{\mathbf{n}}_j \cdot \hat{\boldsymbol{\eta}}_j)^2} \left(\hat{\boldsymbol{\eta}}_j - (\hat{\mathbf{n}}_j \cdot \hat{\boldsymbol{\eta}}_j)\hat{\mathbf{n}}_j\right) \times \frac{\mathbf{e}_{j+1} - \mathbf{e}_j}{|\mathbf{e}_j \times \mathbf{e}_{j+1}|} \Bigg)\,.\end{aligned} \tag{3.15}$$

Finally, the anti self-intersection force, which we adjust by $1/\varepsilon^2$ so it vanishes smoothly when $|\mathbf{p}_j - \mathbf{p}_i| = \varepsilon$,

$$\mathbf{F}_i^{(si)} = -12K_{SS} \sum_{\substack{j=1 \\ j \neq i}}^{N_p} \begin{cases} \frac{\varepsilon^{12}(\mathbf{p}_j - \mathbf{p}_i)}{|\mathbf{p}_j - \mathbf{p}_i|^{14}} - \frac{1}{\varepsilon^2} & |\mathbf{p}_j - \mathbf{p}_i| < \varepsilon \\ & \mathbf{p}_i, \mathbf{p}_j \text{ do not share an edge} \\ 0 & |\mathbf{p}_j - \mathbf{p}_i| \geq \varepsilon \text{ or} \\ & \mathbf{p}_i, \mathbf{p}_j \text{ share an edge} \end{cases}. \tag{3.16}$$

To summarize, the total force $\mathbf{F}_i$, then, on a given vertex point $\mathbf{p}_i$ is given by

$$\begin{aligned}
\mathbf{F}_i = &-12K_{LJ}\sum_{j=1}^{M}\left(\left(\frac{\varepsilon+r_j}{|\mathbf{q}_j-\mathbf{p}_i|}\right)^{12}-\left(\frac{\varepsilon+r_j}{|\mathbf{q}_j-\mathbf{p}_i|}\right)^{6}\right)\frac{\mathbf{q}_j-\mathbf{p}_i}{|\mathbf{q}_j-\mathbf{p}_i|^2}\\
&-\frac{K_b}{6V_0}\left(V-V_0\right)\sum_{j=1}^{n}(\mathbf{q}_j\times\mathbf{q}_{j+1})-\frac{T}{4}\sum_{j=1}^{n}\hat{\mathbf{n}}_j\times(\mathbf{q}_{j+1}-\mathbf{q}_j)\\
&+\frac{4K_c}{\varepsilon}\sum_{j=1}^{n}\Bigg(\frac{1-\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}{1+\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1}}\frac{\mathbf{e}_{j+1}}{|\mathbf{e}_{j+1}|}\\
&\qquad+\frac{2\,|\mathbf{e}_{j+1}|}{(1+\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})^2}\Bigg[(\hat{\mathbf{n}}_{j+1}-(\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})\hat{\mathbf{n}}_j)\times\frac{\mathbf{e}_{j+1}-\mathbf{e}_j}{|\mathbf{e}_j\times\mathbf{e}_{j+1}|}\\
&\qquad\qquad+(\hat{\mathbf{n}}_j-(\hat{\mathbf{n}}_j\cdot\hat{\mathbf{n}}_{j+1})\hat{\mathbf{n}}_{j+1})\times\frac{\mathbf{e}_{j+2}-\mathbf{e}_{j+1}}{|\mathbf{e}_{j+1}\times\mathbf{e}_{j+2}|}\Bigg]\\
&\qquad+\frac{2\,|\mathbf{e}_{j+1}-\mathbf{e}_j|}{(1+\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j)^2}\left(\hat{\boldsymbol{\eta}}_j-(\hat{\mathbf{n}}_j\cdot\hat{\boldsymbol{\eta}}_j)\hat{\mathbf{n}}_j\right)\times\frac{\mathbf{e}_{j+1}-\mathbf{e}_j}{|\mathbf{e}_j\times\mathbf{e}_{j+1}|}\Bigg)\\
&-12K_{SS}\sum_{\substack{j=1\\ j\neq i}}^{N_p}\begin{cases}\frac{\varepsilon^{12}(\mathbf{p}_j-\mathbf{p}_i)}{|\mathbf{p}_j-\mathbf{p}_i|^{14}}-\frac{1}{\varepsilon^2} & |\mathbf{p}_j-\mathbf{p}_i|<\varepsilon,\\ & \mathbf{p}_i,\mathbf{p}_j \text{ do not share an edge}\\ 0 & |\mathbf{p}_j-\mathbf{p}_i|\geq\varepsilon \text{ or}\\ & \mathbf{p}_i,\mathbf{p}_j \text{ share an edge}\end{cases}.
\end{aligned}\tag{3.17}$$

3.2.5 Two-dimensional approximation

In many cases, we wish to perform simulations in two dimensions, for simplicity. The method we adopt for this is to consider the cell to be a cylinder of thickness ε with vertical walls, as shown in Figure 3.2.5. In this scenario, the perimeter is divided into rectangles, which can be further divided into triangles if desired, but such subdivision is unnecessary because each rectangle is planar. We suppose that the top and bottom surface are fixed, hence force computations are irrelevant, and we limit our attention to the narrow edge.

If we consider the cell to be lying on the $x-y$ plane, and we number the vertical

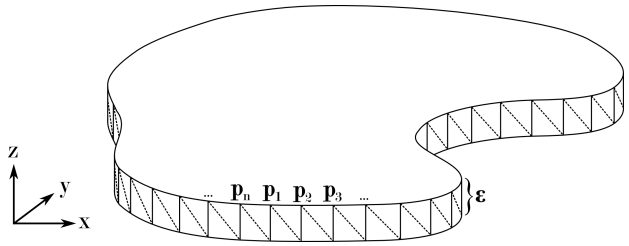

Figure 3.2: A method for generating a two-dimensional approximation of a cell for simplified simulations.

edges of the rectangles that form its boundary as $\mathbf{p}_i$, $i \in \{1, ..., N_p\}$, proceeding in a counterclockwise direction as viewed from a point with positive z coordinate, as shown in Figure 3.2.5.

Under this two dimensional simplification, (3.4), (3.5), (3.7), and (3.9) remain valid, but we compute surface area A_i of a perimeter rectangle using $A_i = \varepsilon\,|\mathbf{p}_{i+1} - \mathbf{p}_i|$, and compute cell volume V using

$$V = \frac{\varepsilon}{2} \sum_{i=1}^{N_p} \left(x_i y_{i+1} - y_i x_{i+1} \right) ,$$

where the two-dimensional coordinates of $\mathbf{p}_i$ are (x_i, y_i).

The curvature energy simplifies slightly due to the constrained geometry,

$$E^{(curv)} = 4K_c \sum_{i=1}^{N_p} \frac{1 - \hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_{i+1}}{1 + \hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_{i+1}} \,. \tag{3.18}$$

In the above, we have made use of the identifications $\hat{\mathbf{n}}_{N_p+1} \equiv \hat{\mathbf{n}}_1$ and $\mathbf{p}_{N_p+1} \equiv \mathbf{p}_1$ to allow the sums to close the boundary.

3.2.6 Force in the two-dimensional approximation

Under the constrained geometry of the two-dimensional approximation discussed in 3.2.5, the force simplifies somewhat. Only the curvature term presents some complex-

ity. If we use $\mathbf{e}_i$ to denote edge $\mathbf{p}_{i+1} - \mathbf{p}_i$, and let $\hat{\mathbf{e}}_i = \mathbf{e}_i/|\mathbf{e}_i|$, then $\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_{i+1} = \hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1}$, and so

$$F_i^{(curv)} = -4K_c \nabla_i \left[\frac{1 - \hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1}}{1 + \hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1}} + \frac{1 - \hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i}{1 + \hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i} + \frac{1 - \hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1}}{1 + \hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1}} \right],$$

where as before, vertex position affects curvature at the vertex and also at adjacent vertices, yielding three terms. Computing gradients,

$$F_i^{(curv)} = 8K_c \left[\frac{\nabla_i(\hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1})}{(1 + \hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1})^2} + \frac{\nabla_i(\hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i)}{(1 + \hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i)^2} + \frac{\nabla_i(\hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1})}{(1 + \hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1})^2} \right].$$

The gradients of the dot products are given by

$$\begin{aligned}
\nabla_i(\hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1}) &= \frac{(\hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1})\hat{\mathbf{e}}_i - \hat{\mathbf{e}}_{i+1}}{|\mathbf{e}_i|}, \\
\nabla_i(\hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i) &= \frac{\mathbf{e}_i - \mathbf{e}_{i-1} - (\hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i)[|\mathbf{e}_i|\hat{\mathbf{e}}_{i-1} - |\mathbf{e}_{i-1}|\hat{\mathbf{e}}_i]}{|\mathbf{e}_{i-1}||\mathbf{e}_i|}, \text{ and} \\
\nabla_i(\hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1}) &= \frac{\hat{\mathbf{e}}_{i-2} - (\hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1})\hat{\mathbf{e}}_{i-1}}{|\mathbf{e}_{i-1}|}.
\end{aligned}$$

Applying these, we obtain

$$\begin{aligned}
F_i^{(curv)} = 8K_c \Bigg[&\frac{(\hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1})\hat{\mathbf{e}}_i - \hat{\mathbf{e}}_{i+1}}{|\mathbf{e}_i|(1 + \hat{\mathbf{e}}_i \cdot \hat{\mathbf{e}}_{i+1})^2} + \frac{\mathbf{e}_i - \mathbf{e}_{i-1} + (\hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i)(|\mathbf{e}_{i-1}|\hat{\mathbf{e}}_i - |\mathbf{e}_i|\hat{\mathbf{e}}_{i-1})}{|\mathbf{e}_{i-1}||\mathbf{e}_i|(1 + \hat{\mathbf{e}}_{i-1} \cdot \hat{\mathbf{e}}_i)^2} \\
&+ \frac{\hat{\mathbf{e}}_{i-2} - (\hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1})\hat{\mathbf{e}}_{i-1}}{|\mathbf{e}_{i-1}|(1 + \hat{\mathbf{e}}_{i-2} \cdot \hat{\mathbf{e}}_{i-1})^2} \Bigg],
\end{aligned} \tag{3.19}$$

and the complete two-dimensional analog to (3.17) is

$$\begin{aligned}
\mathbf{F}_i = &-12K_{LJ}\sum_{j=1}^{M}\left(\left(\frac{\varepsilon+r_j}{|\mathbf{q}_j-\mathbf{p}_i|}\right)^{12}-\left(\frac{\varepsilon+r_j}{|\mathbf{q}_j-\mathbf{p}_i|}\right)^{6}\right)\frac{\mathbf{q}_j-\mathbf{p}_i}{|\mathbf{q}_j-\mathbf{p}_i|^2}\\
&-\frac{\varepsilon K_b}{2V_0}(V-V_0)\begin{bmatrix} y_{i+1}-y_{i-1} \\ x_{i-1}-x_{i+1}\end{bmatrix}-\frac{T\varepsilon}{2}(\mathbf{e}_{i-1}-\mathbf{e}_i)\\
&+8K_c\left[\frac{(\hat{\mathbf{e}}_i\cdot\hat{\mathbf{e}}_{i+1})\hat{\mathbf{e}}_i-\hat{\mathbf{e}}_{i+1}}{|\mathbf{e}_i|(1+\hat{\mathbf{e}}_i\cdot\hat{\mathbf{e}}_{i+1})^2}+\frac{\mathbf{e}_i-\mathbf{e}_{i-1}+(\hat{\mathbf{e}}_{i-1}\cdot\hat{\mathbf{e}}_i)(|\mathbf{e}_{i-1}|\hat{\mathbf{e}}_i-|\mathbf{e}_i|\hat{\mathbf{e}}_{i-1})}{|\mathbf{e}_{i-1}||\mathbf{e}_i|(1+\hat{\mathbf{e}}_{i-1}\cdot\hat{\mathbf{e}}_i)^2}\right.\\
&\qquad\left.+\frac{\hat{\mathbf{e}}_{i-2}-(\hat{\mathbf{e}}_{i-2}\cdot\hat{\mathbf{e}}_{i-1})\hat{\mathbf{e}}_{i-1}}{|\mathbf{e}_{i-1}|(1+\hat{\mathbf{e}}_{i-2}\cdot\hat{\mathbf{e}}_{i-1})^2}\right]\\
&-12K_{SS}\sum_{\substack{j=1\\ j\neq i}}^{N_p}\begin{cases}\frac{\varepsilon^{12}(\mathbf{p}_j-\mathbf{p}_i)}{|\mathbf{p}_j-\mathbf{p}_i|^{14}} & |\mathbf{p}_j-\mathbf{p}_i|<\varepsilon,\ \text{and } \mathbf{p}_i,\mathbf{p}_j \text{ not adjacent}\\ 0 & |\mathbf{p}_j-\mathbf{p}_i|\geq\varepsilon \text{ or } \mathbf{p}_i,\mathbf{p}_j \text{ adjacent}\end{cases}.
\end{aligned} \tag{3.20}$$

3.2.7 Simulation results

An example of simulated two-dimensional membrane behavior are shown in Figure 3.2.7. Here, a fixed boundary was placed around a circular membrane composed of 300 model elements, then a rigid circular probe under constant force was introduced and allowed to collide with the membrane. It can be seen that membrane volume is preserved, but surface area increases as the membrane deforms. Model elements are added as the perimeter expands to maintain a spacing between elements that falls within a target range.

3.3 Cytoskeleton and signal transduction

In the computation above, we have assumed that the internal energy of a cell under compression is due to the deformation of the membrane as well as the pressure of the cytoplasm. In reality, however, a cell possesses internal cytoskeleton giving it

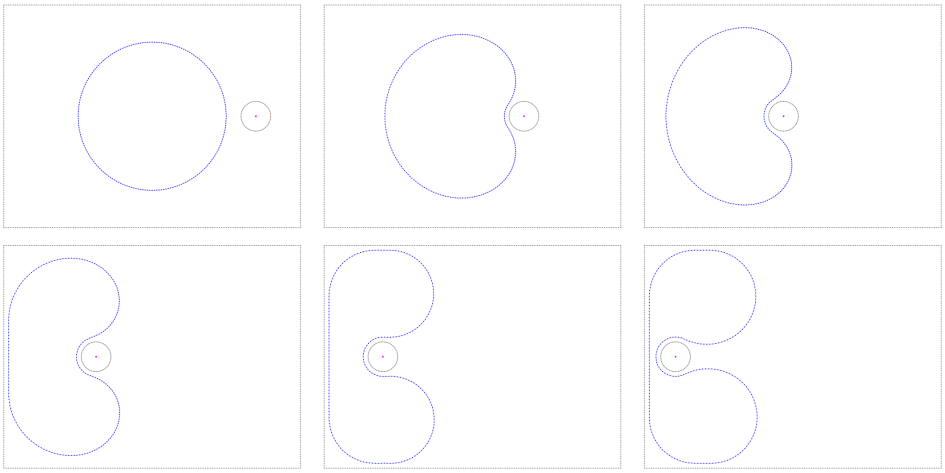

Figure 3.3: A simulation of a two-dimensional membrane reacting to a rigid probe under constant force.

additional rigidity. The purpose of this section is to model a cytoskeleton using a computationally reasonable number of elements.

Cell shape is modulated by an internal cytoskeleton that is constructed and dismantled dynamically within the cytoplasm [43, 44]. The cytoskeleton is a network of interlinked polymer filaments in the interior of a cell (if we disregard blebbing, where the membrane opens and cytoskeleton protrudes outside the cell). There are different types of polymers that make up the cytoskeleton, but the most abundant and most directly related to cell motion is actin. Actin filaments are very small (typically 6–8 nm in diameter [45]) and form a dense network that is concentrated near the cortex of the cell [46]. Actin filaments are linked to trans-membrane proteins in the plasma membrane, and cross-link among themselves to provide rigidity and shape to the cell. They polymerize and depolymerize continuously, with half-lives on the order of tens of minutes, and display asymmetric growth, preferring to add new monomers on one

end during polymerization and lose them from the other during depolymerization [45], a phenomenon known as *treadmilling*. During motility, the leading edge of the cell is characterized by formation of new microfilaments [47]. A number of proteins affect actin microfilament polymerization or depolymerization [48–50], sever existing microfilaments [51–54], or bind microfilaments together [55–57]. Other trans-membrane proteins bind to the actin cytoskeleton to promote adhesion.

The other primary constituents of the cytoskeletal framework are microtubules and intermediate filaments. Microtubules are the stiffest cytoskeletal structures, helical arrangements of the protein tubulin on the order of 25 nm in diameter [45], which form a highly dynamic network, with a typical lifetime on the order of a few minutes for a microtubule [58, 59]. The primary role of microtubules appears to be the transport of vesicles and organelles within a cell, but they also act to regulate actin behavior and play a role in polarizing a cell and determining its direction of motion [60–62]. In interphase (non-dividing) cells, most microtubules are attached to a centrosome near the cell nucleus and project outward toward the periphery of the cell [44, 46, 63]. During cell movement, the centrosome tends to move toward the cell's leading edge [64].

Intermediate filaments consist of polymers based on a variety of different proteins (keratins, desmin, GFAP, vimentin, and others) [65–67]. The proteins form coiled coils then tetramers which align into protofilaments, then aggregate into rope-like filaments on the order of 10nm in size [45]. The intermediate filament network is also a dynamic system, but changes in its structure are not correlated with cell movement as are those of actin microfilaments or microtubules [68, 69]. Rather, its role appears to be inter-cell communication and transport of proteins [70–72].

Some data taken from [45] regarding the mechanical properties of the various forms of cytoskeletal structures are included in Table 3.3.

Table 3.3: Mechanical properties of cytoskeletal filaments

	Diameter (nm)	Persistence Length (μm)	Bending Stiffness (Nm^2)	Young's Modulus (Pa)
Actin microfilament	6–8	15	7×10^{-26}	$1.3–2.5 \times 10^9$
Microtubule	25	6000	2.6×10^{-23}	1.9×10^9
Intermediate filament	10	~ 1	4×10^{-27}	1×10^9

3.3.1 Current Models of Cell Shape Modulation

Early models of cytoskeletal dynamics and its role in shape modulation were simple molecular-level hypotheses of how particular observed behaviors occurred [73–76]. Subsequent modeling efforts were more general, including lattice [77], kinetic [78–80], and continuum models [81]. Recent efforts have extended and improved on these works [82–84], but still model only a subset of cytoskeletal behavior (for example, actin network dynamics, microtubule dynamics, intermediate filament dynamics, or the dynamics of links with transmembrane adhesion proteins). At present, there is no unified model of cell behavior that integrates these disparate models, but some models generalize cell behavior as a whole without trying to account for contributions from these individual factors [85–88].

3.3.2 Modeling actin as a collection of linked spheres

We present a simple but effective model of cytoskeletal actin that exhibits robust chemotactic behavior. In the spirit of the component models described in the introduction, we model actin filaments in the cytoskeleton of a cell as a collection of spherical model elements connected by springs that interact with each other and with membrane model elements.

Consider a cell domain $\mathcal{C}$, with boundary membrane $\Omega_{\mathcal{C}}$, consisting of model elements as described in the prior section. We attach an actin filament to each membrane model element, with a number of linked spheres proportional to the length of the filament. This situation is shown for the two-dimensional case in Figure 3.3.2, but the three-dimensional case is analogous.

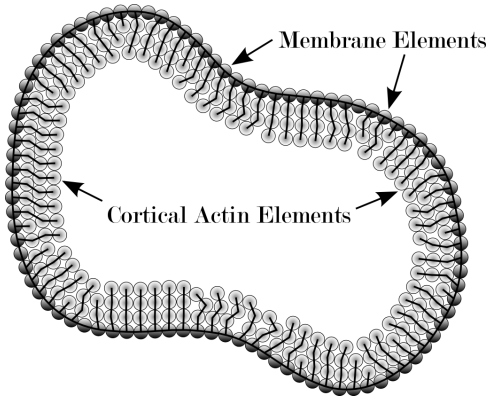

Figure 3.4: An initial arrangement of cortical actin model elements in a cell

We equip these spheres with a simple Lennard-Jones potential and a Hooke's law spring connection, with spring constant K_s, to either the adjacent spheres in the filament, or the membrane element in the case of the first sphere in a filament, and allow them to interact with each other and with membrane elements. The attraction

generated by this potential between nearby elements models the crosslinking of actin filaments within the cytoskeleton. This results in a level of mechanical stiffness in areas where the cortical actin elements are not simple monolayers. The degree of rigidity, provided by the Lennard-Jones attraction, can be controlled by the potential well depth K_{LJ}. If we denote the spheres of the the actin filament attached to membrane element $\mathbf{p}_i$ by $\{\mathbf{q}_{i,1}, ..., \mathbf{q}_{i,m_i}\}$ with radii $\{r_{i,1}, ..., r_{i,m_i}\}$, and assign a fixed radius R to membrane elements, the energy functional for cortical actin is given by

$$\begin{aligned} E =& K_{LJ} \sum_{i=1}^{n} \sum_{k \neq i} \left(\left(\frac{r_i + r_k}{|\mathbf{q}_k - \mathbf{q}_i|} \right)^{12} - 2 \left(\frac{r_i + r_k}{|\mathbf{q}_k - \mathbf{q}_i|} \right)^{6} \right) \\ &+ \frac{K_s}{2} \sum_{i=1}^{n} (|\mathbf{p}_i - \mathbf{q}_{i,1}| - (R + r_{i,1}))^2 \\ &+ \frac{K_s}{2} \sum_{i=1}^{n} \sum_{j=1}^{m_i - 1} \left(|\mathbf{q}_{i,j} - \mathbf{q}_{i,j+1}| - (r_{i,j} + r_{i,j+1}) \right)^2 , \end{aligned} \tag{3.21}$$

where the inner summation in the Lennard-Jones term is taken over other all actin elements within the same cell as well as the membrane elements in that cell. The force $\mathbf{F}_{i,1}$ on the leading actin element $\mathbf{q}_{i,1}$ in the i-th filament is given by

$$\begin{aligned} \mathbf{F}_{i,1} =& 12 K_{LJ} \sum_{j \neq i} \frac{\mathbf{q}_i - \mathbf{q}_j}{|\mathbf{q}_j - \mathbf{q}_i|^2} \left[\left(\frac{r_i + r_j}{|\mathbf{q}_j - \mathbf{q}_i|} \right)^6 - 1 \right] \left(\frac{r_i + r_j}{|\mathbf{q}_j - \mathbf{q}_i|} \right)^6 \\ &+ K_s (\mathbf{p}_i - \mathbf{q}_{i,1})(|\mathbf{p}_i - \mathbf{q}_{i,1}| - (R + r_{i,1})) \\ &+ K_s (\mathbf{q}_{i,2} - \mathbf{q}_{i,1})(|\mathbf{q}_{i,1} - \mathbf{q}_{i,2}| - (r_{i,1} + r_{i,2})) , \end{aligned} \tag{3.22}$$

where we disregard the last term if the filament contains only one element. The force on other actin elements $\mathbf{q}_{i,j}$ $j > 1$ in the filament is given by

$$\begin{aligned} \mathbf{F}_{i,j} =& 12 K_{LJ} \sum_{j \neq i} \frac{\mathbf{q}_i - \mathbf{q}_j}{|\mathbf{q}_j - \mathbf{q}_i|^2} \left[\left(\frac{r_i + r_j}{|\mathbf{q}_j - \mathbf{q}_i|} \right)^6 - 1 \right] \left(\frac{r_i + r_j}{|\mathbf{q}_j - \mathbf{q}_i|} \right)^6 \\ &+ K_s (\mathbf{q}_{i,j-1} - \mathbf{q}_{i,1})(|\mathbf{q}_{i,j} - \mathbf{q}_{i,j-1}| - (r_{i,j} + r_{i,j-1})) \\ &+ K_s (\mathbf{q}_{i,j+1} - \mathbf{q}_{i,1})(|\mathbf{q}_{i,j} - \mathbf{q}_{i,j+1}| - (r_{i,j} + r_{i,j+1})) , \end{aligned} \tag{3.23}$$

where here we disregard the last term if the element is the last in its filament. Where actin interacts via Lennard-Jones or spring forces with membrane elements we will, of course, make the appropriate additions of opposite forces to those membrane elements.

3.3.3 Signal transduction

There are two simple models of chemical signal we can adopt - a particle or a continuum model. In a particle model, a source of signal emits particles at a rate determined by its signaling strength, and those particles diffuse through the surrounding environment until they either decay naturally or encounter a receptor, which is then activated by the signal. In a continuum model, the source has a particular strength, and all receptors in the model are activated based on their distance from the source and on the decay curve of the signal over distance. The difficulty with a continuum model is that it does not take into account that our simulation space is neither isotropic nor homogeneous, and so direct distance computations inherently ignore critical phenomena. Such a system allows signals to diffuse through plasma membranes at the same rate as through ECM or cytoplasm, for example. To adjust the continuum model to factor in densities and diffusion constants for various regions makes the model difficult to implement. In a particle model, on the other hand, diffusion is controlled by local conditions, making the model very simple and facilitating distributed computation. However, this comes with the understanding that in order to accurately simulate continuous chemical diffusion, a large number of diffusing particles will be necessary.

Our model supports sources of chemical signal. A source has characteristics that

govern how fast it releases signal particles, and how much total signal it is allowed to release (perhaps infinite). It may have a signaling curve that describes signal levels emitted over time through the effective life of the source. For example, we can define a set of point sources with infinite lifetime and slow diffusion rate to create a constant gradient in a sample, or we could define a point source with fast release rate and very short lifetime to simulate the release of neurotransmitter from an axon. Once released, the signal diffuses through its surroundings under Brownian motion, or decays probabilistically with a given half-life.

3.3.4 Signal particle detection and activation

Model elements can act as receptors, transmitters, or both. For example, membrane elements (which carry embedded transmembrane proteins) act as receptors of extracellular signals, and transmitters of inter-cellular signals, providing a transduction mechanism that allows the cell to react to its local environment. When a signal particle encounters (comes within a threshold distance of) a receptor that is sensitive to that signal, the signal particle is consumed, and the receptor becomes "activated". Activation level is simply a signed number associated with the receptor, where receipt of a signal adds or subtracts some fixed value to the receptor's activation level. The receptor's response to activation levels is model-dependent. For example, a membrane model element may use its activation to allow ions to flow through a channel, in which case the receptor acts as a signal source to generate ions on the interior side of the membrane, with the number of ions the source can generate proportional to the activation level of the receptor. These ions are then received by nearby cortical

actin elements, which causes changes in polymerization levels, as described below.

3.3.5 Actin dynamics and treadmilling

Actin filaments respond to the activation levels of their parent membrane element by adjusting their polymerization. Within a cell, there exists a fixed pool of actin monomer, meaning that over time, the amount of polymerized actin should remain constant. During cell evolution, membrane elements detect attractant or repellent signals, and take on positive or negative activation levels, respectively. Where a membrane element has a positive activation level, we simulate polymerization of actin by adding or increasing the size of actin model elements in the filament attached to that membrane element. Filaments deploymerize in response to negative activation levels, in which case we shrink or remove elements from the filament. After processing membrane element signals, we apply a global adjustment to all actin filaments such that the total volume of actin elements (polymerized actin) in the cell is conserved.

More formally, if a cell consists of actin elements $\mathbf{q}_i$, $1 \leq i \leq n$, and element $\mathbf{q}_i$ has activation level $\alpha_i \in \mathbb{R}$, then the mean activation $\bar{\alpha}$ per element in the cell is given by

$$\bar{\alpha} = \frac{1}{n} \sum_{i=1}^{n} \alpha_i. \tag{3.24}$$

From this, we compute the volume change δV_i for the element as

$$\delta V_i = \frac{(\alpha_i - \bar{\alpha})}{\bar{\alpha}} V_i, \tag{3.25}$$

where V_i is the current volume of $\mathbf{q}_i$ (above average activation causes growth, below-average causes shrinkage).

Actin filaments preferentially polymerize at their barbed end, and depolymerize at their pointed end, where the barbed end is attached to the membrane, resulting in *treadmilling* as monomers are added at one end and slowly migrate to the other end as the polymer absorbs and releases monomers. We model this by processing polymerization events at the end of the filament that is attached to the membrane, and depolymerization events at the opposite (free) end. The process is illustrated in Figure 3.3.5. We first designate a maximum size for actin model elements. To poly-

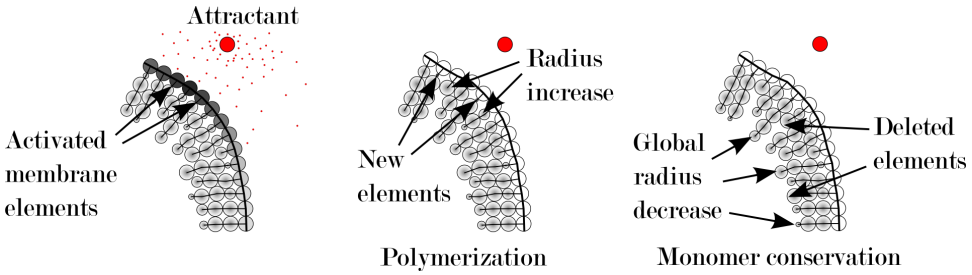

Figure 3.5: Signal transduction and actin dynamics

merize a filament, we either enlarge the leading (membrane-attached) actin element, or if that element is already at the maximum size, we create a new, small element and insert it between the membrane and the first element of the filament. When depolymerizing, we decrease the size of the trailing (free end) actin element, and when that size reaches zero, delete the element. In this way, filaments grow and shrink smoothly, exhibiting the expected treadmilling effects. In regions of positive membrane activation, polymerization of filaments at the membrane-attached end causes an outward force that projects the cell in the direction of the attractive signal source, generating a chemotactic response. In regions of negative activation, the actin depolymerization results in reduced rigidity, which allows the trailing edge of the cell to shift forward

and follow the advancing leading edge while preserving cell volume.

Simulation results

Results from this simple model of cytoskeletal dynamics reveal that it captures many of the behaviors of actin networks that are essential to cell motility. Figure 3.3.5 shows the results of a simulation of a cell with cortical actin cytoskeleton, in the presence of a diffusing chemical attractant. The cell consisted of 200 membrane model elements and 800 actin elements. Cell polarization and the formation of lamellipodia and resulting chemotaxis is clearly visible.

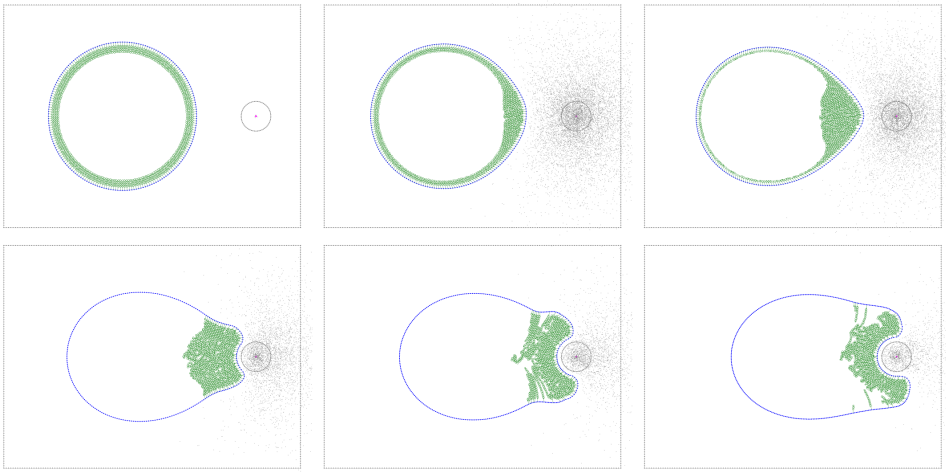

Figure 3.6: Simulation of cells with cortical actin cytoskeleton, and source of diffusing chemical attractant.

In Figure 3.3.5, we allow the source of attractant signal to move at a fixed rate, causing the cell to pursue the attractant.

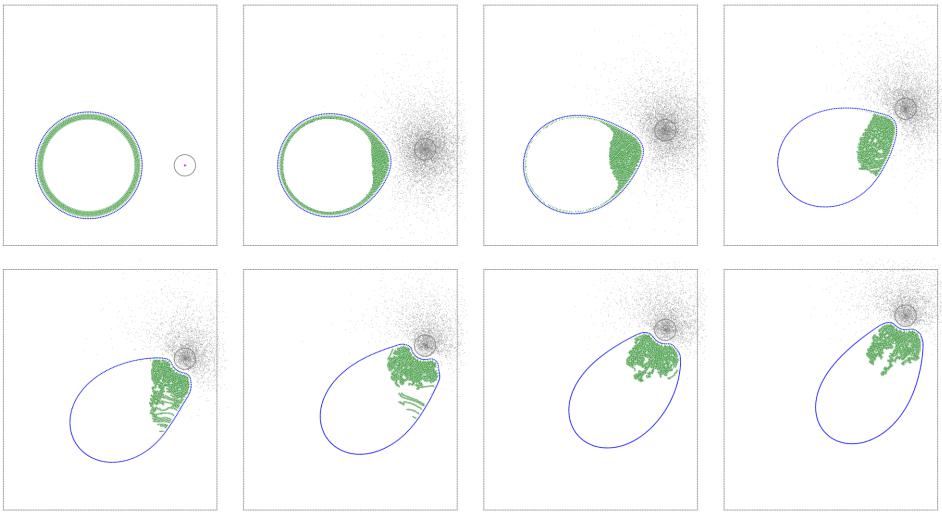

Figure 3.7: Simulation of a cell pursuing a diffusing chemical attractant under constant motion.

3.4 Conclusions

While the accurate modeling of cells and cell behavior remains one of the most challenging problems facing computational biology, new models continue to appear that capture with increasing fidelity the diverse behavior of cells and cell aggregates. Our mesoscale models show that at scales of a few thousand model elements per cell, behavior that includes cell polarization, lamellipodia formation, and chemotaxis can be demonstrated. Further refinement in these models, guided by empirical observations and measurements, will only improve their accuracy.

Certainly, the models presented in the present work represent a simple foray into meso-scale discrete representations of cells and cell structures, but we hope that our generalized framework for constructing and integrating models of diverse cellular systems can serve as a foundation for further model development and refinement.

The complete source code for the authors' implementation of this framework (with which all examples in this chapter were constructed) is included in the Appendix and is freely available on the author's web page [89].

REFERENCES

[1] Georgina Fabro, Roberto A. Rovasio, Silvia Civalero, Anat Frenkel, S. Roy Caplan, Michael Eisenbach, and Laura C. Giojalas. Chemotaxis of capacitated rabbit spermatozoa to follicular fluid revealed by a novel directionality–based assay. *Biol Reprod*, 67(5):1565–1571, 2002.

[2] U. Benjamin Kaupp, Nachiket D. Kashikar, and Ingo Weyand. Mechanisms of sperm chemotaxis. *Annu Rev Physiol*, 70:93–117, March 2008.

[3] T. N. Behar, A. E. Schaffner, C. A. Colton, R. Somogyi, Z. Olah, C. Lehel, and J. L. Barker. GABA–induced chemokinesis and NGF–induced chemotaxis of embryonic spinal cord neurons. *J Neurosci*, 14(1):29–38, January 1994.

[4] T. N. Behar, M. M. Dugich–Djordjevic, Y–X. Li, W. Ma, R. Somogyi, X. Wen, E. Brown, C. Scott, R. D. G. McKay, and J. L. Barker. Neurotrophins stimulate chemotaxis of embryonic cortical neurons. *Eur J Neurosci*, 9(12):2561–2570, April 2006.

[5] Ray Keller. Cell migration during gastrulation. *Curr Opin Cell Biol*, 17(5):533–541, October 2005.

[6] Andrew D. Luster. Chemokines —chemotactic cytokines that mediate inflammation. *New Engl J Med*, 338(7):436–445, February 1998.

[7] Joost J. Oppenheim and De Yang. Alarmins: chemotactic activators of immune responses. *Curr Opin Immunol*, 17(4):359–365, August 2005.

[8] Andrew D. Luster, Ronen Alon, and Ulrich von Andrian. Immune cell migration in inflammation: present and future therapeutic targets. *Nat Immunol*, 6(12):1182–1190, December 2005.

[9] Jonathan R. Mathias, Benjamin J. Perrin, Ting–Xi Liu, John Kanki, A. Thomas Look, and Anna Huttenlocher. Resolution of inflammation by retrograde chemotaxis of neutrophils in transgenic zebrafish. *J Leukocyte Biol*, 80:1281–1288, September 2006.

[10] Richard A. F. Clark. *The molecular and cellular biology of wound repair*. Plenum Press, New York, 2nd edition, 1996.

[11] Reinhard Gillitzer and Matthias Goebeler. Chemokines in cutaneous wound healing. *J Leukocyte Biol*, 69:513–521, April 2001.

[12] Guo–li Ming, Scott T. Wong, John Henley, Xiao–bing Yuan, Hong–jun Song, Nicholas C. Spitzer, and Mu–ming Poo. Adaptation in the chemotactic guidance of nerve growth cones. *Nature*, 417(6887):411–418, May 2002.

[13] William J. Rosoff, Jeffrey S. Urbach, Mark A. Esrick, Ryan G. McAllister, Linda J. Richards, and Geoffrey J. Goodhill. A new chemotaxis assay shows the extreme sensitivity of axons to molecular gradients. *Nat Neurosci*, 7(6):678–682, May 2004.

[14] Duncan Mortimer, Thomas Fothergill, Zac Pujic, Linda J. Richards, and Geoffrey J. Goodhill. Growth cone chemotaxis. *Trends Neurosci*, 31(2):90–98, February 2007.

[15] Michael Eisenbach. *Chemotaxis*. Imperial College Press, London, 2004.

[16] Peter J. M. Van Haastert and Peter N. Devreotes. Chemotaxis: signalling the way forward. *Nat Rev Mol Cell Bio*, 5(8):626–634, August 2004.

[17] Clifford S. Patlak. Random walk with persistence and external bias. *B Math Biophys*, 15(3):311–338, 1953.

[18] Evelyn F. Keller and Lee A. Segel. Model for chemotaxis. *J Theor Biol*, 30(2):225–234, February 1971.

[19] M. P. Brenner, P. Constantin, L. P. Kadanoff, A. Schenkel, and S. C. Venkataramani. Diffusion, attraction and collapse. *Nonlinearity*, 12:1071–1098, 1999.

[20] D. D. Holm and V. Putkaradze. Aggregation of finite sized particles with variable mobility. *Phys. Rev. Lett*, 95:225105, 2005.

[21] D. D. Holm and V. Putkaradze. Formation of clumps and patches in self-aggregation of finite size particles. *Physica D*, 220:183, 2005.

[22] A. Bertozzi and T. Laurent. Finite-time blow-up of solutions of an aggregation equation in R^n. *Communications in Mathematical Physics*, 247:717–735, 2007.

[23] A. Bertozzi and J. Brandman. Finite-time blow-up of L^∞-weak solutions of an aggregation equation. *Communications in Mathematical Sciences*, 8(1):45–65, 2010.

[24] A. Bertozzi C. Topaz and M. Lewis. A nonlocal continuum model for biological aggregation. *Bull. Math. Biology*, 68:1601–1623, 2006.

[25] A. Blanchet, J. A. Carrillo, and P. Laurençot. Critical mass for a Patlak–Keller–Segel model with degenerate diffusion in higher dimensions. *Calculus of variations and partial differential equations*, 35(2):133–168, February 2008.

[26] Eirikur Palsson and Hans G. Othmer. A model for individual and collective cell movement in Dictyostelium discoideum. *P Natl Acad Sci USA*, 97(19):10448–10453, September 2000.

[27] Manolya Eyiyurekli. A computational model of chemotaxis–based cell aggregation. Master's thesis, Drexel University, August 2006.

[28] S. J. Singer and Garth L. Nicolson. Structure of cell membranes. *Science*, 175(4023):720–731, February 1972.

[29] Ken Jacobson, Ole G. Mouritsen, and Richard G. W. Anderson. Lipid rafts: at a crossroad between cell biology and physics. *Nat Cell Biol*, 9(1):7–14, 2007.

[30] John Katsaras and Thomas Gutberlet, editors. *Lipid Bilayers – Structure and Interactions*. Biological Physics Series. Springer Verlag, Berlin, 2001.

[31] Harvey T. McMahon and Jennifer L. Gallop. Membrane curvature and mechanisms of dynamic cell membrane remodelling. *Nat*, 438(7068):531–710, 2005.

[32] Peter J. Quinn, editor. *Membrane Dynamics and Domains*, volume 37 of *Subcellular Biochemistry*. Kluwer Academic/Plenum Publishers, New York, 2004.

[33] S. J. Marrink, A. H. de Vries, and D. P. Tieleman. Lipids on the move: Simulations of membrane pores, domains, stalks and curves. *Biochim Biophys Acta*, 1788(1):149–168, 2009.

[34] Richard G. W. Anderson and Ken Jacobson. A role for lipid shells in targeting proteins to caveolae, rafts, and other lipid domains. *Science*, 296(5574):1821–1825, June 2002.

[35] D. P. Tieleman, S. J. Marrink, and H. J. C. Berendsen. A computer perspective of membranes: molecular dynamics studies of lipid bilayer systems. *Biochim Biophys Acta*, 1331(3):235–270, November 1997.

[36] Yee–Hung M. Chan and Steven G. Boxer. Model membrane systems and their applications. *Current Opinion in Chemical Biology*, 11(6):581–587, December 2007.

[37] Scott E. Feller, editor. *Computational Modeling of Membrane Bilayers*, volume 60 of *Current Topics in Membranes*. Elsevier, Inc., Amsterdam, 2008.

[38] Gregory A. Voth, editor. *Coarse–graining of Condensed Phase and Biomolecular Systems*. CRC Press, Boca Raton, 2009.

[39] D. Marsh. Elastic curvature constants of lipid monolayers and bilayers. *Chemistry and Physics of Lipids*, 144(2):146–159, November–December 2006.

[40] Drazen Raucher and Michael P. Sheetz. Cell spreading and lamellipodial extension rate is regulated by membrane tension. *Journal of Cell Biology*, 148(1):127–136, January 2000.

[41] Norbert Kučerka, Mikhail A. Kiselev, and Pavel Balgavỳ. Determination of bilayer thickness and lipid surface area in unilamellar dimyristoylphosphatidylcholine vesicles from small–angle neutron scattering curves: a comparison of evaluation methods. *European Biophysics Journal*, 33(4):328–334, July 2004.

[42] Drazen Raucher and Michael P. Sheetz. Characteristics of a membrane reservoir buffering membrane tension. *Biophysical Journal*, 77(4):1992–2002, 1999.

[43] Kinneret Keren and Julie A. Theriot. Biophysical aspects of actin–based cell motility in fish epithelial keratocytes. In Peter Lenz, editor, *Cell Motility*, pages 31–58. Springer Science + Business Media, LLC, New York, 2008.

[44] Marileen Dogterom, Julien Husson, Liedewij Laan, Laura Munteanu, and Christian Tischer. Microtubule forces and organization. In Peter Lenz, editor, *Cell Motility*, pages 93–115. Springer Science + Business Media, LLC, New York, 2008.

[45] Mohammad R. K. Mofrad and Roger D. Kamm, editors. *Cytoskeletal Mechanics: models and measurements*. Cambridge University Press, New York, 2006.

[46] Dennis Bray. *Cell Movements from molecules to motility*. Garland Publishing, New York, 2nd edition, 2001.

[47] Julie A. Theriot and Timothy J. Mitchison. Actin microfilament dynamics in locomoting cells. *Nature*, 352(6331):126–131, July 1991.

[48] Alan Weeds and Sutherland Maciver. F–actin capping proteins. *Curr Opin Cell Biol*, 5(1):63–69, February 1993.

[49] Dorothy A. Schafer and John A. Cooper. Control of actin assembly at filament ends. *Annu Rev Cell Dev Biol*, 11:497–518, November 1995.

[50] Marie–France Carlier and Dominique Pantaloni. Control of actin dynamics in cell motility. *J Mol Biol*, 269(4):459–467, June 1997.

[51] Yoram A. Puius, Nicole M. Mahoney, and Steven C. Almo. The modular structure of actin–regulatory proteins. *Curr Opin Cell Biol*, 10(1):23–34, February 1998.

[52] Hui Qiao Sun, Masaya Yamamoto, Marisan Mejillano, and Helen L. Yin. Gelsolin, a multifunctional actin regulatory protein. *J Biol Chem*, 274(47):33179–33182, November 1999.

[53] Amy M. McGough, Chris J. Staiger, Jung–Ki Min, and Karen D. Simonetti. The gelsolin family of actin regulatory proteins: modular structures, versatile functions. *FEBS Lett*, 552(2–3):75–81, September 2003.

[54] Narendra Kumar and Seema Khurana. Identification of a functional switch for actin severing by cytoskeletal proteins. *J Biol Chem*, 279(24):24915–24918, June 2004.

[55] John H. Hartwig and David J. Kwiatkowski. Actin–binding proteins. *Curr Opin Cell Biol*, 3(1):87–97, February 1991.

[56] Enrique M. De La Cruz. Actin–binding proteins: An overview. In C. G. dos Remedios and D. D. Thomas, editors, *Molecular interactions of actin–actin structure and actin–binding proteins*, pages 123–134. Springer–Verlag Berlin, 2001.

[57] Steven J. Winder and Kathryn R. Ayscough. Actin–binding proteins. *J Cell Sci*, 118(4):651–654, 2005.

[58] Tim Mitchison and Marc Kirschner. Dynamic instability of microtubule growth. *Nature*, 312(5991):237–242, November 1984.

[59] Arshad Desai and Timothy J. Mitchison. Microtubule polymerization dynamics. *Annu Rev Cell Dev Biol*, 13:83–117, November 1997.

[60] M. Elbaum, A. Chausovsky, E. T. Levy, M. Shtutman, and A. D. Bershadsky. Microtubule involvement in regulating cell contractility and adhesion–dependent signalling: a possible mechanism for polarization of cell motility. In J. M. Lackie, G. A. Dunn, and G. E. Jones, editors, *Cell Behaviour: Control and Mechanism of Motility*, pages 147–172. Portland Press, London, 1999.

[61] Fred Chang, Becket Feierbach, and Sophie Martin. Regulation of actin assembly by microtubules in fission yeast cell polarity. In Gregory Bock and Jamie Goode, editors, *Signalling networks in cell shape and motility*, pages 59–72. John Wiley and Sons, Ltd., Chinchester, 2005.

[62] J. Victor Small and Irina Kaverina. Polarized cell motility: microtubules show the way. In Doris Wedlich, editor, *Cell Migration in Development and Disease*, pages 15–31. Wiley–VCH Verlag GmbH and Co. KGaA, Weinheim, 2005.

[63] Brigitte Raynaud–Messina and Andreas Merdes. γ–tubulin complexes and microtubule organization. *Curr Opin Cell Biol*, 19(1):24–30, 2007.

[64] Manfred Schliwa, Ursula Euteneuer, Ralph Gräf, and Masahiro Ueda. Centrosomes, microtubules and cell migration. In John M. Lackie, Graham A. Dunn, and Gareth E. Jones, editors, *Cell Behaviour: Control and Mechanism of Motility*. Portland Press, London, 1999.

[65] Elias Lazarides. Intermediate filaments: A chemically heterogeneous developmentally regulated class of proteins. *Annu Rev Biochem*, 51:219–250, July 1982.

[66] Peter M. Steinert and Dennis R. Roop. Molecular and cellular biology of intermediate filaments. *Annu Rev Biochem*, 57:593–625, July 1988.

[67] Elaine Fuchs and Klaus Weber. Intermediate filaments: Structure, dynamics, function and disease. *Annu Rev Biochem*, 63:345–382, 1994.

[68] Miri Yoon, Robert D. Moir, Veena Prahlad, and Robert D. Goldman. Motile properties of vimentin intermediate filament networks in living cells. *J Cell Biol*, 143(1):147–157, October 1998.

[69] Brian T. Helfand, Lynne Chang, and Robert D. Goldman. Intermediate filaments are dynamic and motile elements of cellular architecture. *J Cell Sci*, 117:133–141, 2004.

[70] Lynne Chang and Robert D. Goldman. Intermediate filaments mediate cytoskeletal crosstalk. *Nat Rev Mol Cell Biol*, 5(8):601–613, August 2004.

[71] Seyun Kim and Pierre A. Coulombe. Intermediate filament scaffolds fulfill mechanical, organizational, and signaling functions in the cytoplasm. *Gene Dev*, 21(13):1581–1597, July 2007.

[72] Lynne Chang, Kari Barlan, Ying–Hao Chou, Boris Grinand Margot Lakonishok, Anna S. Serpinskaya, Dale K. Shumaker, Harald Herrmann, Vladimir I. Gelfand, , and Robert D. Goldman. The dynamic properties of intermediate filaments during organelle transport. *J Cell Sci*, 122(16):2914–2923, August 2009.

[73] Tim Mitchison and Marc Kirschner. Cytoskeletal dynamics and nerve growth. *Neuron*, 1(9):761–772, November 1988.

[74] Ralph Nossal. On the elasticity of cytoskeletal networks. *Biophys J*, 53(3):349–359, March 1988.

[75] J. J. Blum and M. C. Reed. A model for slow axonal transport and its application to neurofilamentous neuropathies. *Cell Motil Cytoskeleton*, 12(1):53–65, 1989.

[76] Diedrik Menzel. *The Cytoskeleton of the algae*. CRC Press, 1992.

[77] Maria J. Schilstra Peter M. Bayley and Stephen R. Martin. Microtubule dynamic instability: numerical simulation of microtubule transition properties using a lateral cap model. *J Cell Sci*, 95(1):33–48, 1990.

[78] Leah Edelstein–Keshet. A mathematical approach to cytoskeletal assembly. *Eur Biophys J*, 27(5):521–531, August 1998.

[79] Athan Spiros and Leah Edelstein–Keshet. Testing a model for the dynamics of actin structures with biological parameter values. *Bull Math Biol*, 60(2):275–305, March 1998.

[80] D. Sept, H. J. Limbach, H. Bolterauer, and J. A. Tuszynski. A chemical kinetics model for microtubule oscillations. *J Theor Biol*, 197(1):77–88, March 1999.

[81] Wolfgang Alt and Micah Dembo. Cytoplasm dynamics and cell motion: two–phase flow models. *Math Biosci*, 156(1–2):207–228, March 1999.

[82] Eric Karsenti, François Nédélec, and Thomas Surrey. Modelling microtubule patterns. *Nat Cell Biol*, 8(11):1204–1211, November 2006.

[83] Ju Li, George Lykotrafitis, Ming Dao, and Subra Suresh. Cytoskeletal dynamics of human erythrocyte. *Proc Natl Acad Sci*, 104(12):4937–4942, March 2007.

[84] Robert Kirmse, Stephanie Portet, Norbert Mücke, Ueli Aebi, Harald Herrmann, and Jörg Langowski. A quantitative kinetic model for the in vitro assembly of intermediate filaments from tetrameric vimentin. *J Biol Chem*, 282(25):18563–18572, June 2007.

[85] Vikram S. Deshpande, Robert M. McMeeking, and Anthony G. Evans. A bio-chemo-mechanical model for cell contractility. *Proc Natl Acad Sci*, 103(38):14015–14020, September 2006.

[86] Yaozhi Luo, Xian Xu, Tanmay Lele, Sanjay Kumar, and Donald E. Ingber. A multi–modular tensegrity model of an actin stress fiber. *J Biomech*, 41(11):2379–2387, August 2008.

[87] Esa Kuusela and Wolfgang Alt. Continuum model of cell adhesion and migration. *J Math Biol*, 58(1–2):125–161, January 2009.

[88] Lars Wolff and Klaus Kroy. Mechanical stability: A construction principle for cells. *diffusion-fundamentals.org*, 11(56):1–14, 2009.

[89] S. Benoit. Personal web site. http://lamar.colostate.edu/∼sbenoit/.

CHAPTER 4

MODELS OF HELICAL BIOLOGICAL MOLECULES

Remark This chapter is based on the paper *Helical states of nonlocally interacting molecules and their linear stability: geometric approach* by S. Benoit, D. D. Holm, and V. Putkaradze, published in the Journal of Physics A: Mathematical and Theoretical, Volume 44, Number 5, February, 2011. My work on this paper focused on development of the models and simulations and deriving the energy landscape and stable conformations. The linear stability analysis and geometric analyses, included here for completeness, are due to Drs. Putkaradze and Holm.

In this chapter, the equations for strands of rigid charge configurations interacting nonlocally are formulated on the special Euclidean group, $SE(3)$, which naturally generates helical conformations. Helical stationary shapes are found by minimizing the energy for rigid charge configurations positioned along an infinitely long molecule with charges that are off-axis. The classical energy landscape for such a molecule is complex with many local energy minima, even when limited to helical shapes. The question of linear stability and selection of stationary shapes is studied using an $SE(3)$ method that naturally accounts for the helical geometry. We investigate the linear stability of a general helical polymer that possesses torque-inducing non-

local self-interactions and find the exact dispersion relation for the stability of the helical shapes with an arbitrary interaction potential. We explicitly determine the linearization operators and compute the numerical stability for the particular example of a linear polymer comprising a flexible rod with a repeated configuration of two equal and opposite off-axis charges, thereby showing that even in this simple case the non-local terms can induce instability that leads to the rod assuming a helical shape.

4.1 Introduction

Molecules with repeating subunits that spontaneously form helical shapes are ubiquitous in nature, and are common in the context of cellular structures. A classical example is given by α-helices, which are an important motif in the secondary structure of proteins [1]. Other examples of naturally occurring helical structures within cells include intermediate filament proteins keratin and vimentin [2], the myosin and kinesin families of motor proteins [2], tubulin microtubules [3], RecA and Rad51 filaments on DNA [4], and others. Artificial polymer structures that spontaneously take helical forms have also been obtained, see for example [5, 6]. The spontaneous emergence and persistence of helical shapes has been the subject of several recent studies. For progress in the description of helical shapes based on elastic rod theory, see [7]. The spontaneous formation of helical structures under compaction of an elastic rod has also been shown, by using a Lennard-Jones-like potential whose repulsive core eliminates self-intersections [8].

Molecular shapes arise through an interplay between elastic forces caused by bending and twisting of the molecular bonds, and long- and short-range forces (such as

electrostatic and van der Waals forces) between individual atoms comprising the molecule. For elastic interactions only, Kirchoff's rod theory [9, 10] has been used to model molecular shapes with notable success [7, 11–16]. The recent work on non-local interactions has considered helical molecules as rods with charges concentrated along the axis of the rod only [7, 10, 17, 18]. Other authors used energy optimization with short-range repulsion only, without initial assumptions on the molecular shapes, and yet helical shapes showed surprising persistence [8, 19], either for the whole molecule, or in parts of it. The crucial question is whether such helical shapes persist for more complex interactions or geometries of the molecules. For example, more accurate models of molecules could include charges that are attached at a certain distance from the axis, as shown in Fig. 4.1. Interatomic forces are caused by some potential – or combination of potentials – that depends explicitly on the geometry of the molecule and the Euclidean distance d between two points s and s' along the curve representing the axis of the molecule. Equations of motion for such molecules were derived recently using nonlocal extensions of exact geometric rod theory [20, 21]. However, the question of existence of helical shapes for more complex interactions between the parts of the molecule and the methods for their computations has remained open. In this chapter, we demonstrate a fast and efficient method for finding such states, and in addition show how we can achieve a complete classification of all helical states. We also show that the geometric approach allows exact computation of dispersion relations and linear instability growth rates.

Outline of the chapter Section 4.2 derives the equations for nonlocally interacting rods of rigid charge configurations, called *bouquets*. These bouquets are fixed con-

figurations of charges, rigidly attached to the molecule's central axis. The molecule is comprised of a long repeating chain of such bouquets. The equations are formulated on the special Euclidean group, $SE(3)$, consisting of three-dimensional spatial rotations and translations, which naturally generate helical configurations. Section 4.3 demonstrates how to find helical stationary shapes by minimizing the energy for bouquets positioned along an infinitely long molecule with charges that are off-axis. As shown in Section 4.4, even this simple molecule, when deformed into helical configurations, shows quite a complex and intriguing energy landscape. Section 4.5 briefly discusses how to extend the ideas presented above to include more general shapes. In particular, we consider a molecular shape we refer to as a 2-helix. Here, we define an n-helix as a molecule that is a true helix consisting of repeating groups of n bouquets along the axis. For a 2-helix, one optimizes the energy over both the shape of the helix and the relative configuration of two bouquets. One may treat n-helices by optimizing configurations of n bouquets using the same methods. After revealing the complexity of the energy landscape, even when limited to helical shapes, and the large number of energy minima in that "helical universe", it is natural to pose the question of linear stability and selection of the stationary shapes. By using an $SE(3)$ method that naturally accounts for the helical geometry, in Section 4.6 we investigate the linear stability of a general helical polymer that possesses torque-inducing non-local self-interactions. As far as we know, no such work has been undertaken previously. The geometric approach in Section 4.6 allows us to find the exact dispersion relation for the stability of the helical shapes with an arbitrary interaction potential. Of course, for particular applications it is important to explicitly compute the linearization operators. This is accomplished in Section 4.7 using geometric methods

to compute derivatives of the potential energy. Section 4.8 computes the numerical stability for the particular example of a linear polymer comprising a charged rod with repeated configuration of two equal and opposite charges that interact through a screened electrostatic and Lennard-Jones potential. For the sake of simplicity, we concentrate on the stability of a polymer that is perfectly straight in its unstressed configuration. Such a polymer is neutrally stable in the absence of the nonlocal interactions. It is therefore interesting that nonlocal terms can induce instability that causes the molecule to deform into helical conformations. Physically, this instability is connected to the tendency of the rod to minimize its energy and properly align the dipole moments of each bouquet by twisting, as was already seen in the minimum energy calculation shown in Section 4.4.

We note that these results are difficult to achieve in the traditional Kirchhoff based approach, as the non-local interactions depend on the relative distance and orientation of the charges at different points on the rod. In these traditional methods, the equations of motion are written in a coordinate system that is moving with the rod, and changing with both position on the rod s and time t. Thus, one has to write equations of motion, *i.e.* calculate the momenta and forces, in a frame attached to the rod that is moving and rotating in a non-inertial fashion. The main difficulty arises because the relative distance between the charges depends on both their bouquet positions on the centerline and the local rotation of the bouquet about the centerline. To find this orientation one needs to integrate auxiliary equations of motion at each step (*Darboux*'s vector), so the solution must be known *before* the relative distance between the charges can be computed. However, the solution explicitly depends on the relative distance between the charges, and so the closure

of the system is problematic. These complications make an explicit derivation of equations of motion from Kirchhoff's approach difficult, if not impossible. On the other hand, the approach we suggest here provides a very straightforward derivation. One need not deal with vectors, forces and torques in non-inertial frames of references, in writing conservation laws, and so forth. Instead, in the geometric approach, helices are treated in exactly the same fashion as straight lines, so stationary helical states and even their linear stability properties may be considered conveniently.

4.2 Derivation in $SE(3)$ coordinates

In this section, we derive the full equations of motion for a self-interacting rod in the discrete and continuous cases. This derivation is based on the exact geometric rod theory [22], that derives equations equivalent to Kirchhoff's equations for elastic rods using symmetry-reduced variables. Since the derivation of the exact geometric rod equations for purely elastic rods is well-established, we shall concentrate on the corresponding derivation of the nonlocal equations in the language of the Lie group $SE(3)$ – the Special Euclidean group of orthogonal rotations and translations in $\mathbb{R}^3$. This familiar Lie group is a semidirect product of $SO(3)$ and $\mathbb{R}^3$ with the following definition of group multiplication:

$$(\Lambda_1, \boldsymbol{r}_1) \cdot (\Lambda_2, \boldsymbol{r}_2) = (\Lambda_1\Lambda_2, \Lambda_1\boldsymbol{r}_2 + \boldsymbol{r}_1) \tag{4.1}$$

for $\Lambda_1, \Lambda_2 \in SO(3)$ and $\boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathbb{R}^3$. We use the notation $\mathfrak{se}(3)$ for the corresponding Lie algebra. For the reader's convenience, we summarize the properties of this group, its adjoint and coadjoint actions Ad, Ad^*, ad, ad^* in Appendix C. More details can be found in, for example, [23].

Definition A *bouquet* is a rigid, non-deformable assembly of spheres, each characterized by an interaction radius (which applies to short-range interactions like Lennard-Jones), and an electrostatic charge.

A molecule, as considered in this chapter, consists of a rod that represents the molecule's central axis, with bouquets attached at fixed base points along the rod. Each bouquet is characterized by the spatial position coordinate of its base $\boldsymbol{r}(s) \in \mathbb{R}^3$ and its orientation $\Lambda(s) \in SO(3)$, which together define an element $\sigma(s) = (\Lambda(s), \boldsymbol{r}(s)) \in SE(3) \cong SO(3) \times \mathbb{R}^3$. We are interested in describing the effects of nonlocal interactions, which in this chapter denotes all inter-atomic forces that affect parts of the molecule not immediately adjacent to each other. These are all the forces that do not come from elastic deformation, and include, for example, electrostatic and Lennard-Jones forces.

To describe nonlocal interactions among these bouquets of charges, we define their relative orientation and position variables by the $SE(3)$ product

$$\begin{aligned} \Xi(s,s') = \sigma^{-1}(s)\sigma(s') &= \left(\Lambda^{-1}(s)\Lambda(s')\,,\, \Lambda^{-1}(s)(\boldsymbol{r}(s') - \boldsymbol{r}(s)\right) \\ &:= \left(\xi(s,s')\,,\, \boldsymbol{\kappa}(s,s')\right), \end{aligned} \tag{4.2}$$

where $\xi(s,s') \in SO(3)$ and $\boldsymbol{\kappa}(s,s') \in \mathbb{R}^3$. The position of the k-th charge in the bouquet whose base is at s along the center-line of the rod is given by

$$\mathbf{c}_k(s) = \boldsymbol{r}(s) + \Lambda(s)\boldsymbol{\eta}_k(s)\,. \tag{4.3}$$

Here $\boldsymbol{\eta}_k(s)$ denotes a vector from the base point at s on the rod to the k-th off-axis charge of the bouquet as measured in a rigid Cartesian frame with orientation $\Lambda(s)$. Then, the distance $d_{km}(s,s')$ between the k-th charge at s and m-th charge at s' is

given by

$$\begin{aligned} d_{km}(s,s') &= \left|\mathbf{c}_k(s) - \mathbf{c}_m(s')\right| \\ &= \left|\boldsymbol{r}(s) + \Lambda(s)\boldsymbol{\eta}_k(s) - \boldsymbol{r}(s') - \Lambda(s')\boldsymbol{\eta}_m(s')\right| \\ &= \left|\Lambda^{-1}(s)(\boldsymbol{r}(s) - \boldsymbol{r}(s')) + \boldsymbol{\eta}_k(s) - \Lambda^{-1}(s)\Lambda(s')\boldsymbol{\eta}_m(s')\right| \\ &= \left|\boldsymbol{\kappa}(s,s') + \boldsymbol{\eta}_k(s) - \xi(s,s')\boldsymbol{\eta}_m(s')\right| = d_{km}\big(\Xi(s,s')\big)\,. \end{aligned} \tag{4.4}$$

Remark 4.2.1. *In finding particular helical solutions and analyzing their stability, we consider the case of polymers where all the bouquets are exactly the same, so* $\boldsymbol{\eta}_k(s)$ *and* $\boldsymbol{\eta}_m(s')$ *are independent of* s *and* s'*. Nonetheless, the equations of motion we derive in this section, (4.22) and (4.23), would also be valid for arbitrary dependence of the bouquet's geometry on position. However, the linear stability analysis performed later will explicitly use the fact that all bouquets are identical, and thus cannot be applied to the conformations of molecules with varying geometry.*

Consequently, the total energy of the rod due to nonlocal interactions among the charges that compose it is obtained in the continuous case as

$$E = \sum_{k,m} \int U\big(d_{km}(\Xi(s,s'))\big)\,\mathrm{d}s\,\mathrm{d}s'\,, \tag{4.5}$$

for interaction potential U between individual pairs of charges. In the discrete case, the integral becomes a sum and s, s' are discrete indices, so that

$$E = \sum_{s,s',k,m} U\big(d_{km}(\Xi(s,s'))\big)\,. \tag{4.6}$$

The corresponding contribution of nonlocal interactions to the action in Hamilton's principle for the dynamics of the rod is thus (in the discrete case)

$$S_{nl} = -\int \sum_{s,s',k,m} U\big(d_{km}(\Xi(s,s'))\big)\,\mathrm{d}t\,. \tag{4.7}$$

In order to compute the contribution of the mutual charge interactions to the dynamics of the rod one must take the variation of this nonlocal part of the action with respect to the charge conformation Ξ. Upon denoting

$$\nu(s) = \sigma^{-1}(s)\delta\sigma(s) \in \mathfrak{se}(3)\,, \tag{4.8}$$

where $\mathfrak{se}(3)$ is the Lie algebra of the Lie group $SE(3)$, we find

$$\delta\Xi(s,s') = \delta\big(\sigma^{-1}(s)\sigma(s')\big) = -\nu(s)\Xi(s,s') + \Xi(s,s')\nu(s')\,. \tag{4.9}$$

We now define a pairing $\big\langle\cdot,\cdot\big\rangle_{TSE(3)}$ between the elements of $TSE(3)$ and $TSE(3)^*$ – the tangent and cotangent spaces of $SE(3)$ – as follows. If a vector U is tangent to $SE(3)$ at point $\sigma \in SE(3)$, and W co-tangent to $SE(3)$ at the same point σ, then $\sigma^{-1}U$ brings the vector U to the identity element of the group, so $\sigma^{-1}U \in \mathfrak{se}(3)$ is in the Lie algebra. Similarly, $\sigma^{-1}W \in \mathfrak{se}(3)^*$, so a scalar product between these elements can be taken as defined in the appendix. Thus, we define

$$\big\langle U\,,\,W\big\rangle_{TSE(3)} = \big\langle \sigma^{-1}U\,,\,\sigma^{-1}W\big\rangle_{\mathfrak{se}(3)}\,. \tag{4.10}$$

In what follows, we drop the subscript $TSE(3)$ from the scalar product as not to burden the notation unnecessarily.

In order to use the minimal action principle and derive the equations of motion, we need to take the variations with respect to Ξ. The variation $\delta\Xi$ defined in (4.9) is an element of $TSE(3)$, as $\delta\Xi$ is tangent to $SE(3)$ at the point Ξ. Then, the derivative $\delta U/\delta\Xi$ is an element of $TSE(3)^*$ as it is co-tangent to the group at the same point Ξ. The pairing

$$\Big\langle \frac{\delta U}{\delta\Xi}\,,\,\delta\Xi\Big\rangle$$

can be now defined according to (4.10).

Consequently, the variation of the nonlocal part of the action in (4.7) is given by

$$\begin{aligned}\delta S_{nl} &= -\int \sum_{s,s',k,m} \Big\langle \frac{\delta U}{\delta \Xi}\,,\, \delta\Xi \Big\rangle \mathrm{d}t \\ &= -\int \sum_{s,s',k,m} \Big\langle \frac{\delta U}{\delta \Xi}\,,\, -\nu(s)\Xi(s,s') + \Xi(s,s')\nu(s') \Big\rangle \mathrm{d}t \\ &= -\int \sum_{s,s',k,m} \Big\langle -\frac{\delta U}{\delta \Xi}(s,s')\Xi^{-1}(s,s') + \Xi(s,s')\frac{\delta U}{\delta \Xi}(s',s),\, \nu(s) \Big\rangle \mathrm{d}t,\end{aligned} \tag{4.11}$$

in which the last step uses the relation $\Xi(s',s) = \Xi^{-1}(s,s')$ obtained from the definition of Ξ in (4.2). We shall now proceed with the computation of the variations of action with respect to all dynamical quantities and thereby obtain the equations of motion.

Velocities. The local part of the Lagrangian in Hamilton's principle for the dynamics of the rod is written by introducing the left-invariant variables

$$\mu = \sigma^{-1}\dot{\sigma} = (\Lambda\,,\,\boldsymbol{r})^{-1}\left(\dot{\Lambda}\,,\,\dot{\boldsymbol{r}}\right) = \left(\Lambda^{-1}\dot{\Lambda}\,,\,\Lambda^{-1}\dot{\boldsymbol{r}}\right) \in \mathfrak{se}(3)\,, \tag{4.12}$$

as velocities taking values in the Lie algebra $\mathfrak{se}(3)$.

Elastic deformations. In the continuous case, the invariant variables that describe elastic deformations are,

$$\lambda = \sigma^{-1}\sigma' \in \mathfrak{se}(3)\,. \tag{4.13}$$

In the discrete case, following the Moser-Veselov method for numerical discretization of rigid body dynamics [24] we set,

$$\lambda = \sigma^{-1}(s)\sigma(s+1) \in SE(3)\,, \tag{4.14}$$

where $s = 1, 2, \ldots$ is the discrete index labeling a given base. The elastic part of the Lagrangian will then depend on λ.

Compatibility conditions. The *compatibility conditions* are obtained, in the continuous case, by using equality of cross derivatives, so that $\sigma_{st} = \sigma_{ts}$. Differentiating (4.12) with respect to s and (4.13) with respect to t, then subtracting yields

$$\frac{\partial \mu}{\partial s} - \frac{\partial \lambda}{\partial t} = -\lambda\mu + \mu\lambda := \left[\mu\,,\,\lambda\right]_{\mathfrak{se}(3)} = \mathrm{ad}_\mu \lambda\,, \tag{4.15}$$

where $\left[\mu\,,\,\lambda\right]_{\mathfrak{se}(3)}$ is the commutator of μ and λ in $\mathfrak{se}(3)$. In the discrete case, we write (4.14) as $\sigma(s+1,t) = \sigma(s,t)\lambda(s,t)$. Differentiating this condition with respect to time, we get

$$\frac{\partial \lambda}{\partial t}(s) = \lambda(s)\mu(s+1) - \mu(s)\lambda(s)\,, \tag{4.16}$$

or in terms of $\mathfrak{se}(3)$-algebra quantities only

$$\lambda^{-1}\frac{\partial \lambda}{\partial t}(s) = \mu(s+1) - \mathrm{Ad}_{\lambda^{-1}}\mu(s)\,. \tag{4.17}$$

Dynamical equations: variations In the continuous case, the variations of the velocities on the Lie algebra $\mathfrak{se}(3)$ satisfy,

$$\delta\mu = -\nu\mu + \mu\nu + \frac{\partial \nu}{\partial t}\,. \tag{4.18}$$

In the discrete case, this is replaced by a variation on the Lie group $SE(3)$,

$$\lambda^{-1}\delta\lambda(s) = -\mathrm{Ad}_{\lambda^{-1}}\nu(s) + \nu(s+1)\,. \tag{4.19}$$

For the discrete case, one may then compute the variation of the Lagrangian as,

$$\begin{aligned}
&\sum_s \Big\langle \lambda^{-1}(s)\frac{\delta l}{\delta \lambda}(s)\,,\lambda^{-1}(s)\delta\lambda(s)\Big\rangle \\
&\quad=\sum_s \Big\langle \lambda^{-1}(s)\frac{\delta l}{\delta \lambda}(s)\,,-\mathrm{Ad}_{\lambda^{-1}(s)}\nu(s)+\nu(s+1)\Big\rangle \\
&\quad=\sum_s \Big\langle -\mathrm{Ad}^*_{\lambda^{-1}(s)}\Big(\lambda^{-1}(s)\frac{\delta l}{\delta \lambda}(s)\Big)+\lambda^{-1}(s-1)\frac{\delta l}{\delta \lambda}(s-1),\,\nu(s)\Big\rangle.
\end{aligned} \tag{4.20}$$

In continuous time, the Euler-Poincaré equations emerge from the following direct computation as in [20, 21],

$$\begin{aligned}
\int\int \Big\langle \frac{\delta l}{\delta \mu}\,,\delta\mu\Big\rangle \mathrm{d}t\,\mathrm{d}s &= \int\int \Big\langle \frac{\delta l}{\delta \mu}\,,\,-\nu\mu+\mu\nu+\frac{\partial \nu}{\partial t}\Big\rangle \mathrm{d}t\,\mathrm{d}s \\
&= \int\int \Big\langle \frac{\delta l}{\delta \mu}\,,\,\mathrm{ad}_\mu\nu+\frac{\partial \nu}{\partial t}\Big\rangle \mathrm{d}t\,\mathrm{d}s \\
&= \int\int \Big\langle \Big(-\frac{\partial}{\partial t}+\mathrm{ad}^*_\mu\Big)\frac{\delta l}{\delta \mu}\,,\,\nu\Big\rangle \mathrm{d}t\,\mathrm{d}s\,.
\end{aligned} \tag{4.21}$$

Likewise, the equations of motion in continuous time for the spatially discrete nonlocal rod, may be written in $SE(3)$ coordinates as

$$\begin{aligned}
&\Big(-\frac{\partial}{\partial t}+\mathrm{ad}^*_\mu\Big)\frac{\delta l}{\delta \mu}-\mathrm{Ad}_{\lambda^{-1}(s)}\Big(\lambda^{-1}(s)\frac{\delta l}{\delta \lambda}(s)\Big)+\lambda(s-1)\frac{\delta l}{\delta \lambda}(s-1) \\
&\qquad\qquad -\sum_{s',m,k} -\frac{\delta U}{\delta \Xi}(s,s')\Xi^{-1}(s,s')+\Xi(s,s')\frac{\delta U}{\delta \Xi}(s',s)=0\,.
\end{aligned} \tag{4.22}$$

In the continuous case, a similar calculation gives (note that now $\lambda=\sigma^{-1}\sigma'\in\mathfrak{se}(3)$):

$$\begin{aligned}
&\Big(-\frac{\partial}{\partial t}+\mathrm{ad}^*_\mu\Big)\frac{\delta l}{\delta \mu}+\Big(-\frac{\partial}{\partial s}+\mathrm{ad}^*_\lambda\Big)\frac{\delta l}{\delta \lambda} \\
&\qquad -\sum_{m,k}\int -\frac{\delta U}{\delta \Xi}(s,s')\Xi^{-1}(s,s')+\Xi(s,s')\frac{\delta U}{\delta \Xi}(s',s)\mathrm{d}s'=0\,.
\end{aligned} \tag{4.23}$$

Remark Notice that (4.23) are exactly the equations derived earlier in [20, 21]. However, the discrete equations (4.22) for nonlocally interacting molecules are new, as far as we know.

Lemma 4.2.2. *For an arbitrary potential $U(d)$, equations (4.22) and (4.23) reduce to algebraic equations for helical solutions. The solutions of these algebraic equations are stationary helical shapes.*

Proof Helical configurations in the discrete case are obtained from taking a given element $a \in SE(3)$ and defining $\sigma(s) = a^s$. In the continuous case, a given element $\lambda \in \mathfrak{se}(3)$ generates $\sigma(s)$ from the differential equation $\sigma'(s) = \sigma(s)\lambda_0$ (exponential map). In either case,

$$\Xi(s, s') = \sigma^{-1}(s)\sigma(s') = \sigma(s' - s)\,. \tag{4.24}$$

Then,

$$d_{km}(s, s') = d_{km}\big(\Xi(s, s')\big) = d_{km}\big(\sigma(s' - s)\big)\,,$$

and therefore

$$\frac{\delta U}{\delta \Xi}(s, s') := D_1(s' - s),$$

where D_1 is some $TSE(3)^*$-valued function that depends only on the difference between s and s'. Thus, the right-hand sides under the sum (4.22) and in the integral in (4.23) depend only on the difference between s and s', and are therefore *constant*. Likewise, the left-hand sides of these equations are also constant, as $\mu = 0$ and $\lambda = a \in SE(3)$ in the discrete case and $\lambda = \sigma_0 \in \mathfrak{se}(3)$ in the continuous case. Thus, integro-differential equations (4.22) and (4.23) reduce to algebraic equations for helical solutions, as long as the potential between two charges depends (in an arbitrary fashion) only on the Euclidean distance between those charges.□

Of course, this result had been well-known by molecular biologists for at least 50 years; in their famous paper [25] Pauling, Corey and Branson noted that "*... It is*

likely that these [helical] configurations constitute an important part of the structure of both fibrous and globular proteins, as well as synthetic polypeptides". As it turned out, the helical structures are amazingly robust; they appear for large variety of molecules with widely ranging elastic properties and charge distributions. Lemma 4.2.2 provides a mathematical model for this well-established empirical fact. It is worth noting however that the molecules that are locally helical almost never stay in the shape of a perfect straight helix if they are sufficiently long. This *folding* of molecules is driven by a combination of the elastic and non-local forces and plays a very important role in the ensuing functionality of molecules. The instabilities of the helical states will be considered later using geometric approach.

Remark 4.2.3. *Lemma 4.2.2 includes, as a particular case, results from previous work on the helical solutions of the Kirchhoff's rod (no nonlocal terms) or uniformly charged Kirchhoff rod [18], with the charges*[10] *positioned at the rod's axis. In that case, the rigid charge conformations are positioned at* $\boldsymbol{\eta}(s) = 0$*, and thus*

$$d(s, s') = \left|\boldsymbol{\kappa}(s' - s)\right|.$$

In this case, the distance d *is independent of the mutual orientation* $\xi(s, s')$*, which is the first part of* $SE(3)$ *group element* $\sigma^{-1}(s)\sigma(s') := \big(\xi(s, s'), \boldsymbol{\kappa}(s, s')\big)$*. That is, charges arranged on the axis produce no torque.*

With this particular simplification, the computation still proceeds in the same way as before and reduction to algebraic equations still holds. Of course, more work is needed for each particular potential to demonstrate that the algebraic equations actu-

[01] Here, we use the word "charge" somewhat loosely to denote arbitrary potential interaction, like Morse or Lennard-Jones, and not just electrostatic potential.

ally have a solution, especially if complicated elasticity laws in the rod are assumed. Thus, Lemma 4.2.2 provides a simple justification for helical solutions that have been ubiquitous in the previous literature. However, our result goes further: helical shapes also allow one to search for helical solutions as solutions of algebraic equations for arbitrary charges off axis and for a charge distribution at each point of the molecule, possibly generating torque in the molecule.

4.3 Application to a linear rod with straight unstressed conformation

This section demonstrates how to find helical stationary shapes for charge bouquets positioned along an infinitely long, naturally straight molecule.

To illustrate the general ideas described above, we shall consider a molecule that consists of a chain of centerline atoms connected with elastic springs, and charges that are attached to each unit of the elastic chain with a rigid charge bouquet. In the undisturbed (base) configuration, the molecule is straight. In what follows, we select a charge bouquet containing two charges of $q_i = \pm 0.3e$, where e is the charge of electron. This arrangement approximates any molecule which has a constant dipole moment perpendicular to the axis, for example, vinylidene fluoride oligomers (VDF) [26]. More general charge configurations such as quadrupoles *etc.* can be incorporated by considering more general bouquets.

We assume electrical charges exhibit a screened electrostatic interaction

$$E_{ij,C}(s,s') = \frac{q_i(s)\, q_j(s')}{4\pi\varepsilon_0 d_{ij}(s,s')} e^{-d_{ij}(s,s')/\lambda}\,, \tag{4.25}$$

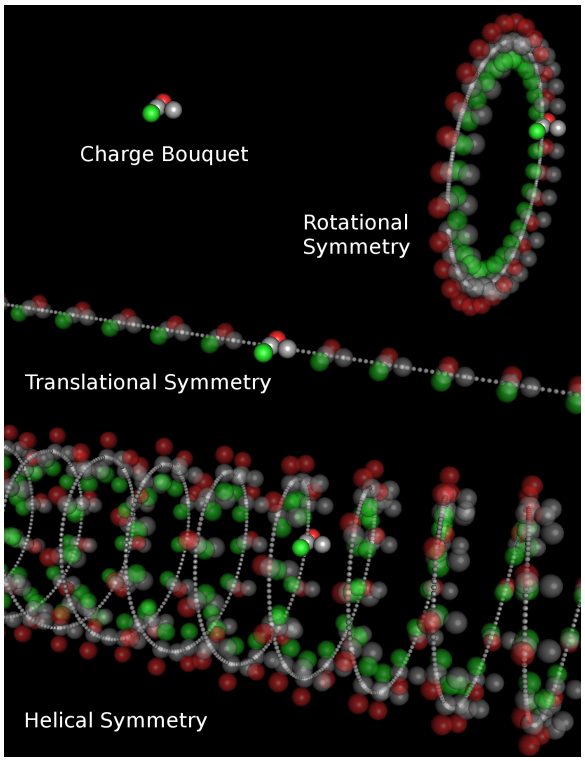

Figure 4.1: Top left: an example of a charge bouquet with four atoms. The electrically charged atoms are shown in red and green (positive or negative). Grey atoms are neutral, but still interact with other atoms through Lennard-Jones potential (4.34). The molecules we consider are long chains of these bouquets. Top right: a circular conformation obtained by applying rotational symmetry to the bouquet. Middle: A linear conformation obtained by applying the translational symmetry to the bouquet that are considered to be undisturbed configuration of the molecule. Bottom: A helical conformation obtained by simultaneous application of rotation and translation to the bouquet.

where d_{ij} is the distance between charges q_i at s and q_j at s', and λ denotes the Debye screening length. In what follows, we consider values of λ corresponding to various ionic strengths in the solution, from $I = 0.001$ M/l (close to de-ionized water) to $I = 10$ M/l (an order of magnitude higher than sea water). Debye length λ and ionic strength I are related by

$$\lambda = \sqrt{\frac{\epsilon_0 \epsilon_r k_B T}{2 N_A e^2 I}}\,, \tag{4.26}$$

where ϵ_0 is the permittivity of space, ϵ_r is dielectric constant of the media, k_B is Boltzmann's constant and N_A is Avogadro's number.

We seek stationary states that are invariant with respect to an affine (helical) transformation of space, which rotates a coordinate frame F by a matrix Λ, and performs a translation of an arbitrary vector $\boldsymbol{r}$ as

$$(F, \boldsymbol{r}) \to (\Lambda F, \Lambda \boldsymbol{r} + \mathbf{a}) \ . \tag{4.27}$$

In particular, we seek solutions with bouquets spaced uniformly on a helix of radius R and pitch C, with the rotation being parallel to the (x, y) plane. By *pitch*, we mean that after making one period of rotation, the atoms are moved by the distance C along the z-axis. The helix is then defined parametrically by

$$x = R\cos t\,, \quad y = R\sin t\,, \quad z = Ct\,, \tag{4.28}$$

or explicitly

$$x = R\cos z/C\,, \quad y = R\sin z/C\,, \tag{4.29}$$

where $R \geq 0$ and $C \neq 0$. The helix is right-handed for $C > 0$ and left-handed for $C < 0$. We use R and C as parameters in our calculations and visualizations. A continuous helix is a curve that is invariant under a set of transformations consisting of a rotation Λ about the $\hat{\mathbf{z}}$ axis and a translation $\mathbf{a}$, parameterized by z,

$$\Lambda(z) = \begin{bmatrix} \cos\left(\frac{z}{C}\right) & \sin\left(\frac{z}{C}\right) & 0 \\ -\sin\left(\frac{z}{C}\right) & \cos\left(\frac{z}{C}\right) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad \mathbf{a}(z) = \begin{bmatrix} R\sin\left(\frac{z}{C}\right) \\ R\cos\left(\frac{z}{C}\right) \\ z \end{bmatrix}. \tag{4.30}$$

For a discrete helix, we consider values of t, (respectively, z) that are discrete multiples

of some value T (respectively, z/C):

$$t = n\,T \quad \text{resp.} \quad z = n\frac{z}{C}\,, \qquad \text{where} \qquad n \in \mathbb{Z}\,. \tag{4.31}$$

Bouquets on the helix are oriented such that the helical invariance described above is maintained. Once a single bouquet is specified, at say $z = 0$, (4.30) generates the entire helix structure. Let $A \in SO(3)$ denote the orientation of the initial bouquet at $z = 0$. In what follows, we limit A to a twist about the tangent to the helix at $z = 0$, as shown in Fig. 4.2, allowing a single angle α to characterize bouquet orientation.

$$A(R, C, \alpha) = \begin{bmatrix} \cos\alpha & \frac{-C}{\sqrt{4\pi^2R^2+C^2}}\sin\alpha & \frac{2\pi R}{\sqrt{4\pi^2R^2+C^2}}\sin\alpha \\ \frac{C}{\sqrt{4\pi^2R^2+C^2}}\sin\alpha & \frac{4\pi^2R^2+C^2\cos\alpha}{4\pi^2R^2+C^2} & \frac{2\pi RC}{4\pi^2R^2+C^2}(1-\cos\alpha) \\ \frac{-2\pi R}{\sqrt{4\pi^2R^2+C^2}}\sin\alpha & \frac{2\pi RC}{4\pi^2R^2+C^2}(1-\cos\alpha) & \frac{4\pi^2R^2\cos\alpha+C^2}{4\pi^2R^2+C^2} \end{bmatrix}. \tag{4.32}$$

The entire helical structure may then be generated by repeated application of (4.30).

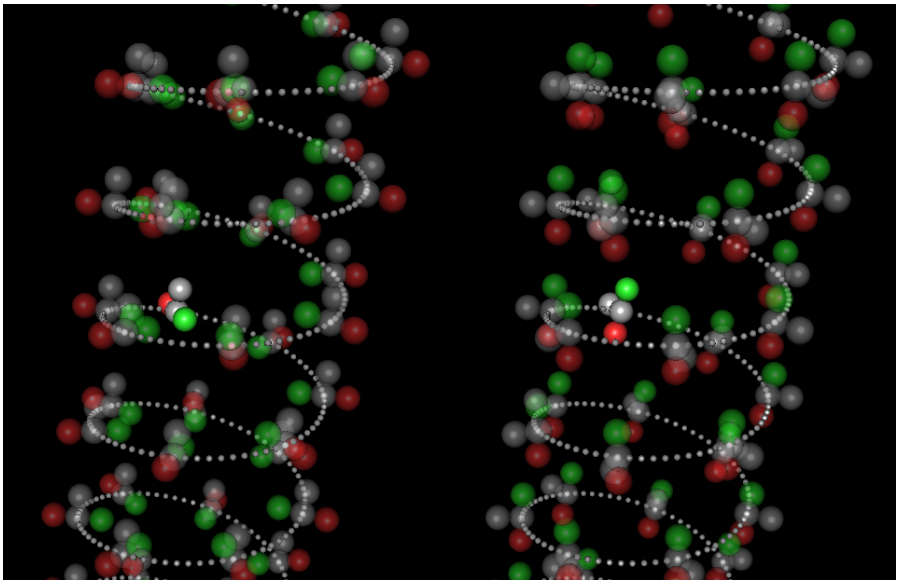

Figure 4.2: Effect of twist transformation - the twisted bouquet is highlighted.

4.4 Energy of a helical conformation

This section explains the interaction energies of a realistic molecule to be used in calculations, both the local (elastic) parts and non-local (electrostatic and Lennard-Jones). As discussed above, we investigate the particular example of a molecule of bouquets with two opposite charges $\pm q$ positioned on either side of the polymer axis, each a distance l from that axis. This simplified model describes a polymer with a constant polarization perpendicular to the polymer's axis. The interaction energies are defined as follows.

Elastic energy. We consider elastic energy as quadratic in bond bend angle,

$$E_{elastic} = \frac{1}{2}\mu\big(\Delta\phi\big)^2, \tag{4.33}$$

where μ is a spring constant, and $\Delta\phi$ is the bond bend angle with respect to its unstressed orientation *in the intrinsic frame*, assumed straight. For naturally helical molecules, $\Delta\phi$ would be the value of rotation angle for a bouquet about its rotation axis in the intrinsic frame from its equilibrium value. In terms of global geometry that would mean that bend energy is punishing creation of local curvature. A value of $\mu = 3.025 \cdot 10^5 J/(rad^2)Mol = 5.022 \cdot 10^{-19} J/rad^2$ is typical for bend rigidity for a carbon-carbon bond. The bend depends (through a complex algebraic formula that we do not present here) on the radius and pitch (axial distance traversed on each rotation) of the helix.

Lennard-Jones energy. To prevent self-intersection, we introduce a truncated Lennard-Jones interaction between charge bouquet nodes with equilibrium distance

d_0 and potential well depth ε. When the centers of two bouquet nodes are a distance d from each other, they experience a potential given by

$$E_{LJ} = \begin{cases} \varepsilon \left[\left(\frac{d_0}{d}\right)^{12} - 2\left(\frac{d_0}{d}\right)^{6} - \frac{1}{3^{12}} + \frac{2}{3^6} \right] & d < 3\,d_0 \\ 0 & d \geq 3\,d_0 \end{cases} . \tag{4.34}$$

Charge potential energy. As mentioned when the model was introduced, each bouquet has charges of $q_{1,2}(s) = \pm 0.3e$ that interact with each other through the screened Coulomb potential (4.25) with Debye length λ.

Total energy of a given conformation. Given a bouquet configuration and values for R, C, and α, we compute the total energy by choosing a reference bouquet, then working outward along the helix in both directions by helical symmetry. For each bouquet we reach, its contribution is the sum of the Lennard-Jones and Coulomb interactions of the bouquet nodes with the reference bouquet nodes. The sum of these, plus the elastic energy corresponding to the given R and C give the energy per bouquet (energy density) of the given conformation.

As an example, we demonstrate the particular energy landscape for $I = 0.001M/l$ in Fig. 4.3. The vertical axis is helix radius R, the horizontal axis is helix pitch C, with $(R, C) = (0, 0)$ in the upper left corner. The energy scale bar is shown on the right, with lowest energies in black, highest in white. The left edge and the upper left-hand corner contain most of the energy minima. The minima along the left-hand edge, corresponding to small pitch C (on the order of the size of the bouquet), corresponds to the helices solutions that are being "ratcheted" into tighter and tighter conformations as we proceed upward (smaller R). The values of the energy in this graph are minima over all twists (values of α) of the bouquet, as shown in Fig. 4.1.

The local minima of this energy landscape are stable helical conformations.

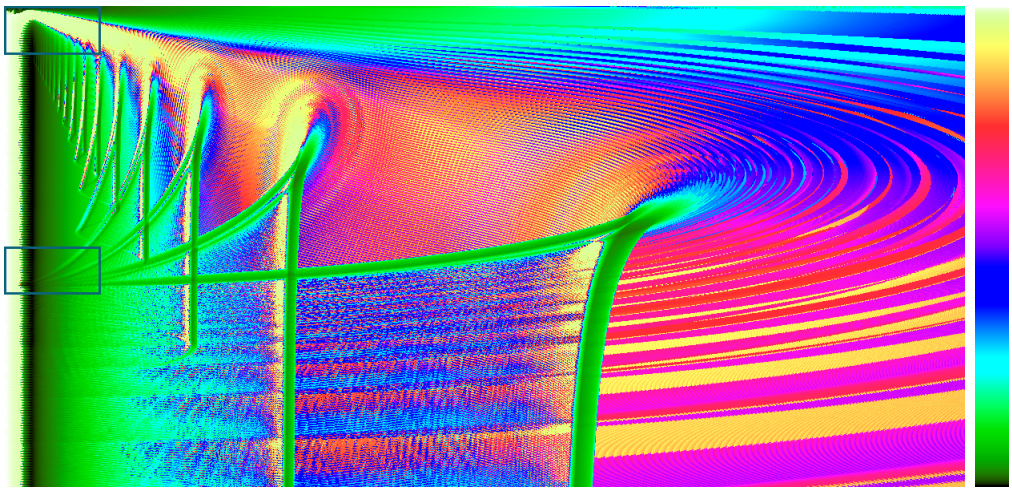

Figure 4.3: The energy landscape for a given ionic strength $I = 0.001M/l$. The vertical coordinate is radius R and the horizontal coordinate is pitch C. Energy is also dependent on the rotation of the bouquet around the axis α, but this coordinate has been projected out onto two dimensions by taking a minimum over all rotation angles for the bouquet. That projection preserves the minima of the energy. Notice that this energy plot provides a complete analysis for all helical conformations. Helical conformations that are potential minima are concentrated on the left-hand side (small pitch, increasing radius) and upper left-hand corner (small radius, increasing pitch. Below, Figure 4.4 shows blow-ups of the framed rectangular regions.

In order to further elucidate the structure of the energy landscape, Fig. 4.4 presents a blow-up of the two boxes in Fig. 4.3. Yellow dots correspond to energy minima and thus show stationary conformations. Since we assume that the base conformation of our molecule is linear, a weakly ionized solution leads to longer electrostatic interaction, resulting in richer structure. Helical conformations become increasingly unlikely for strongly ionized solutions that make long-range electrostatic forces weaker. For strongly ionized solutions, elastic forces tend to unwind the helices into straight lines.

As shown in Fig. 4.4, even this simple molecule, when deformed into helical config-

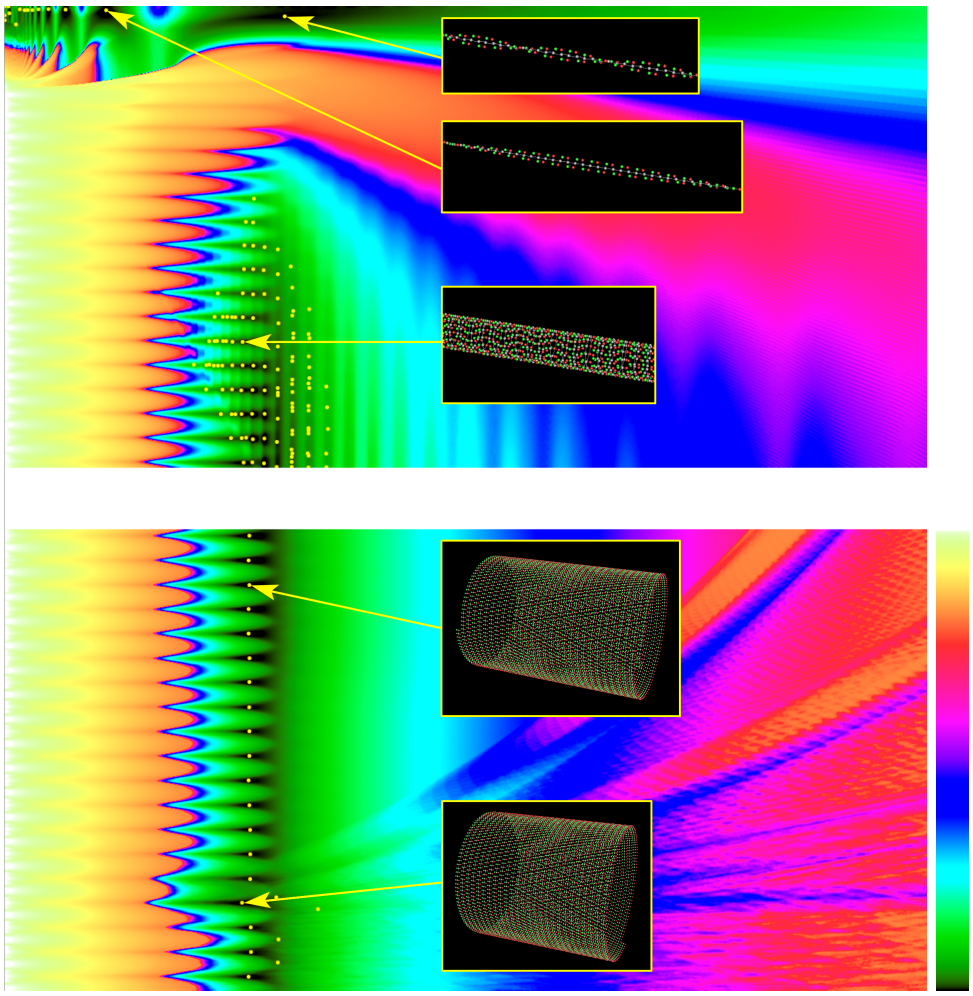

Figure 4.4: Blow-up of energy landscape for the boxes in Fig. 4.3. The energy scale has been adjusted to show more detail, and is shown on the right with lowest energies in black, highest energies in white. The vertical coordinate is radius R and horizontal coordinate is pitch C. Top: upper left corner box in Fig. 4.3. Bottom: box in the middle of left edge in Fig. 4.3. Several examples of the helical conformations corresponding to different energy minima are presented as inserts with yellow arrows indicating the corresponding energy minima.

urations, shows quite a complex and intriguing energy landscape. While the details of the landscape depend on the individual molecular parameters, the presence of "ratcheting" helical states is, we believe, typical for all molecules of this type. In this

figure, several examples of the helical conformations corresponding to different energy minima are presented as inserts. There are two types of helical conformations. The "twist" type is obtained by twisting a base straight conformation, and is characterized by small radius R and increasing pitch C. In a given energy landscape, there are only finitely many conformations of this type, as increasing the twist beyond certain level will cause elastic forces that are too large to be balanced by electrostatic attraction. We show two examples of such twist conformations in the top part of this figure. Another type of helical solution is given by the ratcheting states, characterized by a pitch on the order of bouquet's size and increasing radius. This is shown both in the top and bottom parts of the figure. There are infinitely many of these ratcheting states, as the radius can increase (in principle) to arbitrarily large values. Energy minima then occur every time the opposite charges line up on sequential rolls of the helix. That explains the regularity of occurrence of those minima. They occur every time the helix circumference increases by the distance separating the bouquets, l. This leads to an increase of radius between neighboring ratcheting conformations as $\Delta R = l/(2\pi)$.

To illustrate how varying ionic strength of the solution changes the energy landscape of helical conformations, we compute 10 energy landscapes varying from $I = 0.001M/l$ to $I = 0.16M/l$. These energy landscapes are presented, with the same orientation and axes as in Fig. 4.5 (where the ionic strengths are labeled in M/ml. From this, it is clear that the variation of ionic strength and hence Debye length, has a profound effect on the energy landscape. As we see below, the ionic strength is also of particular importance for linear stability.

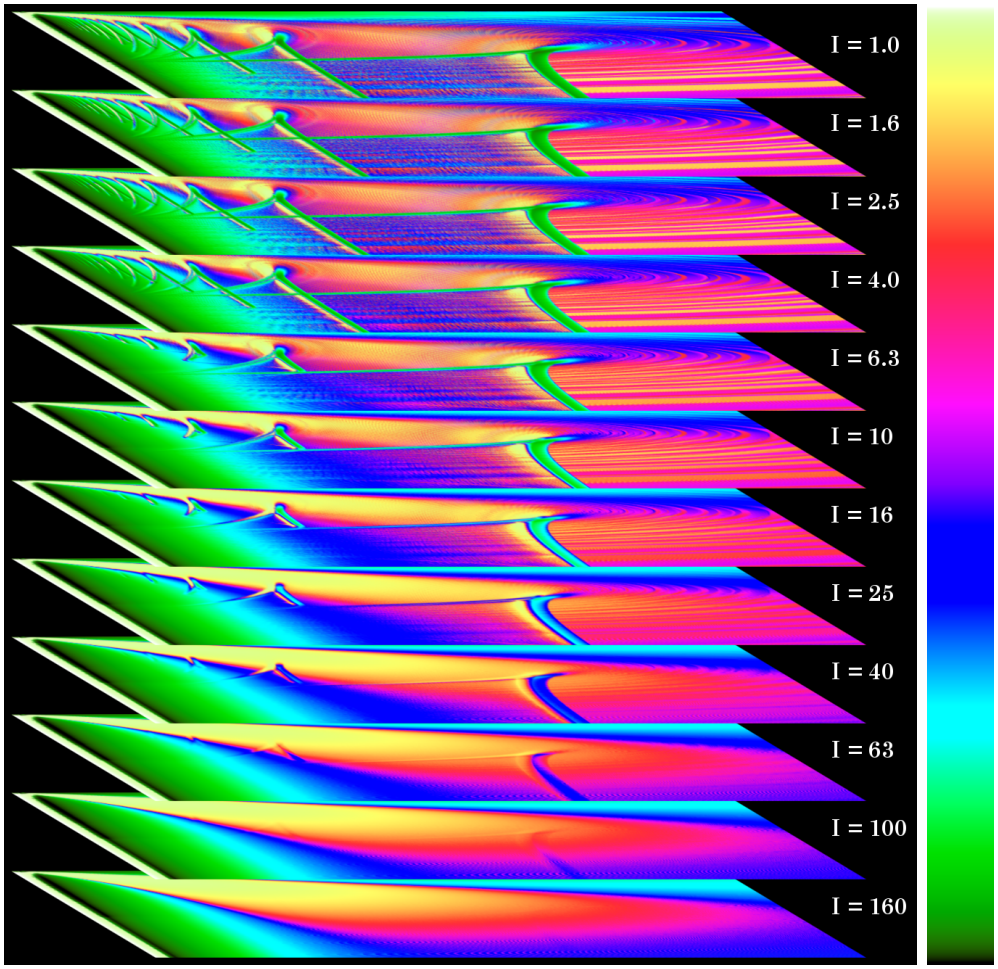

Figure 4.5: (Supplementary Figure) Energy landscapes for different values of ionic strengths I, expressed in Moles/l. A value $I = 0.001M/l$ is a very weakly ionized solution, whereas $I = 0.16M/l$ is a rather strongly ionized solution.

4.5 Multiple helical conformations

In this section, we explore generalizations of the helical shapes considered above. We show how to compute a particular molecular shape that we refer to as a *2-helix* (and, in general, n-helix). By 2-helix we mean a conformation where a group consisting of two bouquets with some relative position and orientation is repeated along a helix.

Alternatively, a 2-helix can be seen as two helices of single bouquets arranged so the polymer axis passes through bouquets from alternating helices over its length. The relative orientation of bouquets in the group can be arbitrary. Such structures have been suggested and analyzed in [27] in the context of describing given molecular configurations: β-helix structure (2-helix) or collagen (3-helix). These structures may be obtained as natural energy minima for a given molecule. In particular, we present an example of an equilibrium configuration of a 2-helix using the bouquet considered in the previous section. An extension of these ideas is possible for n-helices with $n > 2$. In general, n-helices are combinations of n helical conformations, with 2, 3, ..., n being obtained from the first one by a shift, rotation and a twist of the base bouquet. A detailed map of the energy landscape for an n-helix is problematic even for moderate n, because the number of parameters involved in defining the conformation of such a shape may increase prohibitively with n, as the position and orientation of each bouquet is required in defining a repeated group in an n-helix.

For a 2-helix, we optimize the energy over both the shape of the helix and the relative configuration of two bouquets. A minimal energy over these variables will, by the helical symmetry, provide an exact stationary conformation for the whole molecule. Two examples of such conformations are given in Fig. 4.6. Details of calculations of more general multi-helices will be discussed elsewhere.

4.6 Linear stability analysis

This section derives the dispersion relations for the linear stability of helical polymers, based on the linearizations of (4.22) and (4.23) about a helical state that is assumed

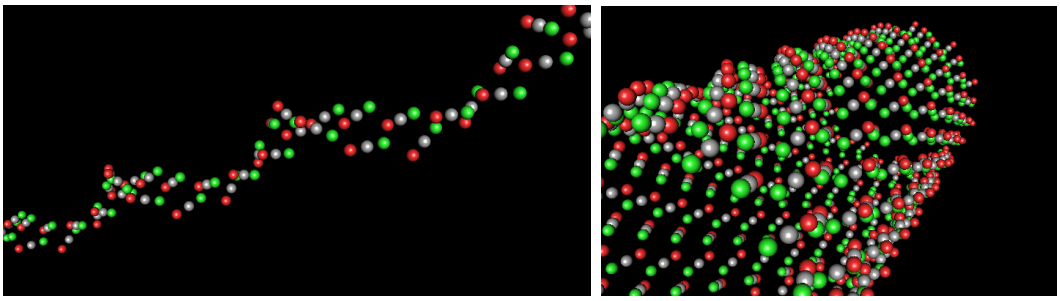

Figure 4.6: 2-helix conformations for the ionic strength I= 0.001 M/l, and red/green charges in each bouquet of $\pm 0.8e$. The elastic centerline (not shown) goes through gray spheres that are the centers of the bouquets. The conformation is not a perfect helix, but instead consists of two helices, in which the second helix has undergone a twist, shift and rotation with respect to the first one. The distance between the bases of the bouquets (gray spheres) is 2 Å.

to be a stationary solution. As we show in this section, the use of geometric methods yields an exact dispersion relation.

The energy landscape for charged polymers with nonlocal interactions is complex, even when limited to helical shapes, because of the large number of energy minima in that "helical universe". It is natural to pose the question of linear stability and selection of the stationary shapes. Here, the geometric approach allows us to find the exact dispersion relation for the stability of the helical shapes. As far as we know, no such work has been undertaken before, perhaps because of the complexity of the linear stability analysis when using traditional methods.

By using an $SE(3)$ method that naturally accounts for the helical geometry, one may investigate the linear stability of a general helical polymer with torque-inducing non-local self-interactions. Studies of the stability of helical elastic rods have been undertaken before, with most studies concentrating on the linearization of the traditional Kirchhoff equations about the stationary helical states [11, 28–32]. The focus

of these works was on the stability analysis based on increasingly complex elastic properties of the rod. Alternatively, the work [33] investigated the stability of elastic rods using the exact geometric rod theory and applied it to DNA dynamics. All these works have used the continuous model of elastic rods. Our results differ in two ways from previous studies. First, our results are formulated for spatially discrete rods, as discussed in the previous sections. Second, and more importantly, our stability analysis includes non-local interaction of charges that in general occupy positions *off the axis* of the elastic rod. The forces on these charges generate torques acting on the rod's centerline. These torques are absent when one considers purely elastic rods, or when the charges are distributed only along the centerline.

This section shows that the presence of the torques due to non-local interaction of off-axis charges generates an *instability* of the rod and that the instability appears even for the simplest possible states – the linear rod. The instability due to nonlocal torques is new, as far as we are aware.

Obtaining similar results using Kirchhoff's rod equations would be problematic. The main difficulty consists in finding the Euclidean distance between two arbitrary points on a rod's axis using coordinates intrinsic to the rod. In addition, the present computation shows the geometric origin of the exact dispersion relation for arbitrary helical configuration of a rod (without charges), something that was noticed already in [11]. In our opinion, the present method based on exact geometric theory is more straightforward, algorithmic and compact than the corresponding linear analysis of Kirchhoff equations.

Suppose we have a helical configuration arising from successive repetition of a given element $a \in SE(3)$, so that $\sigma(s) = a^s$ and $\Xi(s, s') = a^{s'-s}$. Let us linearize

about this solution, so

$$\sigma(s) = a^s + \epsilon a^s \psi_1(s,t) + O(\epsilon^2)\,. \tag{4.35}$$

The linearization written symbolically as $\sigma = \sigma_0 + \epsilon\sigma_1$ is defined by

$$\psi_1 = \sigma_0^{-s}\sigma_1(s) := a^{-s}\sigma_1(s) \in \mathfrak{se}(3)\,. \tag{4.36}$$

As it turns out, this substitution leads to an exact dispersion relation for σ_1. We define the $O(\epsilon)$ perturbations in velocity and deformation as follows. First, perturbations of the velocity are determined, as

$$\mu(s,t) = \epsilon\mu_1(s,t) + O(\epsilon^2)\,, \quad \mu_1 = \sigma_0^{-1}\dot{\sigma}_1 = \dot{\psi}_1 \in \mathfrak{se}(3)\,. \tag{4.37}$$

In the discrete case,

$$\lambda(s,t) = a + \epsilon\lambda_1 + O(\epsilon^2)\,, \tag{4.38}$$

where

$$\lambda_1 = -\sigma_0^{-1}(s)\sigma_1(s)a + \sigma_0^{-1}(s)\sigma_1(s+1) = -\psi_1(s)a + \psi_1(s+1) \in \mathfrak{se}(3)\,. \tag{4.39}$$

In the corresponding continuous case, one denotes $\lambda_0 = \sigma^{-1}(s)\sigma'(s) = \Gamma$ and finds,

$$\begin{aligned} \lambda(s,t) &= \Gamma + \epsilon\lambda_1 + O(\epsilon^2) \\ \lambda_1 &= -\sigma_0^{-1}\sigma_1\Gamma + \sigma_0^{-1}\sigma_1'(s) = -\psi_1\Gamma + \psi_1' \in \mathfrak{se}(3)\,. \end{aligned} \tag{4.40}$$

For simplicity (in order to keep the formulas compact), we shall assume for the discrete case that

$$\frac{\delta l}{\delta \mu} = \Pi_0 + \epsilon I\mu_1 + \dots\,, \quad \lambda^{-1}\frac{\delta l}{\delta \lambda} = K_0 + \epsilon K_1(s) + \dots\,. \tag{4.41}$$

Next, we compute the linearization of the nonlinear terms due to elasticity. In order to find the linearization of the sum of terms

$$\mathrm{Ad}^*_{\lambda^{-1}(s)}\Big(\lambda^{-1}(s)\frac{\delta l}{\delta \lambda}(s)\Big) + \lambda^{-1}(s-1)\frac{\delta l}{\delta \lambda}(s-1)\,, \tag{4.42}$$

in equation (4.22) we utilize the following proposition.

Proposition 4.6.1. *Suppose G is a Lie group with Lie algebra $\mathfrak{g}$ and $\langle \cdot\,,\,\cdot \rangle : \mathfrak{g}^* \times \mathfrak{g} \to \mathbb{R}$ is a pairing between the Lie algebra and its dual. Suppose $A(\epsilon) \in G$ is a curve in G with $A(0) = A_0$ and $A_0^{-1}A'(0) = a \in \mathfrak{g}$, $\alpha(\epsilon) \in \mathfrak{g}^*$, and $\alpha'(0) = \xi$. Then (see, for example, [23], p.60)*

$$\frac{\partial}{\partial \epsilon}\mathrm{Ad}^*_{A^{-1}(\epsilon)}\alpha(\epsilon)\bigg|_{\epsilon=0} = \mathrm{Ad}^*_{A_0^{-1}}\Big(\xi - \mathrm{ad}^*{}_a\alpha_0\Big)\,. \tag{4.43}$$

Note that $\lambda^{-1}\delta l/\delta\lambda$ has the physical meaning of the local stress in the body coordinate frame at point s.

Then, writing

$$\lambda^{-1}\frac{\delta l}{\delta \lambda}(s) = K_0 + \epsilon K_1(s) + O(\epsilon^2)\,,$$

where $K_0\,, K_1 \in \mathfrak{se}(3)^*$. Hence, the linearization of (4.42) is computed as follows

$$\begin{aligned}\frac{\partial}{\partial \epsilon}\bigg|_{\epsilon=0}&\left(- \mathrm{Ad}^*_{\lambda^{-1}(s)}\Big(\lambda^{-1}(s)\frac{\delta l}{\delta \lambda}(s)\Big) + \lambda^{-1}(s-1)\frac{\delta l}{\delta \lambda}(s-1)\right)\\ &= -\mathrm{Ad}^*_{\lambda_0^{-1}}\Big(K_0 - \mathrm{ad}^*_{\psi_1}K_1(s)\Big)(s) + K_1(s-1)\,,\end{aligned} \tag{4.44}$$

where $\psi_1(s) := \lambda_0^{-1}(s)\lambda_1(s) =\in \mathfrak{se}(3)$.

The linearization of the nonlocal terms in equation (4.22) is less straightforward and will be outlined in its own section below. For now, we assume it is possible to compute that linearization, and it is described by some linear operator $\mathbb{L}(\Xi(s,s'))\psi_1(s)$, which is defined as follows. Consider an arbitrary $\eta \in \mathfrak{se}(3)$ and define the *scalar* function of s by the following pairing,

$$I(s) = \int \Big\langle - \frac{\delta U}{\delta \Xi}(s,s')\Xi(s,s') + \Xi^{-1}(s,s')\frac{\delta U}{\delta \Xi}(s',s)\,,\,\eta\Big\rangle \mathrm{d}s'\,. \tag{4.45}$$

The nonlocal term takes values in the space $\mathfrak{se}(3)^*$, so the pairing in (4.45) indeed defines a scalar function. We need to find its linearization with respect to $\psi = \sigma^{-1}\delta\sigma \in \mathfrak{se}(3)$. For this, we compute the derivative of $I(s)$ with respect to Ξ according to

$$\delta I = \Big\langle \frac{\delta I(\Xi,\eta)}{\delta \Xi}\,,\, \Xi_1 \Big\rangle = \Big\langle \Xi^{-1}\frac{\delta I(\Xi,\eta)}{\delta \Xi}\,,\, \Xi^{-1}\Xi_1 \Big\rangle\,, \tag{4.46}$$

where $\psi_1 = \Xi^{-1}\Xi_1 \in \mathfrak{se}(3)$ is the linearization with respect to Ξ. To complete this calculation, we need to express $\Xi^{-1}\Xi_1$ in terms of ψ_1. This step proceeds as follows. The linearization of $\Xi(s,s')$ in (4.24) gives

$$\Xi(s,s') = a^{s'-s} + \epsilon\Xi_1(s,s') + O(\epsilon^2)\,, \tag{4.47}$$

where

$$\begin{aligned}\Xi_1(s,s') =& -a^{-s}\sigma_1(s)a^{s'-s} + a^{-s}\sigma_1(s') \\ =& -\psi_1(s)a^{s'-s} + a^{s'-s}\psi_1(s') \\ =& a^{s'-s}\big(\psi_1(s') - \mathrm{Ad}_{a^{s-s'}}\psi_1(s)\big)\,.\end{aligned} \tag{4.48}$$

Consequently, the quantity $\Xi^{-1}\Xi_1$ is given by

$$\Xi^{-1}\Xi_1 = -\mathrm{Ad}_{a^{s-s'}}\psi_1(s) + \psi_1(s')\,, \tag{4.49}$$

and we have

$$\begin{aligned}\delta I = \Big\langle -\mathrm{Ad}^*_{a^{s-s'}}\Big(\Xi^{-1}(s,s')\frac{\delta I(\Xi(s,s'),\eta)}{\delta\Xi}\Big) \\ + \Xi(s',s)\frac{\delta I(\Xi(s',s),\eta)}{\delta\Xi}\,,\,\psi_1\Big\rangle.\end{aligned} \tag{4.50}$$

Finally, since $I(s)$ in (4.45) is a linear function of an arbitrary η, re-arranging expression (4.50) into a scalar product of an $\mathfrak{se}(3)^*$-valued function with η will give the desired linearization operator $\mathbb{L}(\Xi(s,s'))\psi_1(s)$, from

$$\delta I =: \big\langle \mathbb{L}(\Xi(s,s'))\psi_1(s)\,,\,\eta\big\rangle. \tag{4.51}$$

The equations simplify further upon noticing that for a stationary helical solution, $\sigma = \sigma_0 a^s$ where $a \in SE(3)$ is a given element, so the expression $\lambda_0 = \sigma_0^{-1}(s)\sigma(s+1) = a$ is independent of s. Then, the linearization of equation (4.22) in the discrete case is

$$\begin{aligned} -\frac{\partial^2}{\partial t^2} I\psi_1 - \mathrm{Ad}^*_{a^{-1}}\Big(K_1(s) - \mathrm{ad}^*_{\psi_1}(s)K_0\Big)(s) + K_1(s-1) \\ = \sum_{s',m,k} \mathbb{L}\big(\Xi(s,s')\big)\psi_1(s)\,. \end{aligned} \tag{4.52}$$

It is natural to posit the following ansatz:

$$K_1(s) = a^{-s}\Big[J\big(\psi_1(s+1) - \psi_1(s)\big)\Big]a^s := \mathrm{Ad}^*_{a^s}\Big(J\big(\psi_1(s+1) - \psi_1(s)\big)\Big)\,, \tag{4.53}$$

where $\psi_1(s) \in \mathfrak{se}(3)$ and $J : \mathfrak{se}(3) \to \mathfrak{se}(3)^*$ is a linear operator having the physical meaning of the rigidity matrix. Notice that the linearized system of coordinates is written at the point s, but it encounters the value of the stress at the point $s-1$. In order to connect this stress with the coordinate system at the point s, we will need to transform the coordinates to $s-1$, by shifting one step forward on the helix. We thus need to compute $\mathrm{Ad}^*_{a^{-1}}$ of the term evaluated at $s-1$, *i.e.*

$$K_1(s-1) = \mathrm{Ad}^*{}_{a^{-1}}\Big[J\big(\psi_1(s) - \psi_1(s-1)\big)\Big].$$

Then, the linearization of the discrete case gives

$$\begin{aligned} -\frac{\partial^2}{\partial t^2} I\psi_1 - \mathrm{Ad}^*_{a^{-1}}\Big[J\Big(\psi_1(s+1) - 2\psi_1(s) + \psi_1(s-1)\Big)\Big] \\ + \mathrm{ad}^*_{\psi_1(s)}K_0 = \sum_{s',m,k} \mathbb{L}\big(\Xi(s,s')\big)\psi_1(s)\,. \end{aligned} \tag{4.54}$$

In the continuous case, the corresponding linearization of equation (4.23) gives

$$-\frac{\partial^2}{\partial t^2} I\psi_1 + \Big(-\frac{\partial}{\partial s} + \mathrm{ad}^*_\Gamma\Big)J\psi_1' + \mathrm{ad}^*_{\psi_1}\Gamma = \sum_{m,k}\int \mathbb{L}\big(\Xi(s,s')\big)\psi_1(s)\mathrm{d}s'\,. \tag{4.55}$$

Here, again $\psi = \lambda_0^{-1}(s)\lambda_1(s) \in \mathfrak{se}(3)$.

Further simplification can be obtained for the nonlocal term for the stationary helical state $\sigma(s) = \sigma_0 a^s$. The invariant variable $\Xi = \sigma^{-1}(s)\sigma(s') = a^{s-s'}$ depends only on the difference between s and s'. Thus, all of the derivatives of the potential energy with respect to Ξ when evaluated at the helical configuration depend only on the difference between s and s'. In other words, we have

$$\begin{aligned} &\Xi_0^{-1}\frac{\delta U}{\delta \Xi}_0(s,s') = D_1(s-s') \in \mathfrak{se}(3)^*\,, \\ &\Xi_0^{-1}\frac{\delta}{\delta \Xi_0}\Big(\Xi_0^{-1}\frac{\delta^2 U}{\delta \Xi}_0(s,s')\Big) = D_2(s-s')\,, \end{aligned} \tag{4.56}$$

with $D_2\alpha \in \mathfrak{se}(3)^*$ for any $\alpha \in \mathfrak{se}(3)$, so $D_2 : \mathfrak{se}(3) \to \mathfrak{se}(3)^*$.

Since all the functions on the right-hand side depend only on the difference $s - s'$, the integrals or sums become convolution integrals. Fourier transforming then allows the exact dispersion relation to be obtained, as follows. Let us consider

$$\psi_1(s,t) = Se^{-i\omega t + iks}\,, \quad S \in \mathfrak{se}(3)\,. \tag{4.57}$$

Here s is an integer and k is the dimensionless wave number, measured in the units of $2\pi/l_0$, where l_0 is the distance between the elements of the helical chain. Consequently, assuming S is real, the linearized equation (4.54) gives the following dispersion relation:

$$\omega^2 IS - 4\big(\sin^2\frac{k}{2}\big)\mathrm{Ad}^*_{a^{-1}}\Big(JS - \mathrm{ad}^*_S K_0\Big) = \sum_{s',m,k}\mathbb{L}\big(s'\big)S\,. \tag{4.58}$$

Note that in the absence of non-local interactions, the basic helix must be unstressed, $K_0 = 0$, so $\omega^2 = \lambda$ are given by the generalized eigenvalues of the problem

$$4\sin^2\frac{k}{2}\mathrm{Ad}^*_{a^{-1}}\Big(JS\Big) = I\lambda S\,. \tag{4.59}$$

From physical principles, we require all the generalized eigenvalues of the matrices J and I to satisfy $\lambda = \omega^2 > 0$, so that all purely elastic helices in stationary conformations are neutrally stable. The spatially discrete dispersion relation (4.58) converges to the dispersion relation for the continuum case in the limit $k \to 0$, $a \to \mathrm{Id}_{SE(3)}$, which is

$$k^2 \mathrm{Ad}^*_{a^{-1}}\Big(JS\Big) = I\lambda S\,. \tag{4.60}$$

Again, the right-hand side of (4.58) is a function of $s' - s$ only, while the left-hand side is a constant. Upon summation over s', the dependence on s disappears and the dispersion relation is obtained by setting $s = 0$ on the right-hand side:

$$\omega^2 IS - \big(\sin^2\frac{k}{2}\big)\mathrm{Ad}^*_{a^{-1}}\Big(JS - \mathrm{ad}^*_S K_0\Big) = \sum_{s',m,k} \mathbb{L}\big(s'\big)S\,. \tag{4.61}$$

The right-hand side is a linear operator acting on S. Instability corresponds to generalized eigenvalues ω of equation (4.61) having a positive imaginary part. As it turns out, all eigenvalues $\lambda = \omega^2$ are real, so it is enough to identify the case $\lambda < 0$ as instability. However, the consideration of discrete rods puts an interesting spin on this problem that we will consider below.

4.7 Computation of potential energy

Equation (4.61) provides the stability analysis for an arbitrary interaction potential $U(d)$. However, for particular applications it is important to explicitly compute the linearization operators in (4.61). Again, geometric methods will be advantageous. Therefore, this section computes the linearization of the nonlocal terms for the dispersion relation. We believe it is advantageous to show this computation in some

detail, as it is not trivial.

We shall perform the computation only for the discrete case. The continuous case is derived similarly with the change of sums with respect to s and s' into integrals where necessary. One has to be careful here, as we need to take derivatives of quantities that take values in $T_eSE(3)$ and $T_eSE(3)^*$. The most straightforward way, least likely to lead to a mistake, is to define a corresponding scalar functional by bringing these quantities to the Lie algebra and then pairing them with the corresponding *fixed* element from the dual. The derivatives will then be given by whatever term is paired the chosen fixed element. This is akin to the weak computations of functional derivatives, only performed with geometric quantities.

First, notice that

$$\frac{\delta U}{\delta \Xi}(s,s')\Xi^{-1}(s,s') = \mathrm{Ad}^*_{\Xi^{-1}(s,s')}\Big(\Xi^{-1}(s,s')\frac{\delta U}{\delta \Xi}(s,s')\Big)\,, \tag{4.62}$$

and

$$\Xi(s,s')\frac{\delta U}{\delta \Xi}(s',s) = \Big(\Xi^{-1}(s,s')\frac{\delta U}{\delta \Xi}(s,s')\Big)\Big|_{s\leftrightarrow s'}\,. \tag{4.63}$$

Thus, we start the computation of the linearization operator D_2 for the nonlocal term by linearizing the expression $\Xi^{-1}\frac{\delta U}{\delta \Xi} \in \mathfrak{se}(3)^*$. Let us consider $\Xi = (\xi, \boldsymbol{\kappa})$ with $\Xi^{-1} = (\xi^{-1}, -\xi^{-1}\boldsymbol{\kappa})$, an arbitrary element $(\boldsymbol{\phi}, \boldsymbol{\Psi}) \in \mathfrak{se}(3)$ and a scalar functional

$$I_1 = \Big\langle \Xi^{-1}\frac{\delta U(d_{km})}{\delta \Xi}\,,\,(\boldsymbol{\mu},\boldsymbol{\alpha})\Big\rangle. \tag{4.64}$$

The linearization operator D_2 is therefore defined as

$$\Big\langle D_2(\boldsymbol{\phi},\boldsymbol{\Psi})\,,\,(\boldsymbol{\mu},\boldsymbol{\alpha})\Big\rangle := \Big\langle \Xi^{-1}\frac{\delta I_1(\boldsymbol{\mu},\boldsymbol{\alpha})}{\delta \Xi}\,,\,(\boldsymbol{\phi},\boldsymbol{\Psi})\Big\rangle. \tag{4.65}$$

Here, we introduced a natural pairing between two elements $(a,\boldsymbol{b}) \in T_eSE(3)^*$ and

$(\alpha, \boldsymbol{\beta}) \in T_e SE(3)$:

$$\big\langle (a, \boldsymbol{b})\,,\, (\alpha, \boldsymbol{\beta}) \big\rangle = \frac{1}{2}\mathrm{tr}\big(a^T \alpha\big) + \boldsymbol{b} \cdot \boldsymbol{\beta}\,. \tag{4.66}$$

Given

$$d_{km}\big(\Xi(s,s')\big) = d_{km}\big(\xi, \boldsymbol{\kappa}\big) = \big|\boldsymbol{\kappa} + \boldsymbol{\eta}_k(s) - \xi(s,s')\boldsymbol{\eta}_m(s')\big|\,,$$

we have

$$\begin{aligned} I_1 :&= \Big\langle \Xi^{-1}\frac{\delta U}{\delta \Xi}\,,\, \big(\boldsymbol{\mu}, \boldsymbol{\alpha}\big)\Big\rangle \\ &= \frac{U'(d_{km})}{d_{km}}\Big[\mathrm{tr}\Big(- \big(\xi^{-1}\mathbf{d}_{km} \otimes \boldsymbol{\eta}_m\big)^T \hat{\mu}\Big) + \big(\xi^{-1}\mathbf{d}_{km}\big)\cdot \boldsymbol{\alpha}\Big] \\ &= \frac{U'(d_{km})}{d_{km}}\Big[- \big(\xi^{-1}\mathbf{d}_{km} \times \boldsymbol{\eta}_m\big)^T \cdot \boldsymbol{\mu} + \big(\xi^{-1}\mathbf{d}_{km}\big)\cdot \boldsymbol{\alpha}\Big] \\ &:= Q\big(d_{km}\big) I_2\,, \end{aligned} \tag{4.67}$$

since $(\hat{\mu})_{ij} = \epsilon_{ijk}\mu_k$ by the definition of the hat map. Here, we have defined

$$\begin{aligned} &Q\big(d_{km}\big) := \frac{U'(d_{km})}{d_{km}}\,, \\ &I_2(\Xi, \boldsymbol{\mu}, \boldsymbol{\alpha}) := -\big(\xi^{-1}\mathbf{d}_{km} \times \boldsymbol{\eta}_m\big)^T \cdot \boldsymbol{\mu} + \big(\xi^{-1}\mathbf{d}_{km}\big)\cdot \boldsymbol{\alpha}\,. \end{aligned} \tag{4.68}$$

In order to compute the linearization D_2, we proceed as follows. For $(\boldsymbol{\phi}, \boldsymbol{\Psi}) \in \mathfrak{se}(3)$, calculate

$$\begin{aligned} \Big\langle \Xi^{-1}\frac{\delta I_1}{\delta \Xi}\,,\, \big(\boldsymbol{\phi}, \boldsymbol{\Psi}\big)\Big\rangle = \frac{Q'(d_{km})}{d_{km}} I_2(\Xi, \boldsymbol{\mu}, \boldsymbol{\alpha}) I_2(\Xi, \boldsymbol{\phi}, \boldsymbol{\Psi}) \\ + Q(d_{km})\Big\langle \Xi^{-1}\frac{\delta I_2}{\delta \Xi}\,,\, \big(\boldsymbol{\phi}, \boldsymbol{\Psi}\big)\Big\rangle. \end{aligned} \tag{4.69}$$

We still need to compute the variational derivative of $I_2(\Xi, \boldsymbol{\mu}, \boldsymbol{\alpha})$. The only part of I_2 depending on $\Xi = (\xi, \boldsymbol{\kappa})$ is the quantity

$$\xi^{-1}\mathbf{d}_{km} = \xi^{-1}\big(\boldsymbol{\kappa} + \boldsymbol{\eta}_k\big) - \boldsymbol{\eta}_m\,.$$

Then,

$$\begin{aligned}
&\Big\langle \xi^{-1}\frac{\partial}{\partial\xi}\xi^{-1}\big(\boldsymbol{\kappa}+\boldsymbol{\eta}_k\big)\cdot\boldsymbol{\alpha}\,,\,\hat{\psi}\Big\rangle \\
&\qquad = -\frac{1}{2}\mathrm{tr} < \xi^{-1}\frac{\partial}{\partial\xi}\big(\boldsymbol{\kappa}+\boldsymbol{\eta}_k\big)\cdot\xi\boldsymbol{\alpha}\,,\,\hat{\psi}\Big\rangle - \frac{1}{2}\mathrm{tr}\Big\langle \boldsymbol{\alpha}\otimes\big(\boldsymbol{\kappa}+\boldsymbol{\eta}_k\big)\,,\,\hat{\psi}\Big\rangle \\
&\qquad = \Big(\boldsymbol{\alpha}\times\big(\boldsymbol{\kappa}+\boldsymbol{\eta}_k\big)\Big)\cdot\boldsymbol{\Psi} = \Big(\big(\boldsymbol{\kappa}+\boldsymbol{\eta}_k\big)\times\boldsymbol{\Psi}\Big)\cdot\boldsymbol{\alpha}\,.
\end{aligned} \tag{4.70}$$

Similarly,

$$\Big\langle \xi^{-1}\frac{\partial}{\partial\boldsymbol{\kappa}}\xi^{-1}\big(\boldsymbol{\kappa}+\boldsymbol{\eta}_k\big)\cdot\boldsymbol{\alpha}\,,\,\hat{\psi}\Big\rangle = \Big\langle \xi^{-1}\frac{\partial}{\partial\boldsymbol{\kappa}}\big(\boldsymbol{\kappa}+\boldsymbol{\eta}_k\big)\cdot\xi\boldsymbol{\alpha}\,,\,\hat{\psi}\Big\rangle = \boldsymbol{\alpha}\cdot\boldsymbol{\Psi}\,. \tag{4.71}$$

The derivatives of $\xi^{-1}(\boldsymbol{\kappa}+\boldsymbol{\eta}_k)\times\boldsymbol{\eta}_m$ are computed similarly using standard properties of vector cross products. For brevity, we shall not present these calculations here. The final answer for $D_{2,\boldsymbol{\mu}}$ is given by collecting the terms proportional to $\boldsymbol{\mu}$, and coefficient of $D_{2,\boldsymbol{\alpha}}$ is given by the terms proportional to $\boldsymbol{\alpha}$. The operator D_2 is thus given by

$$\begin{aligned}
D_{2,\boldsymbol{\mu}}(\boldsymbol{\Psi},\boldsymbol{\phi}) =& \frac{Q'(d_{km})}{d_{km}} I_2(\Xi,\boldsymbol{\phi},\boldsymbol{\Psi})\big(\boldsymbol{\eta}_m\times\xi^{-1}\mathbf{d}_{km}\big) \\
&+ Q\big(d_{km}\big)\boldsymbol{\eta}_m\times\Big(-\xi^{-1}(\boldsymbol{\kappa}+\boldsymbol{\eta}_k)\times\boldsymbol{\phi}+\boldsymbol{\Psi}\Big)\,, \\
D_{2,\boldsymbol{\alpha}}(\boldsymbol{\Psi},\boldsymbol{\phi}) =& \frac{Q'(d_{km})}{d_{km}} I_2(\Xi,\boldsymbol{\phi},\boldsymbol{\Psi})\xi^{-1}\mathbf{d}_{km} \\
&+ Q\big(d_{km}\big)\Big(-\xi^{-1}(\boldsymbol{\kappa}+\boldsymbol{\eta}_k)\times\boldsymbol{\phi}+\boldsymbol{\Psi}\Big)\,.
\end{aligned} \tag{4.72}$$

From (4.72) one notices a very interesting relationship, namely,

$$D_{2,\boldsymbol{\mu}}(\boldsymbol{\Psi},\boldsymbol{\phi}) = \boldsymbol{\eta}_m\times D_{2,\boldsymbol{\alpha}}(\boldsymbol{\Psi},\boldsymbol{\phi})\,. \tag{4.73}$$

The linearization operator is computed from D_2 as follows (using $\Xi(s,s') = a^{s'-s}$ and $\psi_1(s) = Se^{iks}$):

$$\begin{aligned}
\mathbb{L}\big(\Xi\big)(\boldsymbol{\phi},\boldsymbol{\Psi}) =& -\mathrm{Ad}^*_{\Xi^{-1}}\Big(D_2(\big(\boldsymbol{\phi},\boldsymbol{\Psi}\big) - \mathrm{ad}^*_{(\boldsymbol{\phi},\boldsymbol{\Psi})}D_1(\Xi)\Big)(s,s') \\
&+ \Big(D_2\big(\boldsymbol{\phi},\boldsymbol{\Psi}\big)\Big)(s',s)\,,
\end{aligned} \tag{4.74}$$

where

$$D_1(\Xi) = \Xi^{-1}\frac{\delta U}{\delta \Xi} = \frac{U'(d_{km})}{d_{km}}\Big(- \big(\xi^{-1}\mathbf{d}_{km} \times \boldsymbol{\eta}_m\big)^T \,,\, \big(\xi^{-1}\mathbf{d}_{km}\big)\Big)\,. \tag{4.75}$$

It is also useful to outline the formula for the change of variables $s \leftrightarrow s'$ that forms the last term of the linearization operator $\mathbb{L}$. Under this change, the operators Ad and Ad^* change their form, and it is essential to perform this transformation correctly. Since

$$(\boldsymbol{\phi}, \boldsymbol{\Psi})(s, s') = -\mathrm{Ad}_{\Xi^{-1}(s,s')}\psi_1(s) + \psi_1(s') = -\mathrm{Ad}_{a^{s-s'}}e^{iks}S + e^{iks'}S\,,$$

an exchange of variables $s \leftrightarrow s'$ gives

$$(\boldsymbol{\phi}, \boldsymbol{\Psi})(s', s) = -\mathrm{Ad}_{\Xi^{-1}(s',s)}e^{iks'}\psi_1(s') + \psi_1(s) = -\mathrm{Ad}_{a^{s'-s}}e^{iks'}S + e^{iks}S\,.$$

Thus, the final expression for the linearization of the nonlocal term is

$$\begin{aligned}\mathbb{L}(s)\psi_1 = -\mathrm{Ad}^*_{a^{-s'}}\Big[&D_2\big(a^{s'}\big)\big(- \mathrm{Ad}_{a^{-s'}}S + e^{iks'}S\big)\\ &- \mathrm{ad}^*_{(-\mathrm{Ad}_{a^{-s'}}S + e^{iks'}S)}D_1(\Xi)\Big] + D_2\big(a^{-s'}\big)\big(- e^{iks'}\mathrm{Ad}_{a^{s'}}S + S\big)\,.\end{aligned} \tag{4.76}$$

4.8 Numerical stability of a linear polymer

In order to apply the general method of geometric linear stability of helical polymers derived in the previous section and show how our theory applies to a real-world case, we will solve the problem of linear stability in an example of a naturally straight, untwisted polymer in its unstressed configuration. The reason we choose this polymer is the relative simplicity of the formulas, where all the Ad and Ad^* operators are identities. Also, this is exactly the PVDF polymer considered in Section 4.4, with the linear state being the most basic energy state of the molecule. It is thus interesting

to determine the conditions for the linear state to become unstable, so other helical states computed in that section can be achieved. We shall note that the stability of the helical states computed in Section 4.4 can be considered analogously using the results of the previous section. Here, however, we shall avoid doing this as it will make the chapter unnecessarily complex, due to the large number of helical states we have described. Thus, for the sake of simplicity, we shall only concentrate on the stability of a polymer that is perfectly straight in an unstressed configuration, as illustrated in Fig. 4.7. We also explain in this section the difference between linear stability of a continuous and discrete polymer, and explain why a short enough polymer may exhibit stability whereas a long polymer will be unstable.

We assume the 2-charge bouquet with a charge of $\pm q$ on each end, resulting in a constant dipole moment perpendicular to the axis. The charges interact through a screened electrostatic interaction (4.25) and Lennard-Jones interactions (4.34). The charges are positioned away from the axis at the distance $l_c = 1\text{Å}$, and the distance between the charges is $l_0 = 1\text{Å}$. We take electrostatic charges to be $0.17e$, leading to the dipole moment of $5.33 \times 10^{-30} C \cdot m \simeq 1.63D$ (Debye units), which is slightly smaller than a PVDF polymer having similar polarization structure. Note that this charge is smaller than the value $q = 0.3e$ taken in the computation of the stationary states, since for $q = 0.3$ the subcritical bifurcation occurs for the values of ionic strength of about $I \sim 200\text{M/l}$ which is orders of magnitude larger than any values of I achievable experimentally. Thus, the linear polymer with charges $q = \pm 0.3e$ at the ends of bouquets will be inherently unstable for all viable experimental conditions.

4.8.1 Setup of the problem

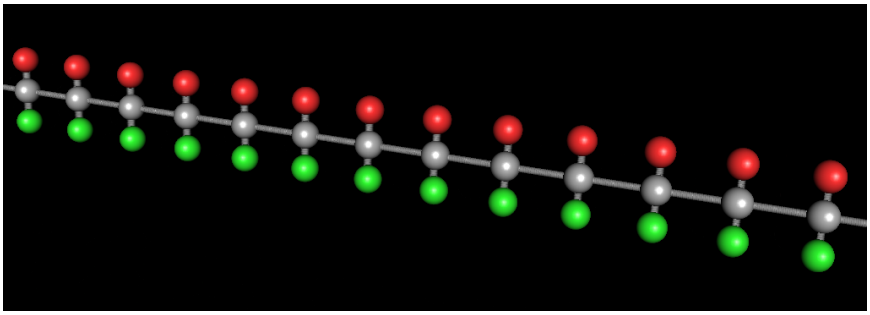

Figure 4.7: A particular example of charged rod with repeated configuration of two charges, plus (top) and minus (bottom) of $q = 0.17e$ that interact through a screened electrostatic and Lennard-Jones potential.

Remark 4.8.1. *It is well known that the stability of* finite *Kirchhoff rods strongly depends on the end conditions imposed on the edges of the rod [34], p.81. The issue of choosing the right boundary conditions is a delicate one and, as far as we know, not entirely understood even for the Kirchhoff rods. Since the focus of this article is the investigation of the effects of nonlocal terms, we shall assume that the boundary conditions at the edges of the rod are such that Fourier transform analysis can be applied.*

For the purpose of this chapter we shall assume the simplest possible shape of the 6×6 elastic tensor J: we take J to be diagonal, with the values of 3 first diagonals $\mu = 3.025 \cdot 10^5 J/(rad^2)Mol = 5.022 \cdot 10^{-19} J/rad^2$being the twist rigidity of a C-C bond, as outlined above, and the values of the last 3 diagonals (stretch rigidity along different directions) being μl^2. Such a choice of the elastic constants achieves $J = \mathrm{Id}_{6\times 6}$ in the dimensionless units. All lengths are then expressed in units of l_0.

We take the inertia tensor to be

$$I_0 = \mathrm{diag}(m_0 l_0^2, m_0 l_0^2, m_0 l_0^2, m_0, m_0, m_0),$$

where m_0 is the mass of the charged atom at the end of the rigid bouquet. In reality, I will be a symmetric positive-definite tensor depending on the exact nature of the polymer selected. Selecting the time scale $\tau = \sqrt{\mu/m_0}$ sets all the coefficients of the temporal and elastic terms exactly equal to unity. It is convenient to choose the unit of electrostatic charge as

$$e_* = \sqrt{4\pi\epsilon_0 \mu l_0} \sim 1.514e\,.$$

The value e_* is chosen in such a way that two charges separated by l_0 interact with potential $\pm\mu$. The dimensionless Lennard-Jones amplitude is $\epsilon/\mu \sim 2.90 \times 10^{-4}$.

Our theory is also applicable for more complex values of elasticity tensors. However, the more complex elastic properties of the rod may themselves lead to instabilities, as earlier works show [28,30]. Thus, we shall assume the simplest possible elastic tensor in order to concentrate on the appearance of instabilities due to the long-range interactions.

Limiting the considerations to unstressed linear polymers provides rather substantial simplifications in the expressions for the nonlocal terms. More precisely, the following simplifications hold:

$$\begin{aligned} a &= \text{Parallel shift along the rod's axis by l,} \\ \mathrm{Ad}_{a^s} &= \text{Identity in } \mathfrak{se}(3), \\ \mathrm{Ad}^*_{a^s} &= \text{Identity in } \mathfrak{se}(3)^*, \\ K_0 &= 0 \quad \text{(no stress in the basic state),} \\ D_1 &= 0 \quad \text{(no twist in the basic state).} \end{aligned} \tag{4.77}$$

Using this information, the dispersion relation $\omega(k)$ can be now directly computed from (4.61). Unfortunately, even though the linearized operator $\mathbb{L}$ is simplified considerably, very little further analytical progress can be made and one has to turn to numerical computations.

In order to compute the frequency $\omega(k)$, for a given k, we need to calculate the linearized operator $\mathbb{L}$. The computation proceeds as follows. First, we identify a basic vector S_i which is a unit vector in six-dimensional space, with 1 at i-th component and 0 otherwise. Then, we compute the matrix

$$M(k) = \big(\mathbf{q}_1, \ldots, \mathbf{q}_6\big)\,, \quad \mathbf{q}_i = \big(\sin^2\frac{k}{2}\big) J S_i + \sum_{s',m,k} \mathbb{L}\big(s'\big) S_i\,. \tag{4.78}$$

The frequencies ω are then computed as generalized eigenvalues of

$$M(k)\mathbf{S} = \omega^2(k) I_0 \mathbf{S}\,. \tag{4.79}$$

In Appendix D, we consider a simplified pedagogical case when the polymer is only allowed to twist, which leads to (almost) analytic expressions for the linear stability. Unfortunately, in our case, the entries of 6x6 matrix $M(k)$ for every wavenumber k have to be computed numerically as outlined above. It is not a difficult or numerically challenging computation though, and the computation of dispersion relation for several hundred values of k only takes a few seconds in *Matlab* on a standard desktop. The results of these computations are presented below.

4.8.2 Results of linear stability computations

For the rod we are considering here, in the absence of the nonlocal interactions the rod is neutrally stable, as the elastic tensor J is diagonal with positive entries. It

is therefore interesting that nonlocal terms introduce instability, corresponding to $\mathrm{Im}(\omega(k)) > 0$. Because $\omega(k)$ only enters as ω^2 in (4.79), the instability occurs when $\lambda = \omega^2(k)$ becomes negative. Physically, this instability is connected to the inclination of the rod to minimize its energy and properly align the dipole moments of each bouquet by twisting, as we have seen in the minimum energy calculation in Sec. 4.4. Mathematically, the instability corresponds to the eigenvalues of the linearization matrix M becoming negative for some k. However, one needs to keep in mind the discrete nature of our rods, making only certain values of k possible. Since we rescale the length by l_0, the distance between the centers of the bouquets, it is natural to take $k \in [0, 2\pi]$. In the continuum case, there is no restriction on the wavenumber, so the condition for the instability is simply

$$\text{Instability} = \min_{0<k<2\pi} \text{eigenvalues } (M) < 0.$$

On the other hand, for a discrete chain of length N, k takes the values $2\pi n/N$, where $0 < n < N$. Thus, for a discrete chain

$$\text{Instability} = \min_{k=2\pi i/N} \text{eigenvalues } (M) < 0.$$

The difference between the discrete and continuous case is illustrated in Figure 4.8. The vertical axis shows the eigenvalue $\lambda = \omega^2(k)$ with $\lambda < 0$ corresponding to the instability. Only the part $0 < k < \pi$ has physical meaning, since dispersion curves are symmetric with respect to reflection about $k = \pi$. There is an instability region for small $k < k_*$. However, a polymer of length $N = 9$ (circles) does not have any allowed wavenumbers in the unstable region, whereas a polymer of length $N = 10$ (blue squares) has one wavenumber $k = \pi/5$ in the unstable region. Thus, from a physical point of view, even for a formally unstable situation, a short enough chain

will be stable. There is some indication that such behavior is indeed observed in VDF polymers [26].

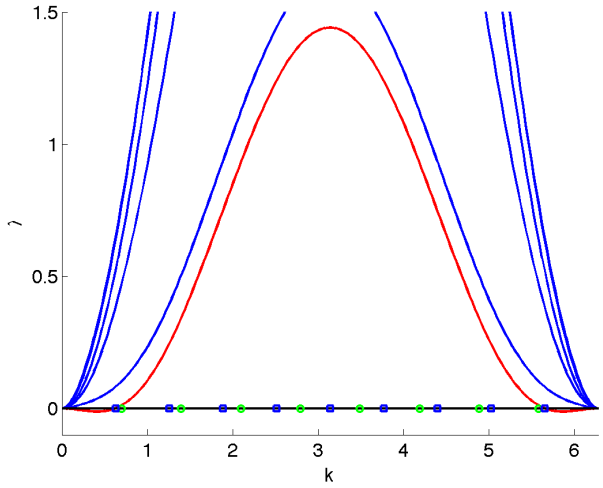

Figure 4.8: The eigenvalue $\lambda(k) = \omega^2(k)$ of (4.79) is shown when the ionic strength of the solution $I = 10^{-2}M/l$. The lowest curve is the unstable dispersion curve. Circles correspond to a chain of length $N = 9$, squares are for $N = 10$.

When the ionic strength I is increased, the Debye screening length decreases according to (4.26), thereby decreasing the electrostatic interaction. It is thus natural to assume that the rod's instability decreases for large values of I until finally the stabilizing elastic forces overcome the nonlocal forces. This is the case here. On the left side of Figure 4.9, we show the maximum unstable wavenumber k_{max} as a function of the ionic strength I, and on the right side of this figure, we show the corresponding maximum stable rod length. We see that stabilization happens at $I \gtrsim 8.9$M/l.

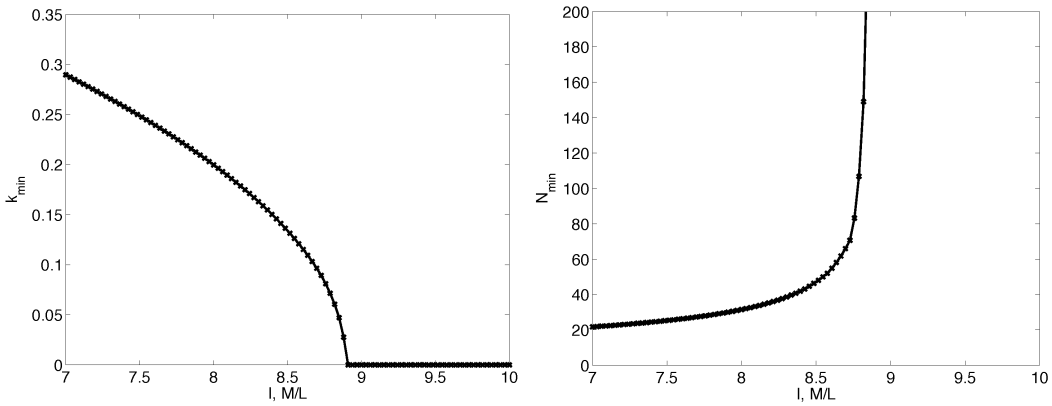

Figure 4.9: The minimum unstable wavenumber (left) and its inverse, the maximum unstable wavelength, (right) are shown as functions of ionic strength in M/l.

4.9 Conclusion

This chapter has investigated the particular example of a simple molecule whose helical configurations possess the complex and intriguing energy landscape shown in Fig. 4.3 and Fig. 4.4. Yellow dots in these figures correspond to energy minima and thus show stationary conformations. We have derived a general scheme for analyzing linear stability of these states, particularly to elucidate the effects of torque on the molecular rod generated by non-local interactions of off-axis charge conformations. The stability analysis was facilitated by the $SE(3)$ symmetry of helical stationary configurations of the rod and it showed that non-local charge-interaction effects could induce instability of helical configurations due to the torques exerted on the rod by off-axis charges. This was illustrated by the instability of a linear polymer in its natural state. We have also shown how the increase of the ionic strength of the solution, in weakening the electrostatic interactions, leads to stabilization of the rod.

REFERENCES

[1] C. Branden and J. Tooze. *Introduction to Protein Structure, 2nd ed.* Garland Publishing, New York, NY, 1999.

[2] P. Burkhard, J. Stetefeld, and S.V. Strelkov. Coiled coils: a highly versatile protein folding motif. *Trends in Cell Biology*, 11(2):82–88, 2001.

[3] H. Li, D.J. DeRosier, W.V. Nicholson, E. Nogales, and K.H. Downing. Microtubule structure at 8 Å resolution. *Structure*, 10:1317–1328, 2002.

[4] X. Yu, S.A. Jacobs, S.C. West, T. Ogawa, and E.H. Egelman. Domain structure and dynamics in the helical filaments formed by RecA and Rad51 on DNA. *Proc. Natl. Acad. Sci.*, 98(15):8419–8424, 2001.

[5] J. J. L. M. Cornelissen, J. J. J. M. Donners, R. de Gelder, W. S. Graswinckel, G. A. Metselaar, A. E. Rowan, N. A. J. M. Sommerdijk, and R. J. M. Nolte. β-helical polymers from isocyanopeptides. *Science*, 293:676–680, 2001.

[6] J. R. Matthews, F. Goldoni, A. P. H. J. Schenning, and E. W. Meijer. Non-ionic polythiophenes: a non-aggregating folded structure in water. *Chem. Comm*, pages 5503–5505, 2005.

[7] N. Chouaieb, A. Goriely, and J. H. Maddocks. Helices. *Proc. Natl. Acad. Sci.*, 103:9398–9403, 2006.

[8] J. R. Banavar, T. X. Hoang, J. H. Maddocks, A. Maritan, C. Poletto, A. Stasiak, and A. Trovato. Structural motifs of macromolecules. *Proc. Natl. Acad. Sci.*, 104:17283–17286, 2007.

[9] E. H. Dill. Kirchhoff's theory of rods. *Arch. Hist. Exact Sci.*, 44:1–23, 1992.

[10] D. J. Dichmann, Y. Li, and J. H. Maddocks. *Hamiltonian Formulation and Symmetries in Rod Mechanics*, volume 82: Mathematical Approaches to Biomolecular Structure and Dynamics. Springer IMA, New York, 1992.

[11] A. Goriely and M.Tabor. New amplitude equations for thin elastic rods. *Phys. Rev. Lett.*, 77:3537–3540, 1996.

[12] R. Goldstein, T. R. Powers, and C. H. Wiggins. Viscous nonlinear dynamics of twist and writhe. *Phys. Rev. Lett.*, 80:5232–5235, 1998.

[13] A. Balaeff, L. Mahadevan, and K. Schulten. Elastic rod model of a DNA loop in the lac operon. *Phys. Rev. Lett*, 83:4900–4903, 1999.

[14] R. Goldstein, A. Goriely, G. Huber, and C. Wolgemuth. Bistable helixes. *Phys. Rev. Lett.*, 84:1631–1634, 2000.

[15] A. Hausrath and A. Goriely. Repeat protien architectures predicted by a continuum representation of fold space. *Protein Science*, 15:1–8, 2006.

[16] S. Neukirch, A. Goriely, and A. C. Hausrath. Chirality of coiled coils: Elasticity matters. *Phys. Rev. Lett.*, 100:038105, 2008.

[17] N. Chouaieb and J. Maddocks. Kirchhoffs problem of helical equilibria of uniform rods. *J. of Elasticity*, 77:221–247, 2004.

[18] S. DeLillo, G. Lupo, and M. Sommacal. Helical configurations of elastic rods in the presence of a long-range interaction potential. *J. Phys. A: Math. Gen*, 43:085214, 2010.

[19] A. Maritan, C. Micheletti, A. Trovato, and J. R. Banavar. Optimal shapes of compact strings. *Nature*, 406:287–290, 2000.

[20] D. D. Holm and V. Putkaradze. Nonlocal orientation-dependent dynamics of charged strands and ribbons. *C. R. Acad. Sci. Paris, Sér. I: Mathematique*, 347:1093–1098, 2009.

[21] D. Ellis, D. D. Holm, F. Gay-Balmaz, V. Putkaradze, and T. Ratiu. Geometric mechanics of flexible strands of charged molecules. *Arch. Rat. Mech. Anal., To appear*, 2009.

[22] J. C. Simó, J. E. Marsden, and P. S. Krishnaprasad. The Hamiltonian structure of nonlinear elasticity: The material and convective representations of solids, rods, and plates. *Arch. Rat. Mech. Anal*, 104:125–183, 1988.

[23] D. D. Holm. *Geometric Mechanics II: Rotation, Translation and Rolling*. Imperial College Press, 2009.

[24] J. Moser and A. P. Veselov. Discrete versions of some classical integrable systems and factorization of matrix polynomials. *Comm. Math. Phys.*, 139:217–243, 1991.

[25] L. Pauling, R. B. Corey, and H. R. Branson. The structure of proteins: Two hydrogen-bonded helical configurations of the polypeptide chain. *Proc. Natl. Acad. Sci*, 37:205–2011, 1951.

[26] K. Noda, K. Ishida, A. Kubono, T. Horiuchi, and H. Yamada. Remanent polarization of evaporated films of vinylidene fluoride oligomers. *J. Appl. Phys*, 93:2866–2870, 2003.

[27] J. R. Quine. Helix parameters and protein structure using quaternions. *Journal of Molecular Structure (Theochem)*, 460:53–66, 1999.

[28] A. Goriely and M. Tabor. Nonlinear dynamics of filaments. iii. instabilities of helical rods. *Proc. Roy. Soc. A*, 453:2583–2601, 1997.

[29] A. Goriely and M. Tabor. The nonlinear dynamics of filaments. *Nonlinear Dynamics*, 21:101–133, 2000.

[30] P. Shipman and A. Goriely. Dynamics of helical strips. *Physical Review E*, 61:4508–4517, 2000.

[31] A. Goriely, M. Nizette, and M. Tabor. On the dynamics of elastic strips. *J. Nonlinear Science*, 11:3–45, 2001.

[32] S. Lafortune, A. Goriely, and M. Tabor. The dynamics of stretchable rods in the inertial case. *Nonlinear Dynamics*, 43:173–195, 2005.

[33] T. C. Bishop, R. Cortez, and O. O. Zhmudsky. Investigation of bend and shear waves in a geometrically exact elastic rod model. *J. Comp. Physics*, 193:642–665, 2004.

[34] N. Chouaieb. *Kirchhoff's problem of helical solutions of uniform rods and their stability properites*. PhD thesis, EPFL, Sec. Math., 2003.

APPENDIX A

PROOF OF LEMMA 3.2.1

We begin with the divergence theorem,

$$\iint_{\Omega_{\mathcal{D}}} \mathbf{F}(\mathbf{p}) \cdot \hat{\mathbf{n}} \, \mathrm{d}A = \iiint_{\mathcal{D}} \left(\frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{F}}{\partial y} + \frac{\partial \mathbf{F}}{\partial z} \right) \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \, .$$

If we choose $\mathbf{F}(\mathbf{p}) = \mathbf{p}/3$, then

$$\iint_{\Omega_{\mathcal{D}}} \frac{\mathbf{p}}{3} \cdot \hat{\mathbf{n}} \, \mathrm{d}A = \iiint_{\mathcal{D}} \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z = V \, .$$

Because the surface is made of the set $\mathcal{F}$ of triangles $f_i = (\mathbf{q}_{i_1}, \mathbf{q}_{i_3}, \mathbf{q}_{i_3})$, with normal vectors $\hat{\mathbf{n}}_i$, which we can write explicitly as

$$\hat{\mathbf{n}}_i = \frac{(\mathbf{q}_{i_2} - \mathbf{q}_{i_1}) \times (\mathbf{q}_{i_3} - \mathbf{q}_{i_1})}{|(\mathbf{q}_{i_2} - \mathbf{q}_{i_1}) \times (\mathbf{q}_{i_3} - \mathbf{q}_{i_1})|} \, ,$$

we can replace the surface integral with a sum of integrals over each triangle,

$$V = \frac{1}{3} \sum_{f_i \in \mathcal{F}} \iint_{f_i} \mathbf{p} \cdot \hat{\mathbf{n}}_i \, dA \, .$$

We parametrize a point $\mathbf{p}$ on triangle f_i using α and β as

$$\mathbf{p} = \mathbf{q}_{i_1} + \alpha \, (\mathbf{q}_{i_3} - \mathbf{q}_{i_1}) + \beta(1 - \alpha) \, (\mathbf{q}_{i_2} - \mathbf{q}_{i_1}) \, ,$$

where the area element is

$$\mathrm{d}A = |(\mathbf{q}_{i_2} - \mathbf{q}_{i_1}) \times (\mathbf{q}_{i_3} - \mathbf{q}_{i_1})| \, \mathrm{d}\alpha \, \mathrm{d}\beta \, ,$$

and integrate over $\alpha \in (0,1)$ and $\beta \in (0, 1-\alpha)$,

$$V = \frac{1}{3} \sum_{f_i \in \mathcal{F}} \int_0^1 \int_0^{1-\alpha} \left(\mathbf{q}_{i_1} \cdot \hat{\mathbf{n}}_i + \alpha \left(\mathbf{q}_{i_3} - \mathbf{q}_{i_1} \right) \cdot \hat{\mathbf{n}}_i + \beta(1-\alpha) \left(\mathbf{q}_{i_2} - \mathbf{q}_{i_1} \right) \cdot \hat{\mathbf{n}}_i \right) \mathrm{d}A \,.$$

Noticing that $(\mathbf{q}_{i_2} - \mathbf{q}_{i_1}) \perp \hat{\mathbf{n}}_i$ and $(\mathbf{q}_{i_2} - \mathbf{q}_{i_1}) \perp \hat{\mathbf{n}}_i$, this simplifies to

$$V = \frac{1}{6} \sum_{f_i \in \mathcal{F}} \left| (\mathbf{q}_{i_2} - \mathbf{q}_{i_1}) \times (\mathbf{q}_{i_3} - \mathbf{q}_{i_1}) \right| \mathbf{q}_{i_1} \cdot \hat{\mathbf{n}}_i \,.$$

APPENDIX B

ALGORITHM FOR MESH ADJUSTMENT

Given a fixed constant ε, a degeneracy threshold $\mu << \varepsilon$, and a maximal angle ϕ_{max} with $0 < \pi - \phi_{max} << \pi$, we consider a triangulated mesh to be "valid" if the following conditions hold:

1. no edge has length exceeding ε,
2. no face has a largest angle exceeding ϕ_{max}, and
3. no edges have length smaller than μ, unless removing that edge would violate condition (2)

After each step in the evolution of the mesh representation of a membrane, the mesh must be adjusted to correct any violations of the above conditions. We modify the procedure in [1] to accomplish this.

B.0.1 Edge length adjustment

The first step is to ensure that no edge in the mesh has length exceeding $(\varepsilon - 2\mu)$. To do this, we simply divide any edge with length exceeds $(\varepsilon - 2\mu)$ at its midpoint with a new vertex point, and create a new edge from this new vertex to the opposite vertex

of the two faces that share the edge being divided, as shown in Figure B.1. Since there are finitely many such edges, and each requires at a finite number of subdivisions to drive edge lengths below ε, this process is guaranteed to terminate.

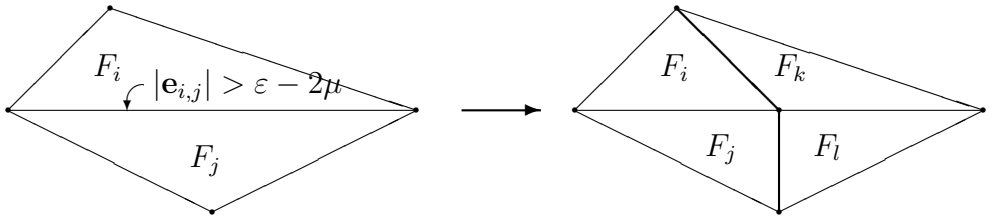

Figure B.1: Subdividing an edge for which $|\mathbf{e}_{i,j}| > \varepsilon$. The edge is divided into two edges, each half the original length. This operation adds one vertex, three edges, and two faces.

B.0.2 Elimination of caps

Following [1], we define a *cap* as a face whose largest angle exceeds θ_{max} but whose smallest edge length is larger than μ (we deal with faces with edge lengths below μ in a later step). θ_{max} is related to ψ_{max} as follows. Given a triangle

For each cap face in the mesh, we define a plane perpendicular to the cap face and to the edge opposite the large angle, passing through the vertex where the large angle occurs, and slice the entire model with this plane. We refer the reader to [1] for details of the method to prevent such an operation from creating new cap faces. The results is the elimination of the cap face without the introduction of any new cap faces. As there are finitely many cap faces in the mesh, this process is guaranteed to terminate. Also, since this process only subdivides existing faces, it cannot introduce an edge that is longer than an existing edge, and so the resulting mesh will have no

edge exceeding $(\varepsilon - \mu)$ in length.

B.0.3 Elimination of microfaces

We define a *microface* as a face with at least two edges of length smaller than μ. We eliminate such faces, when doing so would not create a cap, by replacing the face by a vertex at its center, as shown in Figure B.2. Note that this operation can move a vertex by at most $\varepsilon\sqrt{3}/3$.

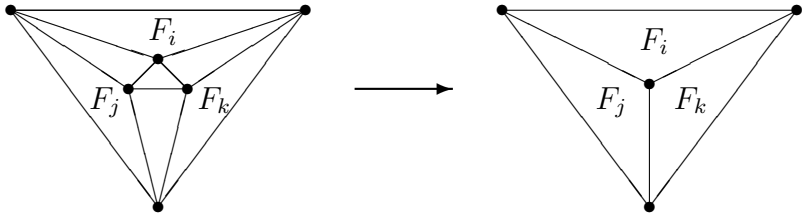

Figure B.2: Removing a microface by replacing it with a single vertex. This operation removes two vertices, six edges, and four faces.

B.0.4 Elimination of needles

We next define a *needle* as a face with a single edge has length smaller than μ. Needles are eliminated, when doing so would not create a cap, by replacing the small edge by a vertex at its midpoint, eliminating the small edge and the two triangles that share that edge, as shown in Figure B.3. This operation can move a vertex by, at most, $\mu/2$.

The removal of microfaces and needles can move each vertex, at most, $\mu\sqrt{3}/3$, and so the length of each edge cannot exceed ε after these operations, leaving the mesh in a valid state.

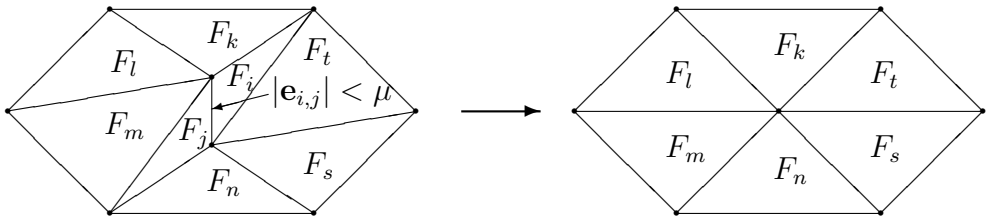

Figure B.3: Merging vertices to remove a needle. The short edge is replaced by a single vertex at its midpoint. This operation removes one vertex, three edges, and two faces.

B.0.5 Maximal movement for subsequent modeling iteration

Having ensured that the mesh is valid, the maximum movement that a vertex can be allowed to undergo during the subsequent time step in the model evolution is given by half the length of the shortest edge or altitude associated with that vertex, where we compute the altitude d_i of a face relative to vertex $\mathbf{p}_i$ from the face area A_i using the edge $\mathbf{e}_{j,k}$ opposite the vertex in the face, by

$$d_i = \frac{2A_i}{|\mathbf{e}_{j,k}|} \, .$$

This constraint ensures that faces will not become degenerate during model evolution, and that the adjustment process described here will be able to restore the mesh to a valid state after the vertex positions are updated.

APPENDIX C

GEOMETRIC PROPERTIES OF $SE(3)$ GROUP

This Appendix describes the geometric structure of $SE(3)$ group and defines the aspects of its adjoint and co-adjoint actions that are needed for the computations in the text. For more details and the theoretical framework, see [2].

Suppose G is a Lie group, and g and h are elements of G. Then, the AD operator – the conjugacy class of h – is defined as

$$\mathrm{AD}_g h = ghg^{-1}\,, \tag{C.1}$$

for all $g \in G$. Assume that $h(t)$ changes smoothly with respect to a parameter t starting at the unit element of the group. Then, $h(0) = e$ and $h'(0) = \eta$ is the velocity at the initial point, taken to be unity. Note that η is an element of the tangent space at unity which is the Lie algebra of G. We denote this fact as $T_eG \simeq \mathfrak{g}$. In this notation, the Adjoint operation is defined as

$$\mathrm{Ad}_g \eta = \frac{d}{dt} gh(t)g^{-1}\bigg|_{t=0} = g\eta g^{-1}\,. \tag{C.2}$$

Note that Ad takes an element in $\eta \in \mathfrak{g}$ and produces another element in $\mathfrak{g}$. Now, suppose $g(\epsilon)$ is also varying with respect to a parameter ϵ, and again $g(0) = e$,

$g'(0) = \xi \in \mathfrak{g}$. In this notation, the *adjoint* operator ad is defined as

$$\mathrm{ad}_\xi \eta = \frac{d}{d\epsilon}\mathrm{Ad}_g(\epsilon)\eta\bigg|_{\epsilon=0} = \xi\eta - \eta\xi =: \left[\xi\,,\eta\right], \tag{C.3}$$

where $[\cdot\,,\cdot]$ is the commutator in the Lie algebra $\mathfrak{g}$.

In order to derive equations, it is important to consider the co-adjoint operators Ad^* and ad^*. The operation $\mathrm{Ad}^* : G \times \mathfrak{g}^* \to \mathfrak{g}^*$, is defined for $g \in G$ and $a \in \mathfrak{g}^*$ as

$$\left\langle \eta\,,\, \mathrm{Ad}^*_g a\right\rangle = \left\langle \mathrm{Ad}_g \eta\,,\, a\right\rangle \tag{C.4}$$

for every $\eta \in \mathfrak{g}$, in terms of a suitable pairing $\langle\,\cdot\,,\,\cdot\,\rangle : \mathfrak{g}^* \times \mathfrak{g} \to \mathbb{R}$. Similarly, the operation $\mathrm{ad}^* : \mathfrak{g} \times \mathfrak{g}^* \to \mathfrak{g}^*$, is defined for $\eta, \xi \in \mathfrak{g}$ and $a \in \mathfrak{g}^*$ as

$$\left\langle \eta\,,\, \mathrm{ad}^*_\xi a\right\rangle = \left\langle \mathrm{ad}_\xi \eta\,,\, a\right\rangle. \tag{C.5}$$

Let us now see how these formulas are expressed for the $SE(3)$ group. The Lie group multiplication of two elements $(\Lambda_1, \mathbf{r}_1) \in SE(3)$ and $(\Lambda_2, \mathbf{r}_2) \in SE(3)$, where $\Lambda_1, \Lambda_2 \in SO(3)$ and $\mathbf{r}_1, \mathbf{r}_2 \in \mathbb{R}^3$, is defined as follows:

$$\big(\Lambda_1, \mathbf{r}_1\big) \cdot \big(\Lambda_2, \mathbf{r}_2\big) = \big(\Lambda_1\Lambda_2, \Lambda_1\mathbf{r}_2 + \mathbf{r}_1\big) \tag{C.6}$$

with the meaning of subsequent application of rotation and shift. The inverse element is then

$$\big(\Lambda, \mathbf{r}\big)^{-1} = \big(\Lambda^{-1}, -\Lambda^{-1}\mathbf{r}\big). \tag{C.7}$$

The tangent space $T_{(\Lambda_0,\mathbf{r}_0)}SE(3)$ at a point $(\Lambda_0, \mathbf{r}_0)$ is defined as the space of derivatives of curves $\big(\Lambda(t), \mathbf{r}(t)\big)$ at $t = 0$ given that $\Lambda(0) = \Lambda_0$, $\mathbf{r}(0) = \mathbf{r}_0$. In order to obtain the element of the tangent space at the identity – that is, the Lie algebra $T_eSE(3) \simeq \mathfrak{se}(3)$ – we compute the derivative at $\Lambda_0 = \mathrm{Id}_{3\times 3}$ (a 3×3 unity matrix), $\mathbf{r}_0 = \mathbf{0}$. Hence, an

element of this Lie algebra can be written as

$$\eta = \big(\hat{\omega}\,,\,\mathbf{v}\big) = \big(\boldsymbol{\omega}\,,\,\mathbf{v}\big) \in \mathfrak{se}(3)\,,$$

where $\hat{\omega}$ denotes a skew-symmetric 3×3 matrix and $\mathbf{v}$ is a vector in three dimensions. Here, we may use the so-called "hat map" correspondence between the skew-symmetric matrices and vectors to define a vector $\boldsymbol{\omega} \in \mathbb{R}^3$ as $\hat{\omega}_{ij} = -\epsilon_{ijk}\omega_k$, such that $\hat{\omega}\mathbf{r} = \boldsymbol{\omega} \times \mathbf{r}$ for all $\mathbf{r}$. Thus, $\mathfrak{se}(3)$ is a six-dimensional vector space, with the first three components having the physical meaning of the angular velocity, and the last three components being the linear velocity.

Using this preliminary information, we are ready to define the adjoint actions. After some relatively straightforward computations, we have:

$$\mathrm{AD}_{(\Lambda,\mathbf{r})}\big(\tilde{\Lambda}\,,\,\tilde{\mathbf{r}}\big) = (\Lambda\tilde{\Lambda}\Lambda^{-1}, -\Lambda\tilde{\Lambda}\Lambda^{-1}\mathbf{r} + \Lambda\tilde{\mathbf{r}} + \mathbf{r})\,. \tag{C.8}$$

Then if $(\hat{\omega}, \mathbf{v}) = \frac{\mathrm{d}}{\mathrm{d}t}(\tilde{\Lambda}(t), \tilde{\mathbf{r}}(t))|_{t=0}$,

$$\mathrm{Ad}_{(\Lambda,\mathbf{r})}\big(\hat{\omega}\,,\,\mathbf{v}\big) = \big(\Lambda\hat{\omega}\Lambda^{-1}, -\Lambda\hat{\omega}\Lambda^{-1}\mathbf{r} + \Lambda\mathbf{v}\big)\,, \tag{C.9}$$

and using $\hat{\omega}\Lambda^{-1}\mathbf{r} = \boldsymbol{\omega} \times \Lambda^{-1}\mathbf{r} = \Lambda^{-1}\,(\Lambda\boldsymbol{\omega} \times \mathbf{r})$,

$$\mathrm{Ad}_{(\Lambda,\mathbf{r})}\big(\hat{\omega}\,,\,\mathbf{v}\big) = \big(\Lambda\hat{\omega}\Lambda^{-1}\,,\,-\Lambda\boldsymbol{\omega} \times \mathbf{r} + \Lambda\mathbf{v}\big)\,. \tag{C.10}$$

To express this in vector form, we note that for arbitrary $\mathbf{u}$, $\Lambda\hat{\omega}\Lambda^{-1}\mathbf{u} = \Lambda(\boldsymbol{\omega}\times\Lambda^{-1}\mathbf{u}) = \Lambda\boldsymbol{\omega} \times \mathbf{u} = \widehat{(\Lambda\boldsymbol{\omega})}\,\mathbf{u}$, one has

$$\mathrm{Ad}_{(\Lambda,\mathbf{r})}\big(\boldsymbol{\omega}\,,\,\mathbf{v}\big) = \big(\Lambda\boldsymbol{\omega}\,,\,-\Lambda\boldsymbol{\omega} \times \mathbf{r} + \Lambda\mathbf{v}\big)\,. \tag{C.11}$$

Letting $(\hat{\omega}_1, \boldsymbol{\alpha}_1) = \frac{\mathrm{d}}{\mathrm{d}\epsilon}(\Lambda(\epsilon), \mathbf{r}(\epsilon))|_{\epsilon=0}$, one finds

$$\mathrm{ad}_{(\hat{\omega}_1,\boldsymbol{\alpha}_1)}(\hat{\omega}_2\,,\,\boldsymbol{\alpha}_2) = (\hat{\omega}_1\hat{\omega}_2 - \hat{\omega}_2\hat{\omega}_1, -\hat{\omega}_2\boldsymbol{\alpha}_1 + \hat{\omega}_1\boldsymbol{\alpha}_2)\,, \tag{C.12}$$

then because $[\hat{\omega}_1, \hat{\omega}_2]\mathbf{u} = (\boldsymbol{\omega}_1 \times \boldsymbol{\omega}_2) \times \mathbf{u}$ for all $\mathbf{u}$, we can express this in vector form as

$$\mathrm{ad}_{(\boldsymbol{\omega}_1, \boldsymbol{\alpha}_1)}(\boldsymbol{\omega}_2\,,\, \boldsymbol{\alpha}_2) = \big(\boldsymbol{\omega}_1 \times \boldsymbol{\omega}_2\,,\, \boldsymbol{\omega}_1 \times \boldsymbol{\alpha}_2 - \boldsymbol{\omega}_2 \times \boldsymbol{\alpha}_1\big)\,. \tag{C.13}$$

The physically relevant pairing between two elements $(\boldsymbol{\omega}, \boldsymbol{\alpha}) \in \mathfrak{se}(3)$ and $(\mathbf{u}, \mathbf{a}) \in \mathfrak{se}(3)^*$ is given by

$$\big\langle (\boldsymbol{\omega}, \boldsymbol{\alpha})\,,\, (\mathbf{u}, \mathbf{a}) \big\rangle = \boldsymbol{\omega} \cdot \mathbf{u} + \boldsymbol{\alpha} \cdot \mathbf{a}\,. \tag{C.14}$$

With this choice of pairing, we may also write $\mathfrak{se}(3)$ as a pair of two 3D vectors. In that notation, the co-Adjoint operator is

$$\mathrm{Ad}^*_{(\Lambda, \mathbf{r})^{-1}}(\mathbf{u}, \mathbf{a}) = \big(\Lambda\mathbf{u} + \mathbf{r} \times \Lambda\mathbf{a}\,,\, \Lambda\mathbf{a}\big)\,, \tag{C.15}$$

and the co-adjoint action is given by

$$\mathrm{ad}^*_{(\boldsymbol{\omega}, \boldsymbol{\alpha})}(\mathbf{u}, \mathbf{a}) = \big(\mathbf{u} \times \boldsymbol{\omega} - \boldsymbol{\alpha} \times \mathbf{a}\,,\, -\boldsymbol{\omega} \times \mathbf{a}\big)\,. \tag{C.16}$$

APPENDIX D

TWIST DYNAMICS OF A STRAIGHT POLYMER

In this section, we consider the linear polymer drawn in Fig. 4.7 with the restriction that the charge bouquets can only twist about the axis, and only in the plane perpendicular to the axis. The rod itself is assumed to be completely rigid. This problem – in a slightly different configuration – was considered in [3] as a model of DNA dynamics. For convenience, we take the mass of the charges to be $m_0/2$ so the moment of inertia has the value $m_0 l_0^2$ (no factor of 2), to coincide with the formulas derived in Sec. 4.8. We show two ways to analyze the linear stability of this problem.

Standard solution

The configuration space for this problem is described by the angles of rotation ϕ_i. The coordinate for the positive and negative charges are given by

$$\mathbf{r}_{k,\pm} = \big(kl, \pm l_0 \cos\phi_k, \pm l_0 \sin\phi_k\big).$$

The state $\phi_i = 0$ for all i is an equilibrium state of the system. In order to linearize around that state, we proceed as follows.

The distance between a charge at m-th unit and a charge at n-th unit depends on whether the charges are at the same or opposite sides of the chain. For the same

side we get

$$d_{mn}^{+} = \sqrt{l_0^2 (m-n)^2 + l^2 (\cos\phi_m - \cos\phi_n)^2 + l^2 (\sin\phi_m - \sin\phi_n)^2}$$
$$\simeq \sqrt{l_0^2 (m-n)^2 + l^2 \frac{1}{4} (\phi_m^2 - \phi_n^2)^2 + l^2 (\phi_m - \phi_n)^2 + O(\phi^4)}.$$

For charges on the opposite sides of the chain,

$$d_{mn}^{-} = \sqrt{l_0^2 (m-n)^2 + l^2 (\cos\phi_m + \cos\phi_n)^2 + l^2 (\sin\phi_m + \sin\phi_n)^2}$$
$$\simeq \sqrt{l_0^2 (m-n)^2 + l^2 \left(2 - \frac{1}{2}\phi_n^2 - \frac{1}{2}\phi_n^2\right)^2 + l^2 (\phi_m + \phi_n)^2 + O(\phi^4)}.$$

The electrostatic energy is positive for the charges on the same side, and negative for the charges on the opposite side, whereas Lennard-Jones energy only depends on the distance between the charges. Thus, the total potential energy $\mathbb{P}$ is given by the sum

$$\mathbb{P} = -\frac{1}{2}\sum_m J\phi_m^2 + \frac{1}{2}\sum m, n, \pm U\left(d_{mn}^{+}\right) + U\left(d_{mn}^{-}\right). \tag{D.1}$$

The linearized equation of state is given by

$$I\frac{d^2\phi_m}{dt^2} = -\frac{\partial \mathbb{P}}{\partial \phi_m}, \tag{D.2}$$

where $I = m_0 l_0^2$ is the moment of inertia, m_0 being the mass of the charged particle. We choose the length to be measured in terms of l, and wavenumber $0 < k < 2\pi$ will be dimensionless. Assuming $\phi_m = e^{ikm - i\omega t}S$, after some fairly simple algebra we obtain the following dispersion relation:

$$I\omega^2(k) = J\sin^2\frac{kl_0}{2} - \sum_{m=-\infty, m\neq 0}^{\infty}\left[\frac{U'\left(|m|\right)}{|m|} + \frac{U'\left(\sqrt{m^2+4}\right)}{\sqrt{m^2+4}}\right]\left(1 - e^{ikm}\right).$$

On summing up terms with opposite signs of m, one finds the real expression

$$I\omega^2(k) = J\sin^2\frac{kl_0}{2} - \sum_{m=1}^{\infty}\left[\frac{U'\left(|m|\right)}{|m|} + \frac{U'\left(\sqrt{m^2+4}\right)}{\sqrt{m^2+4}}\right]4\sin^2\frac{mk}{2}. \tag{D.3}$$

For the chosen values of physical parameters, $\omega^2 > 0$ and the twist dynamics is stable.

Geometric method

Assume that the axis of the rod is along x-axis, and all the charges in the undisturbed configuration are aligned along the z-axis. The deformations are rotations about the x-axis, which are given by the first coordinate in $\mathfrak{se}(3)$ representation. The resulting rotation is also about the x-axis, so we need to look at the first row of the matrix M in (4.78). Thus, the stability of the problem of the twist about the axis is considered as simply the $(1,1)$ component of (4.79):

$$I_{11}\omega^2 = \big(MS_1\big)_1\,, \tag{D.4}$$

where we take $S_1 = (1,0,0,0,0,0)^T$ to be the vector of infinitesimal rotations about the x-axis. Again, after some rather straightforward algebra (not presented here) we see that the right-hand side of (D.4) gives exactly (D.3). As verification of our code, we also compared the numerical results given by these two methods; they were identical within numerical accuracy of the computations.

We note that (D.3) *does not* correspond to any eigenvalues, since the vector MS_1 is full, *i.e.* it has non-trivial components at other entries besides the first. These components arise because a realistic twist deformation about the x-axis induces twists and stretches in other directions. Nevertheless, we feel that such a simplified physical example still provides a useful verification procedure for the full stability calculation.

APPENDIX E

JAVA REFERENCE IMPLEMENTATION OF SOFTWARE

The reference software library, which is also available on the author's web site, is included here. It is divided into sections as follows

- Utility Code
  - Buffer Management
  - 2D and 3D Points and Vectors
  - Logging
  - Pooled Object Management
  - High performance sparse arrays
  - Color Management
  - Font Management
  - XML File Management
  - User Interface Utilities
  - General utilities
- Media Management

  - QuickTime Movie Creation

- 3-D Visualization
  - 3-D Rendering Pipeline

- Mathematics
  - Math Utilities
  - Graphing
  - Linear Algebra
  - Solvers
  - Optimization
  - Delaunay Triangulation

- Filter Tree Management and Video Microscopy Analysis
  - Filter Tree Infrastructure
  - Video Microscopy Analysis

- Mathematical Modeling
  - Grids for Neighbor Finding
  - Triangulated Meshes
  - Models of Cells and Cell Behavior

As the author of the source code available by herein, I hereby place this source code in the public domain. You can use, modify, and distribute the source code

and executable programs based on the source code. However, note the following:

DISCLAIMEROFWARRANTYThissourcecodeisprovided"asis"andwithoutanyexpressorimp

E.1 Utility Code

E.1.1 Buffer Management (com.srbenoit.buffer)

This package provides for pooled buffers. Management of pools is performed by the com.srbenoit.pool package, but this package provides the specific pooled objects to manage buffers of bytes and characters. These buffers can be chained to support arbitrarily large blocks of data, and are intended to allow programs to manage large and variable sized units of data without frequent allocation operations.

```
package com.srbenoit.buffer;

import java.nio.Buffer;
import java.nio.InvalidMarkException;
import com.srbenoit.pool.AbstractPoolObject;

/**
 * A base class with common methods for buffers that can be stored in a <code>Pool</code>. Each
 * buffer contains a pointer to a subsequent buffer ( <code>null</code> for the last buffer in a
 * chain), allowing buffers to be assembled into a linked list.
 */
public abstract class AbstractPooledBuffer extends AbstractPoolObject {

    /** the backing buffer */
    private transient Buffer buf;

    /** flag indicating that this is the last buffer in a unit of data */
    private transient boolean terminal;

    /**
     * Constructs a new <code>AbstractPooledBuffer</code>.
     *
     * @param  buffer  the backing buffer
     */
    public AbstractPooledBuffer(final Buffer buffer) {

        super();

        this.buf = buffer;
        this.terminal = false;
    }

    /**
     * Gets the buffer.
     *
     * @return  the buffer
     */
    public Buffer getBuffer() {

        return this.buf;
    }

    /**
     * Replaces the current buffer with a new buffer.
     *
     * @param  newBuf  the new buffer
     */
```

```
protected void setBuffer(final Buffer newBuf) {

    this.buf = newBuf;
}

/**
 * Allocates a new buffer of a specified capacity.
 *
 * @param  capacity  the capacity of the new buffer
 */
public abstract void newBuffer(final int capacity);

/**
 * Sets the flag that indicates that this buffer is the last buffer in a unit of data. This
 * could be used, for example, to tell a server to close a channel once this data has been
 * successfully sent.
 *
 * @param  isTerminal  <code>true</code> if this is the last buffer in the unit of data; <code>
 *                     false</code> otherwise
 */
public void setTerminal(final boolean isTerminal) {

    synchronized (this.synch) {
        this.terminal = isTerminal;
    }
}

/**
 * Gets the flag that indicates that this buffer is the last buffer in a unit of data.
 *
 * @return  <code>true</code> if this is the last buffer in the unit of data; <code>
 *          false</code> otherwise
 */
public boolean isTerminal() {

    synchronized (this.synch) {
        return this.terminal;
    }
}

/**
 * Resets this buffer to a virgin state (used before the buffer is returned to a pool).
 */
@Override public void toVirginState() {

    synchronized (this.synch) {
        this.buf.clear();
        this.terminal = false;
    }
}

/**
 * Informs the pool object that it is being released for garbage collection and should free any
 * internal resources. The object may not be reused after this method is called.
 */
@Override public void die() {

    toVirginState();
}

/**
 * Clears the backing byte buffer. The position is set to zero, the limit is set to the
 * capacity, and the mark is discarded.
 */
public void clear() {

    synchronized (this.synch) {
        this.buf.clear();
    }
}

/**
 * Flips the backing buffer. The limit is set to the current position and then the position is
 * set to zero. If the mark is defined then it is discarded.
 */
public void flip() {

    synchronized (this.synch) {
        this.buf.flip();
    }
}

/**
 * Rewinds the backing buffer. The position is set to zero and the mark is discarded.
 */
public void rewind() {
```

```
        synchronized (this.synch) {
            this.buf.rewind();
        }
    }

    /**
     * Sets the backing buffer's mark at its position.
     */
    public void mark() {

        synchronized (this.synch) {
            this.buf.mark();
        }
    }

    /**
     * Resets this buffer's position to the previously-marked position.
     *
     * <p>Invoking this method neither changes nor discards the mark's value.
     *
     * @throws  InvalidMarkException  if the mark has not been set
     */
    public void reset() throws InvalidMarkException {

        synchronized (this.synch) {
            this.buf.reset();
        }
    }

    /**
     * Returns the backing buffer's position.
     *
     * @return  the position
     */
    public int position() {

        synchronized (this.synch) {
            return this.buf.position();
        }
    }

    /**
     * Sets the backing buffer's position. If the mark is defined and larger than the new position
     * then it is discarded.
     *
     * @param    offset  the new position value; must be non-negative and no larger than the current
     *                   limit
     * @throws  IllegalArgumentException  if the preconditions on <code>offset</code> do not hold
     */
    public void position(final int offset) throws IllegalArgumentException {

        synchronized (this.synch) {
            this.buf.position(offset);
        }
    }

    /**
     * Returns the number of elements between the current position and the limit.
     *
     * @return  the number of elements remaining
     */
    public int remaining() {

        synchronized (this.synch) {
            return this.buf.remaining();
        }
    }

    /**
     * Returns the backing buffer's capacity.
     *
     * @return  the capacity
     */
    public int capacity() {

        synchronized (this.synch) {
            return this.buf.capacity();
        }
    }

    /**
     * Returns the backing buffer's limit.
     *
     * @return  the limit
     */
    public int limit() {
```

```
        synchronized (this.synch) {
            return this.buf.limit();
        }
    }
}

package com.srbenoit.buffer;

import java.io.IOException;
import java.nio.BufferOverflowException;
import java.nio.BufferUnderflowException;
import java.nio.ByteBuffer;
import java.nio.ReadOnlyBufferException;
import java.nio.channels.NotYetConnectedException;
import java.nio.channels.SocketChannel;
import com.srbenoit.pool.AbstractPoolObject;

/**
 * A buffer that holds byte data and that can be stored in a <code>Pool</code>. Each buffer
 * contains a pointer to a subsequent buffer (null for the last buffer in a chain), allowing
 * buffers to be assembled into a linked list.
 */
public class PooledByteBuffer extends AbstractPooledBuffer {

    /** the next buffer in a linked list */
    private transient PooledByteBuffer next;

    /**
     * Constructs a new <code>PooledByteBuffer</code>.
     *
     * @param  capacity  the capacity of a buffer, in bytes
     */
    public PooledByteBuffer(final int capacity) {

        super(ByteBuffer.allocate(capacity));

        this.next = null;
    }

    /**
     * Gets the buffer.
     *
     * @return  the buffer
     */
    @Override public ByteBuffer getBuffer() {

        return (ByteBuffer) super.getBuffer();
    }

    /**
     * Allocates a new buffer of a specified capacity;
     *
     * @param  capacity  the capacity of the new buffer
     */
    @Override public void newBuffer(final int capacity) {

        setBuffer(ByteBuffer.allocate(capacity));
    }

    /**
     * Sets the next buffer in the linked list.
     *
     * @param  nextBuffer  the next buffer
     */
    public void setNext(final PooledByteBuffer nextBuffer) {

        synchronized (this.synch) {
            this.next = nextBuffer;
        }
    }

    /**
     * Gets the next buffer in the linked list.
     *
     * @return  the next buffer
     */
    public PooledByteBuffer getNext() {

        synchronized (this.synch) {
            return this.next;
        }
    }

    /**
     * Reads the byte at the backing buffer's current position, and then increments that position.
     *
     * @return  the byte at the backing buffer's current position
```

```
     * @throws  BufferUnderflowException  if the backing buffer's current position is not smaller
     *                                    than its limit
     */
    public byte get() throws BufferUnderflowException {

        synchronized (this.synch) {
            return getBuffer().get();
        }
    }

    /**
     * Writes the given byte into the backing buffer at the current position, and then increments
     * that position.
     *
     * @param   data  the byte to be written
     * @throws  BufferOverflowException  if the backing buffer's current position is not smaller
     *                                   than its limit
     * @throws  ReadOnlyBufferException  if the backing buffer is read-only
     */
    public void put(final byte data) throws BufferOverflowException, ReadOnlyBufferException {

        synchronized (this.synch) {
            getBuffer().put(data);
        }
    }

    /**
     * Reads the byte from the backing buffer at the given index.
     *
     * @param   index  the index from which the byte will be read
     * @return  the byte at the given index
     * @throws  IndexOutOfBoundsException  if <code>index</code> is negative or not smaller than
     *                                     the backing buffer's limit
     */
    public byte get(final int index) {

        synchronized (this.synch) {
            return getBuffer().get(index);
        }
    }

    /**
     * Writes the given byte into the backing buffer at the given index.
     *
     * @param   index  the index at which the byte will be written
     * @param   data   the byte value to be written
     * @throws  IndexOutOfBoundsException  if <code>index</code> is negative or not smaller than
     *                                     the buffer's limit
     * @throws  ReadOnlyBufferException    if this buffer is read-only
     */
    public void put(final int index, final byte data) throws IndexOutOfBoundsException,
        ReadOnlyBufferException {

        synchronized (this.synch) {
            getBuffer().put(index, data);
        }
    }

    /**
     * Transfers bytes from the backing buffer into the given destination array. If there are fewer
     * bytes remaining in the buffer than are required to satisfy the request, that is, if <code>
     * length &gt; remaining()</code>, then no bytes are transferred and a <code>
     * BufferUnderflowException</code> is thrown.
     *
     * <p>Otherwise, this method copies <code>length</code> bytes from this buffer into the given
     * array, starting at the current position of this buffer and at the given offset in the array.
     * The position of this buffer is then incremented by <code>length</code>.
     *
     * @param   dst     the array into which bytes are to be written
     * @param   offset  the offset within the array of the first byte to be written; must be
     *                  non-negative and no larger than <code>dst.length</code>
     * @param   length  the maximum number of bytes to be written to the given array; must be
     *                  non-negative and no larger than <code>dst.length - offset</code>
     * @throws  BufferUnderflowException   if there are fewer than <code>length</code> bytes
     *                                     remaining in the backing buffer
     * @throws  IndexOutOfBoundsException  if the preconditions on the <code>offset</code> and
     *                                     <code>length</code> parameters do not hold
     */
    public void get(final byte[] dst, final int offset, final int length) {

        synchronized (this.synch) {
            getBuffer().get(dst, offset, length);
        }
    }

    /**
     * Transfers bytes from the backing buffer into the given destination array.
```

```
     *
     * @param   dst  the array into which bytes are to be written
     * @throws  BufferUnderflowException  if there are fewer than <code>length</code> bytes
     *                                    remaining in the backing buffer
     */
    public void get(final byte[] dst) {

        synchronized (this.synch) {
            getBuffer().get(dst);
        }
    }

    /**
     * Transfers bytes into the backing buffer from the given source array. If there are more bytes
     * to be copied from the array than remain in this buffer, that is, if <code>length &gt;
     * remaining()</code>, then no bytes are transferred and a <code>BufferOverflowException</code>
     * is thrown.
     *
     * <p>Otherwise, this method copies <code>length</code> bytes from the given array into the
     * backing buffer, starting at the given offset in the array and at the current position of the
     * backing buffer. The position of this buffer is then incremented by <code>length</code>.
     *
     * @param   src      the array from which bytes are to be read
     * @param   offset   the offset within the array of the first byte to be read; must be
     *                   non-negative and no larger than <code>array.length</code>
     * @param   length   the number of bytes to be read from the given array; must be non-negative
     *                   and no larger than <code>array.length - offset</code>
     * @throws  BufferOverflowException     if there is insufficient space in this buffer
     * @throws  IndexOutOfBoundsException   if the preconditions on the <code>offset</code> and
     *                                      <code>length</code> parameters do not hold
     * @throws  ReadOnlyBufferException     if this buffer is read-only
     */
    public void put(final byte[] src, final int offset, final int length)
        throws BufferOverflowException, IndexOutOfBoundsException, ReadOnlyBufferException {

        synchronized (this.synch) {
            getBuffer().put(src, offset, length);
        }
    }

    /**
     * This method transfers the entire content of the given source byte array into the backing
     * buffer.
     *
     * @param   src  the array from which bytes are to be read
     * @throws  BufferOverflowException  if there is insufficient space in the backing buffer
     * @throws  ReadOnlyBufferException  if this buffer is read-only
     */
    public void put(final byte[] src) {

        synchronized (this.synch) {
            getBuffer().put(src);
        }
    }

    /**
     * Reads data into the byte buffer from a socket channel.
     *
     * @param   channel  the channel from which to read
     * @return  the number of bytes read, possibly zero, or -1 if the channel has reached
     *          end-of-stream
     * @throws  IOException                if there was an I/O error while reading data
     * @throws  NotYetConnectedException   if the channel is not yet connected
     */
    public int readFrom(final SocketChannel channel) throws IOException, NotYetConnectedException {

        synchronized (this.synch) {
            return channel.read(getBuffer());
        }
    }

    /**
     * Writes data from the byte buffer to a socket channel.
     *
     * @param   channel  the channel from which to read
     * @return  the number of bytes written, possibly zero
     * @throws  IOException                if there was an I/O error while reading data
     * @throws  NotYetConnectedException   if the channel is not yet connected
     */
    public int writeTo(final SocketChannel channel) throws IOException, NotYetConnectedException {

        synchronized (this.synch) {
            return channel.write(getBuffer());
        }
    }

    /**
```

```
     * Creates a copy of the object, but with an independent backing buffer - used to create new
     * objects when the pool is empty. This is more efficient than creating a new object.
     *
     * @return  the copy
     */
    @Override public AbstractPoolObject copy() {

        PooledByteBuffer copy;

        synchronized (this.synch) {

            try {
                copy = (PooledByteBuffer) super.clone();
                copy.newBuffer(capacity());
            } catch (CloneNotSupportedException e) {
                copy = new PooledByteBuffer(capacity());
            }
        }

        return copy;
    }

    /**
     * Resets this buffer to a virgin state (used before the buffer is returned to a pool).
     */
    @Override public void toVirginState() {

        synchronized (this.synch) {
            this.next = null;
            super.toVirginState();
        }
    }

    /**
     * Informs the pool object that it is being released for garbage collection and should free any
     * internal resources. The object may not be reused after this method is called.
     */
    @Override public void die() {

        synchronized (this.synch) {
            this.next = null;
            super.die();
        }
    }
}

package com.srbenoit.buffer;

import java.nio.BufferOverflowException;
import java.nio.BufferUnderflowException;
import java.nio.CharBuffer;
import java.nio.ReadOnlyBufferException;
import com.srbenoit.pool.AbstractPoolObject;

/**
 * A buffer that holds character data and that can be stored in a <code>Pool</code>. Each buffer
 * contains a pointer to a subsequent buffer (null for the last buffer in a chain), allowing
 * buffers to be assembled into a linked list.
 */
public class PooledCharBuffer extends AbstractPooledBuffer {

    /** the next buffer in a linked list */
    private transient PooledCharBuffer next;

    /**
     * Constructs a new <code>PooledCharBuffer</code>.
     *
     * @param  capacity  the capacity of a buffer, in characters
     */
    public PooledCharBuffer(final int capacity) {

        super(CharBuffer.allocate(capacity));
        this.next = null;
    }

    /**
     * Gets the buffer.
     *
     * @return  the buffer
     */
    @Override public CharBuffer getBuffer() {

        return (CharBuffer) super.getBuffer();
    }

    /**
     * Allocates a new buffer of a specified capacity;
```

```
     *
     * @param  capacity  the capacity of the new buffer
     */
    @Override public void newBuffer(final int capacity) {

        setBuffer(CharBuffer.allocate(capacity));
    }

    /**
     * Sets the next buffer in the linked list.
     *
     * @param  nextBuffer  the next buffer
     */
    public void setNext(final PooledCharBuffer nextBuffer) {

        synchronized (this.synch) {
            this.next = nextBuffer;
        }
    }

    /**
     * Gets the next buffer in the linked list.
     *
     * @return  the next buffer
     */
    public PooledCharBuffer getNext() {

        synchronized (this.synch) {
            return this.next;
        }
    }

    /**
     * Reads the character at the backing buffer's current position, and then increments that
     * position.
     *
     * @return  the character at the backing buffer's current position
     * @throws  BufferUnderflowException  if the backing buffer's current position is not smaller
     *                                    than its limit
     */
    public char get() throws BufferUnderflowException {

        synchronized (this.synch) {
            return getBuffer().get();
        }
    }

    /**
     * Writes the given character into the backing buffer at the current position, and then
     * increments that position.
     *
     * @param   data  the character to be written
     * @throws  BufferOverflowException  if the backing buffer's current position is not smaller
     *                                   than its limit
     * @throws  ReadOnlyBufferException  if the backing buffer is read-only
     */
    public void put(final char data) throws BufferOverflowException, ReadOnlyBufferException {

        synchronized (this.synch) {
            getBuffer().put(data);
        }
    }

    /**
     * Reads the character from the backing buffer at the given index.
     *
     * @param   index  the index from which the character will be read
     * @return  the character at the given index
     * @throws  IndexOutOfBoundsException  if <code>index</code> is negative or not smaller than
     *                                     the backing buffer's limit
     */
    public char get(final int index) {

        synchronized (this.synch) {
            return getBuffer().get(index);
        }
    }

    /**
     * Writes the given character into the backing buffer at the given index.
     *
     * @param   index  the index at which the character will be written
     * @param   data   the character value to be written
     * @throws  IndexOutOfBoundsException  if <code>index</code> is negative or not smaller than
     *                                     the buffer's limit
     * @throws  ReadOnlyBufferException    if this buffer is read-only
     */
```

```
public void put(final int index, final char data) throws IndexOutOfBoundsException,
    ReadOnlyBufferException {

    synchronized (this.synch) {
        getBuffer().put(index, data);
    }
}

/**
 * Transfers characters from the backing buffer into the given destination array. If there are
 * fewer characters remaining in the buffer than are required to satisfy the request, that is,
 * if <code>length &gt; remaining()</code>, then no characters are transferred and a <code>
 * BufferUnderflowException</code> is thrown.
 *
 * <p>Otherwise, this method copies <code>length</code> characters from this buffer into the
 * given array, starting at the current position of this buffer and at the given offset in the
 * array. The position of this buffer is then incremented by <code>length</code>.
 *
 * @param   dst      the array into which characters are to be written
 * @param   offset   the offset within the array of the first byte to be written; must be
 *                   non-negative and no larger than <code>dst.length</code>
 * @param   length   the maximum number of characters to be written to the given array; must be
 *                   non-negative and no larger than <code>dst.length - offset</code>
 * @throws  BufferUnderflowException    if there are fewer than <code>length</code> characters
 *                                      remaining in the backing buffer
 * @throws  IndexOutOfBoundsException   if the preconditions on the <code>offset</code> and
 *                                      <code>length</code> parameters do not hold
 */
public void get(final char[] dst, final int offset, final int length) {

    synchronized (this.synch) {
        getBuffer().get(dst, offset, length);
    }
}

/**
 * Transfers characters from the backing buffer into the given destination array.
 *
 * @param   dst  the array into which characters are to be written
 * @throws  BufferUnderflowException  if there are fewer than <code>length</code> characters
 *                                    remaining in the backing buffer
 */
public void get(final char[] dst) {

    synchronized (this.synch) {
        getBuffer().get(dst);
    }
}

/**
 * Transfers characters into the backing buffer from the given source array. If there are more
 * characters to be copied from the array than remain in this buffer, that is, if <code>length
 * &gt; remaining()</code>, then no bytes are transferred and a <code>
 * BufferOverflowException</code> is thrown.
 *
 * <p>Otherwise, this method copies <code>length</code> characters from the given array into
 * the backing buffer, starting at the given offset in the array and at the current position of
 * the backing buffer. The position of this buffer is then incremented by <code>length</code>.
 *
 * @param   src      the array from which characters are to be read
 * @param   offset   the offset within the array of the first character to be read; must be
 *                   non-negative and no larger than <code>array.length</code>
 * @param   length   the number of characters to be read from the given array; must be
 *                   non-negative and no larger than <code>array.length - offset</code>
 * @throws  BufferOverflowException     if there is insufficient space in this buffer
 * @throws  IndexOutOfBoundsException   if the preconditions on the <code>offset</code> and
 *                                      <code>length</code> parameters do not hold
 * @throws  ReadOnlyBufferException     if this buffer is read-only
 */
public void put(final char[] src, final int offset, final int length)
    throws BufferOverflowException, IndexOutOfBoundsException, ReadOnlyBufferException {

    synchronized (this.synch) {
        getBuffer().put(src, offset, length);
    }
}

/**
 * This method transfers the entire content of the given source character array into the
 * backing buffer.
 *
 * @param   src  the array from which characters are to be read
 * @throws  BufferOverflowException  if there is insufficient space in the backing buffer
 * @throws  ReadOnlyBufferException  if this buffer is read-only
 */
public void put(final char[] src) {
```

```
            synchronized (this.synch) {
                getBuffer().put(src);
            }
        }

        /**
         * Creates a copy of the object, but with an independent backing buffer - used to create new
         * objects when the pool is empty. This is more efficient than creating a new object.
         *
         * @return  the copy
         */
        @Override public AbstractPoolObject copy() {

            PooledCharBuffer copy;

            synchronized (this.synch) {

                try {
                    copy = (PooledCharBuffer) super.clone();
                    copy.newBuffer(capacity());
                } catch (CloneNotSupportedException e) {
                    copy = new PooledCharBuffer(capacity());
                }
            }

            return copy;
        }

        /**
         * Resets this buffer to a virgin state (used before the buffer is returned to a pool).
         */
        @Override public void toVirginState() {

            synchronized (this.synch) {
                this.next = null;
                super.toVirginState();
            }
        }

        /**
         * Informs the pool object that it is being released for garbage collection and should free any
         * internal resources. The object may not be reused after this method is called.
         */
        @Override public void die() {

            synchronized (this.synch) {
                this.next = null;
                super.die();
            }
        }
}
```

E.1.2 2D and 3D Points and Vectors (com.srbenoit.geom)

This package provides interfaces and classes to represent 2-D and 3-D points and vectors, as well as sparse array classes to manage lists of these points and vecors, and affine transformation classes to transform them efficiently. This package also provides a based vector class, representing a base point and associated vector that are treated as a unit.

```
package com.srbenoit.geom;

import com.srbenoit.log.LoggedObject;

/**
 * A simple two-dimensional based vector, combining a base point with a vector.
 */
public class BasedVector2 extends LoggedObject implements Point2Int, Vector2Int {
```

```
/** the X coordinate of the base point */
private double posX;

/** the Y coordinate of the base point */
private double posY;

/** the X component of the vector */
private double vecX;

/** the Y component of the vector */
private double vecY;

/** the length of the vector, computed lazily */
private double len;

/**
 * Constructs a <code>BasedVector2</code> with base point at the origin and null vector.
 */
public BasedVector2() {

    this.posX = 0;
    this.posY = 0;
    this.vecX = 0;
    this.vecY = 0;
    this.len = 0;
}

/**
 * Constructs a <code>BasedVector2</code> with specified base point coordinates and vector
 * components.
 *
 * @param  baseX  the X coordinate of the base point
 * @param  baseY  the Y coordinate of the base point
 * @param  vecX   the X component of the vector
 * @param  vecY   the Y component of the vector
 */
public BasedVector2(final double baseX, final double baseY, final double vecX,
    final double vecY) {

    this.posX = baseX;
    this.posY = baseY;
    this.vecX = vecX;
    this.vecY = vecY;
    this.len = -1;
}

/**
 * Constructs a <code>BasedVector2</code> with specified base point and vector.
 *
 * @param  base  the point whose coordinates are to be copied into the base point
 * @param  vec   the vector whose components are to be copied into the vector
 */
public BasedVector2(final Point2Int base, final Vector2Int vec) {

    this.posX = base.getPosX();
    this.posY = base.getPosY();
    this.vecX = vec.getVecX();
    this.vecY = vec.getVecY();
    this.len = vec.lazyLength();
}

/**
 * Gets the X coordinate of the base point.
 *
 * @return  the X coordinate
 */
public double getPosX() {

    return this.posX;
}

/**
 * Sets the X coordinate of the base point.
 *
 * @param  xCoord  the X coordinate
 */
public void setPosX(final double xCoord) {

    this.posX = xCoord;
}

/**
 * Gets the Y coordinate of the base point.
 *
 * @return  the Y coordinate
 */
public double getPosY() {
```

```
    return this.posY;
}

/**
 * Sets the Y coordinate of the base point.
 *
 * @param  yCoord  the Y coordinate
 */
public void setPosY(final double yCoord) {

    this.posY = yCoord;
}

/**
 * Sets the coordinates of the base point.
 *
 * @param  xCoord  the x coordinate
 * @param  yCoord  the y coordinate
 */
public void setPos(final double xCoord, final double yCoord) {

    this.posX = xCoord;
    this.posY = yCoord;
}

/**
 * Sets the coordinates of the base point from another point.
 *
 * @param  source  the point whose position is to be copied
 */
public void setPos(final Point2Int source) {

    this.posX = source.getPosX();
    this.posY = source.getPosY();
}

/**
 * Moves the coordinates of the base point.
 *
 * @param  xDelta  the change to the x coordinate
 * @param  yDelta  the change to the y coordinate
 */
public void move(final double xDelta, final double yDelta) {

    this.posX += xDelta;
    this.posY += yDelta;
}

/**
 * Moves the coordinates of the base point by a vector.
 *
 * @param  vector  the vector by which to move the point
 */
public void move(final Vector2Int vector) {

    this.posX += vector.getVecX();
    this.posY += vector.getVecY();
}

/**
 * Adds a scaled version of a vector to this base point position (base = base + scale tuple).
 *
 * @param  scale   the scalar value
 * @param  vector  the vector to be scaled then added
 */
public void moveScaled(final double scale, final Vector2Int vector) {

    this.posX += vector.getVecX() * scale;
    this.posY += vector.getVecY() * scale;
}

/**
 * Computes the square of the Euclidean distance between the base point and another point.
 *
 * @param   otherPoint  the other point
 * @return  the square of the distance
 */
public double distSquared(final Point2Int otherPoint) {

    double distX;
    double distY;

    distX = this.posX - otherPoint.getPosX();
    distY = this.posY - otherPoint.getPosY();

    return (distX * distX) + (distY * distY);
```

```
}

/**
 * Computes the Euclidean distance between the base point and another point.
 *
 * @param    otherPoint  the other point
 * @return   the distance
 */
public double dist(final Point2Int otherPoint) {

    double distX;
    double distY;

    distX = this.posX - otherPoint.getPosX();
    distY = this.posY - otherPoint.getPosY();

    return Math.sqrt((distX * distX) + (distY * distY));
}

/**
 * Gets the X component of the vector.
 *
 * @return  the X component
 */
public double getVecX() {

    return this.vecX;
}

/**
 * Sets the X component of the vector.
 *
 * @param  xComp  the X component
 */
public void setVecX(final double xComp) {

    this.vecX = xComp;
    this.len = -1;
}

/**
 * Gets the Y component of the vector.
 *
 * @return  the Y component
 */
public double getVecY() {

    return this.vecY;
}

/**
 * Sets the Y component of the vector.
 *
 * @param  yComp  the Y component
 */
public void setVecY(final double yComp) {

    this.vecY = yComp;
    this.len = -1;
}

/**
 * Sets the coordinates of the vector.
 *
 * @param  xComp  the x component
 * @param  yComp  the y component
 */
public void setVec(final double xComp, final double yComp) {

    this.vecX = xComp;
    this.vecY = yComp;
    this.len = -1;
}

/**
 * Sets the coordinates of the vector from another vector.
 *
 * @param  source  the vector whose components are to be copied
 */
public void setVec(final Vector2Int source) {

    this.vecX = source.getVecX();
    this.vecY = source.getVecY();
    this.len = source.lazyLength();
}

/**
```

```
     * Adds to the components of the vector.
     *
     * @param  xDelta  the change to the x component
     * @param  yDelta  the change to the y component
     */
    public void addVec(final double xDelta, final double yDelta) {

        this.vecX += xDelta;
        this.vecY += yDelta;
        this.len = -1;
    }

    /**
     * Adds a vector to this vector.
     *
     * @param  vector  the vector to add to this vector
     */
    public void addVec(final Vector2Int vector) {

        this.vecX += vector.getVecX();
        this.vecY += vector.getVecY();
        this.len = -1;
    }

    /**
     * Subtracts a vector from this vector.
     *
     * @param  vector  the vector to subtract from this vector
     */
    public void subVec(final Vector2Int vector) {

        this.vecX -= vector.getVecX();
        this.vecY -= vector.getVecY();
        this.len = -1;
    }

    /**
     * Subtracts <code>vector2</code> from <code>vector1</code> and stores the result in this
     * vector.
     *
     * @param  vector1  the vector from which to subtract
     * @param  vector2  the vector to subtract
     */
    public void subVec(final Vector2Int vector1, final Vector2Int vector2) {

        this.vecX = vector2.getVecX() - vector1.getVecX();
        this.vecY = vector2.getVecY() - vector1.getVecY();
        this.len = -1;
    }

    /**
     * Adds a scaled version of a vector to this vector (this = this + scale tuple).
     *
     * @param  scale   the scalar value
     * @param  vector  the vector to be scaled then added
     */
    public void addVecScaled(final double scale, final Vector2Int vector) {

        this.vecX += vector.getVecX() * scale;
        this.vecY += vector.getVecY() * scale;
        this.len = -1;
    }

    /**
     * Sets this vector to the vector from <code>point1</code> to <code>point2</code> (this =
     * point2 - point1).
     *
     * @param  point1  the first point
     * @param  point2  the second point
     */
    public void vectorBetween(final Point2Int point1, final Point2Int point2) {

        this.vecX = point2.getPosX() - point1.getPosX();
        this.vecY = point2.getPosY() - point1.getPosY();
        this.len = -1;
    }

    /**
     * Negates this vector in place.
     */
    public void negateVec() {

        // Length is not affected
        this.vecX = -this.vecX;
        this.vecY = -this.vecY;
    }
```

```
/**
 * Scales this vector by a scalar factor.
 *
 * @param  scale  the scalar factor
 */
public void scaleVec(final double scale) {

    this.vecX *= scale;
    this.vecY *= scale;

    // if a length is known, the length is scaled by the absolute value of scale
    if (this.len != -1) {

        if (scale < 0) {
            this.len *= -scale;
        } else {
            this.len *= scale;
        }
    }
}

/**
 * Sets this vector to a scaled version of another vector.
 *
 * @param  scale  the scalar factor
 * @param  vec    the vector to scale
 */
public void scaleVec(double scale, Vector2Int vec) {

    this.vecX = vec.getVecX() * scale;
    this.vecY = vec.getVecY() * scale;

    // if a length is known, the length is scaled by the absolute value of scale
    this.len = vec.lazyLength();

    if (vec.lazyLength() != -1) {

        if (scale < 0) {
            this.len *= -scale;
        } else {
            this.len *= scale;
        }
    }
}

/**
 * Gets the lazily computed length of the vector.
 *
 * @return  the length of the vector, or -1 if the length has not yet been computed
 */
public double lazyLength() {

    return this.len;
}

/**
 * Gets the squared length of the vector.
 *
 * @return  the squared length of the vector
 */
public double lengthSquared() {

    double result;

    if (this.len == -1) {
        result = (this.vecX * this.vecX) + (this.vecY * this.vecY);
    } else {
        result = this.len * this.len;
    }

    return result;
}

/**
 * Gets the length of the vector.
 *
 * @return  the length of the vector
 */
public double length() {

    if (this.len == -1) {
        this.len = Math.sqrt((this.vecX * this.vecX) + (this.vecY * this.vecY));
    }

    return this.len;
}
```

```
    /**
     * Computes the dot product of this vector with another vector.
     *
     * @param   vector  the other vector
     * @return  the dot product
     */
    public double dot(final Vector2Int vector) {

        return (this.vecX * vector.getVecX()) + (this.vecY * vector.getVecY());
    }

    /**
     * Normalizes this vector in place. The null vector is normalized to (1,0).
     */
    public void normalize() {

        double before;

        before = length();

        if (before < Double.MIN_NORMAL) {

            // Don't want to divide by that, so consider it zero.
            this.vecX = 1;
            this.vecY = 0;
        } else {
            this.vecX /= before;
            this.vecY /= before;
        }

        this.len = 1;
    }

    /**
     * Generates the string representation of the point.
     *
     * @return  the <code>String</code> representation
     */
    @Override public String toString() {

        StringBuilder str;

        str = new StringBuilder(100);

        str.append("Base: (");
        str.append(this.posX);
        str.append(", ");
        str.append(this.posY);
        str.append(") Vector: [");
        str.append(this.vecX);
        str.append(", ");
        str.append(this.vecY);
        str.append(']');

        return str.toString();
    }
}

package com.srbenoit.geom;

import com.srbenoit.log.LoggedObject;

/**
 * A simple three-dimensional based vector, combining a base point with a vector.
 */
public class BasedVector3 extends LoggedObject implements Point3Int, Vector3Int {

    /** the X coordinate of the base point */
    private double posX;

    /** the Y coordinate of the base point */
    private double posY;

    /** the Z coordinate of the base point */
    private double posZ;

    /** the X component of the vector */
    private double vecX;

    /** the Y component of the vector */
    private double vecY;

    /** the Z component of the vector */
    private double vecZ;

    /** the length of the vector, computed lazily */
    private double len;
```

```
/**
 * Constructs a <code>BasedVector3</code> with base point at the origin and null vector.
 */
public BasedVector3() {

    this.posX = 0;
    this.posY = 0;
    this.posZ = 0;
    this.vecX = 0;
    this.vecY = 0;
    this.vecZ = 0;
    this.len = 0;
}

/**
 * Constructs a <code>BasedVector3</code> with specified base point coordinates and vector
 * components.
 *
 * @param  baseX  the X coordinate of the base point
 * @param  baseY  the Y coordinate of the base point
 * @param  baseZ  the Z coordinate of the base point
 * @param  vecX   the X component of the vector
 * @param  vecY   the Y component of the vector
 * @param  vecZ   the Z component of the vector
 */
public BasedVector3(final double baseX, final double baseY, final double baseZ,
    final double vecX, final double vecY, final double vecZ) {

    this.posX = baseX;
    this.posY = baseY;
    this.posZ = baseZ;
    this.vecX = vecX;
    this.vecY = vecY;
    this.vecZ = vecZ;
    this.len = -1;
}

/**
 * Constructs a <code>BasedVector3</code> with specified base point and vector.
 *
 * @param  base  the point whose coordinates are to be copied into the base point
 * @param  vec   the vector whose components are to be copied into the vector
 */
public BasedVector3(final Point3Int base, final Vector3Int vec) {

    this.posX = base.getPosX();
    this.posY = base.getPosY();
    this.posZ = base.getPosZ();
    this.vecX = vec.getVecX();
    this.vecY = vec.getVecY();
    this.vecZ = vec.getVecZ();
    this.len = vec.lazyLength();
}

/**
 * Gets the X coordinate of the base point.
 *
 * @return  the X coordinate
 */
public double getPosX() {

    return this.posX;
}

/**
 * Sets the X coordinate of the base point.
 *
 * @param  xCoord  the X coordinate
 */
public void setPosX(final double xCoord) {

    this.posX = xCoord;
}

/**
 * Gets the Y coordinate of the base point.
 *
 * @return  the Y coordinate
 */
public double getPosY() {

    return this.posY;
}

/**
 * Sets the Y coordinate of the base point.
```

```
     *
     * @param  yCoord  the Y coordinate
     */
    public void setPosY(final double yCoord) {

        this.posY = yCoord;
    }

    /**
     * Gets the Z coordinate of the base point.
     *
     * @return  the Z coordinate
     */
    public double getPosZ() {

        return this.posZ;
    }

    /**
     * Sets the Z coordinate of the base point.
     *
     * @param  zCoord  the Z coordinate
     */
    public void setPosZ(final double zCoord) {

        this.posZ = zCoord;
    }

    /**
     * Sets the coordinates of the base point.
     *
     * @param  xCoord  the x coordinate
     * @param  yCoord  the y coordinate
     * @param  zCoord  the z coordinate
     */
    public void setPos(final double xCoord, final double yCoord, final double zCoord) {

        this.posX = xCoord;
        this.posY = yCoord;
        this.posZ = zCoord;
    }

    /**
     * Sets the coordinates of the base point from another point.
     *
     * @param  source  the point whose position is to be copied
     */
    public void setPos(final Point3Int source) {

        this.posX = source.getPosX();
        this.posY = source.getPosY();
        this.posZ = source.getPosZ();
    }

    /**
     * Moves the coordinates of the base point.
     *
     * @param  xDelta  the change to the x coordinate
     * @param  yDelta  the change to the y coordinate
     * @param  zDelta  the change to the z coordinate
     */
    public void move(final double xDelta, final double yDelta, final double zDelta) {

        this.posX += xDelta;
        this.posY += yDelta;
        this.posZ += yDelta;
    }

    /**
     * Moves the coordinates of the base point by a vector.
     *
     * @param  vector  the vector by which to move the point
     */
    public void move(final Vector3Int vector) {

        this.posX += vector.getVecX();
        this.posY += vector.getVecY();
        this.posZ += vector.getVecZ();
    }

    /**
     * Adds a scaled version of a vector to this base point position (base = base + scale tuple).
     *
     * @param  scale   the scalar value
     * @param  vector  the vector to be scaled then added
     */
    public void moveScaled(final double scale, final Vector3Int vector) {
```

```
        this.posX += vector.getVecX() * scale;
        this.posY += vector.getVecY() * scale;
        this.posZ += vector.getVecZ() * scale;
    }

    /**
     * Computes the square of the Euclidean distance between the base point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the square of the distance
     */
    public double distSquared(final Point3Int otherPoint) {

        double distX;
        double distY;
        double distZ;

        distX = this.posX - otherPoint.getPosX();
        distY = this.posY - otherPoint.getPosY();
        distZ = this.posZ - otherPoint.getPosZ();

        return (distX * distX) + (distY * distY) + (distZ * distZ);
    }

    /**
     * Computes the Euclidean distance between the base point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the distance
     */
    public double dist(final Point3Int otherPoint) {

        double distX;
        double distY;
        double distZ;

        distX = this.posX - otherPoint.getPosX();
        distY = this.posY - otherPoint.getPosY();
        distZ = this.posZ - otherPoint.getPosZ();

        return Math.sqrt((distX * distX) + (distY * distY) + (distZ * distZ));
    }

    /**
     * Gets the X component of the vector.
     *
     * @return  the X component
     */
    public double getVecX() {

        return this.vecX;
    }

    /**
     * Sets the X component of the vector.
     *
     * @param  xComp  the X component
     */
    public void setVecX(final double xComp) {

        this.vecX = xComp;
        this.len = -1;
    }

    /**
     * Gets the Y component of the vector.
     *
     * @return  the Y component
     */
    public double getVecY() {

        return this.vecY;
    }

    /**
     * Sets the Y component of the vector.
     *
     * @param  yComp  the Y component
     */
    public void setVecY(final double yComp) {

        this.vecY = yComp;
        this.len = -1;
    }

    /**
```

```
     * Gets the Z component of the vector.
     *
     * @return  the Z component
     */
    public double getVecZ() {

        return this.vecZ;
    }

    /**
     * Sets the Z component of the vector.
     *
     * @param  zComp  the Z component
     */
    public void setVecZ(final double zComp) {

        this.vecZ = zComp;
        this.len = -1;
    }

    /**
     * Sets the coordinates of the vector.
     *
     * @param  xComp  the x component
     * @param  yComp  the y component
     * @param  zComp  the z component
     */
    public void setVec(final double xComp, final double yComp, final double zComp) {

        this.vecX = xComp;
        this.vecY = yComp;
        this.vecZ = zComp;
        this.len = -1;
    }

    /**
     * Sets the coordinates of the vector from another vector.
     *
     * @param  source  the vector whose components are to be copied
     */
    public void setVec(final Vector3Int source) {

        this.vecX = source.getVecX();
        this.vecY = source.getVecY();
        this.vecZ = source.getVecZ();
        this.len = source.lazyLength();
    }

    /**
     * Adds to the components of the vector.
     *
     * @param  xDelta  the change to the x component
     * @param  yDelta  the change to the y component
     * @param  zDelta  the change to the z component
     */
    public void addVec(final double xDelta, final double yDelta, final double zDelta) {

        this.vecX += xDelta;
        this.vecY += yDelta;
        this.vecZ += zDelta;
        this.len = -1;
    }

    /**
     * Adds a vector to this vector.
     *
     * @param  vector  the vector to add to this vector
     */
    public void addVec(final Vector3Int vector) {

        this.vecX += vector.getVecX();
        this.vecY += vector.getVecY();
        this.vecZ += vector.getVecZ();
        this.len = -1;
    }

    /**
     * Subtracts a vector from this vector.
     *
     * @param  vector  the vector to subtract from this vector
     */
    public void subVec(final Vector3Int vector) {

        this.vecX -= vector.getVecX();
        this.vecY -= vector.getVecY();
        this.vecZ -= vector.getVecZ();
        this.len = -1;
```

```
}

/**
 * Subtracts <code>vector2</code> from <code>vector1</code> and stores the result in this
 * vector.
 *
 * @param  vector1  the vector from which to subtract
 * @param  vector2  the vector to subtract
 */
public void subVec(final Vector3Int vector1, final Vector3Int vector2) {

    this.vecX = vector2.getVecX() - vector1.getVecX();
    this.vecY = vector2.getVecY() - vector1.getVecY();
    this.vecZ = vector2.getVecZ() - vector1.getVecZ();
    this.len = -1;
}

/**
 * Adds a scaled version of a vector to this vector (this = this + scale tuple).
 *
 * @param  scale   the scalar value
 * @param  vector  the vector to be scaled then added
 */
public void addVecScaled(final double scale, final Vector3Int vector) {

    this.vecX += vector.getVecX() * scale;
    this.vecY += vector.getVecY() * scale;
    this.vecZ += vector.getVecZ() * scale;
    this.len = -1;
}

/**
 * Sets this vector to the vector from <code>point1</code> to <code>point2</code> (this =
 * point2 - point1).
 *
 * @param  point1  the first point
 * @param  point2  the second point
 */
public void vectorBetween(final Point3Int point1, final Point3Int point2) {

    this.vecX = point2.getPosX() - point1.getPosX();
    this.vecY = point2.getPosY() - point1.getPosY();
    this.vecZ = point2.getPosZ() - point1.getPosZ();
    this.len = -1;
}

/**
 * Negates this vector in place.
 */
public void negateVec() {

    // Length is not affected
    this.vecX = -this.vecX;
    this.vecY = -this.vecY;
    this.vecZ = -this.vecZ;
}

/**
 * Scales this vector by a scalar factor.
 *
 * @param  scale  the scalar factor
 */
public void scaleVec(final double scale) {

    this.vecX *= scale;
    this.vecY *= scale;
    this.vecZ *= scale;

    // if a length is known, the length is scaled by the absolute value of scale
    if (this.len != -1) {

        if (scale < 0) {
            this.len *= -scale;
        } else {
            this.len *= scale;
        }
    }
}

/**
 * Sets this vector to a scaled version of another vector.
 *
 * @param  scale  the scalar factor
 * @param  vec    the vector to scale
 */
public void scaleVec(double scale, Vector3Int vec) {
```

```
        this.vecX = vec.getVecX() * scale;
        this.vecY = vec.getVecY() * scale;
        this.vecZ = vec.getVecZ() * scale;

        // if a length is known, the length is scaled by the absolute value of scale
        this.len = vec.lazyLength();

        if (vec.lazyLength() != -1) {

            if (scale < 0) {
                this.len *= -scale;
            } else {
                this.len *= scale;
            }
        }
    }

    /**
     * Gets the lazily computed length of the vector.
     *
     * @return  the length of the vector, or -1 if the length has not yet been computed
     */
    public double lazyLength() {

        return this.len;
    }

    /**
     * Gets the squared length of the vector.
     *
     * @return  the squared length of the vector
     */
    public double lengthSquared() {

        double result;

        if (this.len == -1) {
            result = (this.vecX * this.vecX) + (this.vecY * this.vecY) + (this.vecZ * this.vecZ);
        } else {
            result = this.len * this.len;
        }

        return result;
    }

    /**
     * Gets the length of the vector.
     *
     * @return  the length of the vector
     */
    public double length() {

        if (this.len == -1) {
            this.len = Math.sqrt((this.vecX * this.vecX) + (this.vecY * this.vecY)
                    + (this.vecZ * this.vecZ));
        }

        return this.len;
    }

    /**
     * Computes the dot product of this vector with another vector.
     *
     * @param   vector  the other vector
     * @return  the dot product
     */
    public double dot(final Vector3Int vector) {

        return (this.vecX * vector.getVecX()) + (this.vecY * vector.getVecY())
            + (this.vecZ * vector.getVecZ());
    }

    /**
     * Computes the cross product of <code>first</code> and <code>second</code> and stores the
     * result in this vector.
     *
     * @param  first   the first tuple in the cross product
     * @param  second  the second tuple in the cross product
     */
    public void cross(final Vector3Int first, final Vector3Int second) {

        this.vecZ = (first.getVecY() * second.getVecZ()) - (first.getVecZ() * second.getVecY());
        this.vecY = (first.getVecZ() * second.getVecX()) - (first.getVecX() * second.getVecZ());
        this.vecZ = (first.getVecX() * second.getVecY()) - (first.getVecY() * second.getVecX());
        this.len = -1;
    }
```

```
    /**
     * Normalizes this vector in place. The null vector is normalized to (1,0).
     */
    public void normalize() {

        double before;

        before = length();

        if (before < Double.MIN_NORMAL) {

            // Don't want to divide by that, so consider it zero.
            this.vecX = 1;
            this.vecY = 0;
            this.vecZ = 0;
        } else {
            this.vecX /= before;
            this.vecY /= before;
            this.vecZ /= before;
        }

        this.len = 1;
    }

    /**
     * Generates the string representation of the point.
     *
     * @return  the <code>String</code> representation
     */
    @Override public String toString() {

        StringBuilder str;

        str = new StringBuilder(140);

        str.append("Base: (");
        str.append(this.posX);
        str.append(", ");
        str.append(this.posY);
        str.append(", ");
        str.append(this.posZ);
        str.append(") Vector: [");
        str.append(this.vecX);
        str.append(", ");
        str.append(this.vecY);
        str.append(", ");
        str.append(this.vecZ);
        str.append(']');

        return str.toString();
    }
}

package com.srbenoit.geom;

import com.srbenoit.log.LoggedObject;

/**
 * A simple two-dimensional point.
 */
public class Point2 extends LoggedObject implements Point2Int {

    /** the X coordinate */
    private double posX;

    /** the Y coordinate */
    private double posY;

    /**
     * Constructs a point at the origin.
     */
    public Point2() {

        this.posX = 0;
        this.posY = 0;
    }

    /**
     * Constructs a point.
     *
     * @param  xCoord  the X coordinate
     * @param  yCoord  the Y coordinate
     */
    public Point2(final double xCoord, final double yCoord) {

        this.posX = xCoord;
        this.posY = yCoord;
```

```
}

/**
 * Constructs a point.
 *
 * @param  source  the point whose coordinates are to be copied
 */
public Point2(final Point2Int source) {

    this.posX = source.getPosX();
    this.posY = source.getPosY();
}

/**
 * Gets the X coordinate.
 *
 * @return  the X coordinate
 */
public double getPosX() {

    return this.posX;
}

/**
 * Sets the X coordinate.
 *
 * @param  xCoord  the X coordinate
 */
public void setPosX(final double xCoord) {

    this.posX = xCoord;
}

/**
 * Gets the Y coordinate.
 *
 * @return  the Y coordinate
 */
public double getPosY() {

    return this.posY;
}

/**
 * Sets the Y coordinate.
 *
 * @param  yCoord  the Y coordinate
 */
public void setPosY(final double yCoord) {

    this.posY = yCoord;
}

/**
 * Sets the coordinates of the point.
 *
 * @param  xCoord  the x coordinate
 * @param  yCoord  the y coordinate
 */
public void setPos(final double xCoord, final double yCoord) {

    this.posX = xCoord;
    this.posY = yCoord;
}

/**
 * Sets the coordinates of the point from another point.
 *
 * @param  source  the point whose position is to be copied
 */
public void setPos(final Point2Int source) {

    this.posX = source.getPosX();
    this.posY = source.getPosY();
}

/**
 * Moves the coordinates of the point.
 *
 * @param  xDelta  the change to the x coordinate
 * @param  yDelta  the change to the y coordinate
 */
public void move(final double xDelta, final double yDelta) {

    this.posX += xDelta;
    this.posY += yDelta;
}
```

```
    /**
     * Moves the coordinates of the point by a vector.
     *
     * @param  vector  the vector by which to move the point
     */
    public void move(final Vector2Int vector) {

        this.posX += vector.getVecX();
        this.posY += vector.getVecY();
    }

    /**
     * Adds a scaled version of a vector to this point's position (this = this + scale tuple).
     *
     * @param  scale   the scalar value
     * @param  vector  the vector to be scaled then added
     */
    public void moveScaled(final double scale, final Vector2Int vector) {

        this.posX += vector.getVecX() * scale;
        this.posY += vector.getVecY() * scale;
    }

    /**
     * Computes the square of the Euclidean distance between this point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the square of the distance
     */
    public double distSquared(final Point2Int otherPoint) {

        double distX;
        double distY;

        distX = this.posX - otherPoint.getPosX();
        distY = this.posY - otherPoint.getPosY();

        return (distX * distX) + (distY * distY);
    }

    /**
     * Computes the Euclidean distance between this point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the distance
     */
    public double dist(final Point2Int otherPoint) {

        double distX;
        double distY;

        distX = this.posX - otherPoint.getPosX();
        distY = this.posY - otherPoint.getPosY();

        return Math.sqrt((distX * distX) + (distY * distY));
    }

    /**
     * Generates the string representation of the point.
     *
     * @return  the <code>String</code> representation
     */
    @Override public String toString() {

        StringBuilder str;

        str = new StringBuilder(30);

        str.append('(');
        str.append(this.posX);
        str.append(",_");
        str.append(this.posY);
        str.append(')');

        return str.toString();
    }
}

package com.srbenoit.geom;

import com.srbenoit.sparsearray.SparseArray;

/**
 * An array of points that supports addition and deletion of points, but will not change the index
 * of a point once added. That is, deleting points makes the array sparse, and adding new points
 * may fill in gaps left by deleting points before appending to the end of the array. Storage is
```

```
 * allocated in blocks of fixed size as needed to add points.
 */
public class Point2Array extends SparseArray<Point2Int> {

    /**
     * Constructs a new <code>Point2Array</code> with a default capacity.
     */
    public Point2Array() {

        this(16);
    }

    /**
     * Constructs a new <code>Point2Array</code> with capacity for a specified number of points.
     * Use this constructor if you know how many points will ultimately be added to the array.
     *
     * @param  initialSize  the initial capacity of array to allocate
     */
    public Point2Array(final int initialSize) {

        super(Point2Int.class, initialSize);
    }

    /**
     * Transforms the points in this list into another list using a given transformation matrix. It
     * is assumed that the target array contains the same number and arrangement of points as this
     * list.
     *
     * @param  target     the array into which to transform the points
     * @param  transform  the transformation matrix
     */
    public void transformInto(final Point2Array target, final Transform2 transform) {

        int len;

        len = capacity();

        for (int i = 0; i < len; i++) {

            if (isFilled(i)) {
                transform.transformPoint(get(i), target.get(i));
            }
        }
    }
}

package com.srbenoit.geom;

/**
 * An interface for objects that can be represented by a 2-dimensional point.
 */
public interface Point2Int {

    /**
     * Gets the X coordinate.
     *
     * @return  the X coordinate
     */
    double getPosX();

    /**
     * Sets the X coordinate.
     *
     * @param  xCoord  the X coordinate
     */
    void setPosX(double xCoord);

    /**
     * Gets the Y coordinate.
     *
     * @return  the Y coordinate
     */
    double getPosY();

    /**
     * Sets the Y coordinate.
     *
     * @param  yCoord  the Y coordinate
     */
    void setPosY(double yCoord);

    /**
     * Sets the coordinates of the point.
     *
     * @param  xCoord  the x coordinate
     * @param  yCoord  the y coordinate
     */
```

```
    void setPos(double xCoord, double yCoord);

    /**
     * Sets the coordinates of the point from another point.
     *
     * @param  source  the point whose position is to be copied
     */
    void setPos(Point2Int source);

    /**
     * Moves the coordinates of the point.
     *
     * @param  xDelta  the change to the x coordinate
     * @param  yDelta  the change to the y coordinate
     */
    void move(double xDelta, double yDelta);

    /**
     * Moves the coordinates of the point by a vector.
     *
     * @param  vector  the vector by which to move the point
     */
    void move(final Vector2Int vector);

    /**
     * Adds a scaled version of a vector to this point's position (this = this + scale tuple).
     *
     * @param  scale   the scalar value
     * @param  vector  the vector to be scaled then added
     */
    void moveScaled(double scale, Vector2Int vector);

    /**
     * Computes the square of the Euclidean distance between this point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the square of the distance
     */
    double distSquared(Point2Int otherPoint);

    /**
     * Computes the Euclidean distance between this point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the distance
     */
    double dist(Point2Int otherPoint);
}

package com.srbenoit.geom;

import com.srbenoit.log.LoggedObject;

/**
 * A simple three-dimensional point.
 */
public class Point3 extends LoggedObject implements Point3Int {

    /** the X coordinate */
    private double posX;

    /** the Y coordinate */
    private double posY;

    /** the Z coordinate */
    private double posZ;

    /**
     * Constructs a point at the origin.
     */
    public Point3() {

        this.posX = 0;
        this.posY = 0;
        this.posZ = 0;
    }

    /**
     * Constructs a point.
     *
     * @param  xCoord  the X coordinate
     * @param  yCoord  the Y coordinate
     * @param  zCoord  the Z coordinate
     */
    public Point3(final double xCoord, final double yCoord, final double zCoord) {

        this.posX = xCoord;
```

```
        this.posY = yCoord;
        this.posZ = zCoord;
    }

    /**
     * Constructs a point.
     *
     * @param  source  the point whose coordinates are to be copied
     */
    public Point3(final Point3Int source) {

        this.posX = source.getPosX();
        this.posY = source.getPosY();
        this.posZ = source.getPosZ();
    }

    /**
     * Gets the X coordinate.
     *
     * @return  the X coordinate
     */
    public double getPosX() {

        return this.posX;
    }

    /**
     * Sets the X coordinate.
     *
     * @param  xCoord  the X coordinate
     */
    public void setPosX(final double xCoord) {

        this.posX = xCoord;
    }

    /**
     * Gets the Y coordinate.
     *
     * @return  the Y coordinate
     */
    public double getPosY() {

        return this.posY;
    }

    /**
     * Sets the Y coordinate.
     *
     * @param  yCoord  the Y coordinate
     */
    public void setPosY(final double yCoord) {

        this.posY = yCoord;
    }

    /**
     * Gets the Z coordinate.
     *
     * @return  the Z coordinate
     */
    public double getPosZ() {

        return this.posZ;
    }

    /**
     * Sets the Z coordinate.
     *
     * @param  zCoord  the Z coordinate
     */
    public void setPosZ(final double zCoord) {

        this.posZ = zCoord;
    }

    /**
     * Sets the coordinates of the point.
     *
     * @param  xCoord  the x coordinate
     * @param  yCoord  the y coordinate
     * @param  zCoord  the z coordinate
     */
    public void setPos(final double xCoord, final double yCoord, final double zCoord) {

        this.posX = xCoord;
        this.posY = yCoord;
```

```
        this.posZ = zCoord;
    }

    /**
     * Sets the coordinates of the point from another point.
     *
     * @param  source  the point whose position is to be copied
     */
    public void setPos(final Point3Int source) {

        this.posX = source.getPosX();
        this.posY = source.getPosY();
        this.posZ = source.getPosZ();
    }

    /**
     * Moves the coordinates of the point.
     *
     * @param  xDelta  the change to the x coordinate
     * @param  yDelta  the change to the y coordinate
     * @param  zDelta  the change to the z coordinate
     */
    public void move(final double xDelta, final double yDelta, final double zDelta) {

        this.posX += xDelta;
        this.posY += yDelta;
        this.posZ += zDelta;
    }

    /**
     * Moves the coordinates of the point by a vector.
     *
     * @param  vector  the vector by which to move the point
     */
    public void move(final Vector3Int vector) {

        this.posX += vector.getVecX();
        this.posY += vector.getVecY();
        this.posZ += vector.getVecZ();
    }

    /**
     * Adds a scaled version of a vector to this point's position (this = this + scale tuple).
     *
     * @param  scale   the scalar value
     * @param  vector  the vector to be scaled then added
     */
    public void moveScaled(final double scale, final Vector3Int vector) {

        this.posX += vector.getVecX() * scale;
        this.posY += vector.getVecY() * scale;
        this.posZ += vector.getVecZ() * scale;
    }

    /**
     * Computes the square of the Euclidean distance between this point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the square of the distance
     */
    public double distSquared(final Point3Int otherPoint) {

        double distX;
        double distY;
        double distZ;

        distX = this.posX - otherPoint.getPosX();
        distY = this.posY - otherPoint.getPosY();
        distZ = this.posZ - otherPoint.getPosZ();

        return (distX * distX) + (distY * distY) + (distZ * distZ);
    }

    /**
     * Computes the Euclidean distance between this point and another point.
     *
     * @param   otherPoint  the other point
     * @return  the distance
     */
    public double dist(final Point3Int otherPoint) {

        double distX;
        double distY;
        double distZ;

        distX = this.posX - otherPoint.getPosX();
        distY = this.posY - otherPoint.getPosY();
```

```
        distZ = this.posZ - otherPoint.getPosZ();

        return Math.sqrt((distX * distX) + (distY * distY) + (distZ * distZ));
    }

    /**
     * Generates the string representation of the point.
     *
     * @return  the <code>String</code> representation
     */
    @Override public String toString() {

        StringBuilder str;

        str = new StringBuilder(50);

        str.append('(');
        str.append(this.posX);
        str.append(", ");
        str.append(this.posY);
        str.append(", ");
        str.append(this.posZ);
        str.append(')');

        return str.toString();
    }
}
```

```
package com.srbenoit.geom;

import com.srbenoit.sparsearray.SparseArray;

/**
 * An array of points that supports addition and deletion of points, but will not change the index
 * of a point once added. That is, deleting points makes the array sparse, and adding new points
 * may fill in gaps left by deleting points before appending to the end of the array. Storage is
 * allocated in blocks of fixed size as needed to add points.
 */
public class Point3Array extends SparseArray<Point3Int> {

    /**
     * Constructs a new <code>Point3Array</code> with a default capacity.
     */
    public Point3Array() {

        this(16);
    }

    /**
     * Constructs a new <code>Point3Array</code> with capacity for a specified number of points.
     * Use this constructor if you know how many points will ultimately be added to the array.
     *
     * @param  initialSize  the initial capacity of array to allocate
     */
    public Point3Array(final int initialSize) {

        super(Point3Int.class, initialSize);
    }

    /**
     * Transforms the points in this list into another list using a given transformation matrix. It
     * is assumed that the target array contains the same number and arrangement of points as this
     * list.
     *
     * @param  target     the array into which to transform the points
     * @param  transform  the transformation matrix
     */
    public void transformInto(final Point3Array target, final Transform3 transform) {

        int len;

        len = capacity();

        for (int i = 0; i < len; i++) {

            if (isFilled(i)) {
                transform.transformPoint(get(i), target.get(i));
            }
        }
    }
}
```

```
package com.srbenoit.geom;

/**
 * An interface for objects that can be represented by a 3-dimensional point.
 */
```

```
public interface Point3Int {

    /**
     * Gets the X coordinate.
     *
     * @return  the X coordinate
     */
    double getPosX();

    /**
     * Sets the X coordinate.
     *
     * @param  xCoord  the X coordinate
     */
    void setPosX(double xCoord);

    /**
     * Gets the Y coordinate.
     *
     * @return  the Y coordinate
     */
    double getPosY();

    /**
     * Sets the Y coordinate.
     *
     * @param  yCoord  the Y coordinate
     */
    void setPosY(double yCoord);

    /**
     * Gets the Z coordinate.
     *
     * @return  the Z coordinate
     */
    double getPosZ();

    /**
     * Sets the Z coordinate.
     *
     * @param  zCoord  theZ coordinate
     */
    void setPosZ(double zCoord);

    /**
     * Sets the coordinates of the point.
     *
     * @param  xCoord  the x coordinate
     * @param  yCoord  the y coordinate
     * @param  zCoord  the z coordinate
     */
    void setPos(double xCoord, double yCoord, double zCoord);

    /**
     * Sets the coordinates of the point from another point.
     *
     * @param  source  the point whose position is to be copied
     */
    void setPos(Point3Int source);

    /**
     * Moves the coordinates of the point.
     *
     * @param  xDelta  the change to the x coordinate
     * @param  yDelta  the change to the y coordinate
     * @param  zDelta  the change to the z coordinate
     */
    void move(double xDelta, double yDelta, double zDelta);

    /**
     * Moves the coordinates of the point by a vector.
     *
     * @param  vector  the vector by which to move the point
     */
    void move(final Vector3Int vector);

    /**
     * Adds a scaled version of a vector to this point's position (this = this + scale tuple).
     *
     * @param  scale   the scalar value
     * @param  vector  the vector to be scaled then added
     */
    void moveScaled(double scale, Vector3Int vector);

    /**
     * Computes the square of the Euclidean distance between this point and another point.
     *
```

```
     * @param    otherPoint  the other point
     * @return   the square of the distance
     */
    double distSquared(Point3Int otherPoint);

    /**
     * Computes the Euclidean distance between this point and another point.
     *
     * @param    otherPoint  the other point
     * @return   the distance
     */
    double dist(Point3Int otherPoint);
}
```

```
package com.srbenoit.geom;

/**
 * An exception thrown when an attempt is made to invert a singular matrix.
 */
public final class SingularMatrixException extends RuntimeException {

    /** version number for serialization */
    private static final long serialVersionUID = -5321393728885816859L;

    /**
     * Constructs a new <code>SingularMatrixException</code> with <code>null</code> as its detail
     * message.
     */
    public SingularMatrixException() {

        super();
    }

    /**
     * Constructs a new <code>SingularMatrixException</code> with the specified detail message.
     *
     * @param  message  the detail message
     */
    public SingularMatrixException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>SingularMatrixException</code> with the specified detail message and
     * cause.
     *
     * <p>Note that the detail message associated with <code>cause</code> is <i>not</i>
     * automatically incorporated in this runtime exception's detail message.
     *
     * @param  message  the detail message
     * @param  cause    the cause
     */
    public SingularMatrixException(final String message, final Throwable cause) {

        super(message, cause);
    }

    /**
     * Constructs a new <code>SingularMatrixException</code> with the specified cause and a detail
     * message of <code>(cause==null ? null : cause.toString())</code> (which typically contains the
     * class and detail message of <code>cause</code> ).
     *
     * @param  cause  the cause
     */
    public SingularMatrixException(final Throwable cause) {

        super(cause);
    }
}
```

```
package com.srbenoit.geom;

/**
 * A transformation matrix capable representing an affine transformation and acting on vectors. We
 * store only 6 matrix elements rather than 9 since three of the elements are constant.
 */
public final class Transform2 {

    /** the first element of the first row */
    private transient double m00;

    /** the second element of the first row */
    private transient double m01;

    /** the third element of the first row */
    private transient double m02;
```

```
/** the first element of the second row */
private transient double m10;

/** the second element of the second row */
private transient double m11;

/** the third element of the second row */
private transient double m12;

/**
 * Constructs and initializes a <code>Transform2</code> to the identity transformation.
 */
public Transform2() {

    this.m00 = 1.0;
    this.m01 = 0.0;
    this.m02 = 0.0;

    this.m10 = 0.0;
    this.m11 = 1.0;
    this.m12 = 0.0;
}

/**
 * Returns a string that contains the values of this <code>Transform2</code>.
 *
 * @return  the <code>String</code> representation
 */
@Override public String toString() {

    return this.m00 + ", " + this.m01 + ", " + this.m02 + "\n" + this.m10 + ", " + this.m11
        + ", " + this.m12 + "\n";
}

/**
 * Sets the value of an element in the matrix.
 *
 * @param  row    the row
 * @param  col    the column
 * @param  value  the new value to place at that location in the matrix
 */
public void set(final int row, final int col, final double value) {

    switch (row) {

    case 0:
        switch (col) {

        case 0:
            this.m00 = value;
            break;

        case 1:
            this.m01 = value;
            break;

        case 2:
            this.m02 = value;
            break;

        default:
            break;
        }

        break;

    case 1:
        switch (col) {

        case 0:
            this.m10 = value;
            break;

        case 1:
            this.m11 = value;
            break;

        case 2:
            this.m12 = value;
            break;

        default:
            break;
        }

        break;
```

```
        default:

            break;
        }
    }

    /**
     * Gets the value of an element in the matrix.
     *
     * @param   row  the row
     * @param   col  the column
     * @return  the value at that location in the matrix
     */
    public double get(final int row, final int col) {

        double value;

        switch (row) {

        case 0:
            switch (col) {

            case 0:
                value = this.m00;
                break;

            case 1:
                value = this.m01;
                break;

            case 2:
                value = this.m02;
                break;

            default:
                value = 0;
                break;
            }

            break;

        case 1:
            switch (col) {

            case 0:
                value = this.m10;
                break;

            case 1:
                value = this.m11;
                break;

            case 2:
                value = this.m12;
                break;

            default:
                value = 0;
                break;
            }

            break;

        default:
            value = 0;
            break;
        }

        return value;
    }

    /**
     * Sets the elements of this matrix from another matrix.
     *
     * @param  matrix  the source matrix
     */
    public void set(final Transform2 matrix) {

        this.m00 = matrix.get(0, 0);
        this.m01 = matrix.get(0, 1);
        this.m02 = matrix.get(0, 2);
        this.m10 = matrix.get(1, 0);
        this.m11 = matrix.get(1, 1);
        this.m12 = matrix.get(1, 2);
    }
```

```
/**
 * Right-multiplies this matrix by <code>matrix</code>.
 *
 * @param  matrix  the matrix by which to multiply this matrix
 */
public void mul(final Transform2 matrix) {

    double c00;
    double c01;
    double c02;
    double c10;
    double c11;
    double c12;

    c00 = (this.m00 * matrix.get(0, 0)) + (this.m01 * matrix.get(1, 0));
    c01 = (this.m00 * matrix.get(0, 1)) + (this.m01 * matrix.get(1, 1));
    c02 = (this.m00 * matrix.get(0, 2)) + (this.m01 * matrix.get(1, 2)) + this.m02;

    c10 = (this.m10 * matrix.get(0, 0)) + (this.m11 * matrix.get(1, 0));
    c11 = (this.m10 * matrix.get(0, 1)) + (this.m11 * matrix.get(1, 1));
    c12 = (this.m10 * matrix.get(0, 2)) + (this.m11 * matrix.get(1, 2)) + this.m12;

    this.m00 = c00;
    this.m01 = c01;
    this.m02 = c02;
    this.m10 = c10;
    this.m11 = c11;
    this.m12 = c12;
}

/**
 * Inverts the <code>MatrixF2</code> in place. Rather than use an LU decomposition, it is
 * faster to just directly compute the inverse of a 3x3 matrix where the bottom row is [0 0 1]
 * directly.
 *
 * @throws  SingularMatrixException  if the matrix is singular (not invertible)
 */
public void invert() throws SingularMatrixException {

    double det;
    double recip;
    double c00;
    double c01;
    double c02;
    double c10;
    double c11;
    double c12;

    // Compute the determinant and see if it is zero (non-invertible)
    det = (this.m00 * this.m11) - (this.m01 * this.m10);

    if (Math.abs(det) < 0.0001) {
        throw new SingularMatrixException();
    }

    recip = 1.0 / det;

    c00 = this.m11 * recip;
    c01 = -this.m01 * recip;
    c10 = -this.m10 * recip;
    c11 = this.m00 * recip;

    c02 = (-c01 * this.m12) - (c00 * this.m02);
    c12 = (-c10 * this.m02) - (c11 * this.m12);

    this.m00 = c00;
    this.m01 = c01;
    this.m02 = c02;
    this.m10 = c10;
    this.m11 = c11;
    this.m12 = c12;
}

/**
 * Transforms the point <code>pointIn</code> using this matrix and places the result into
 * <code>pointOut</code>.
 *
 * @param  pointIn   the point to be transformed
 * @param  pointOut  the point into which the transformed values are placed
 */
public void transformPoint(final Point2Int pointIn, final Point2Int pointOut) {

    pointOut.setPos((this.m00 * pointIn.getPosX()) + (this.m01 * pointIn.getPosY()) + this.m02,
        (this.m10 * pointIn.getPosX()) + (this.m11 * pointIn.getPosY()) + this.m12);
}

/**
```

```
     * Transforms the vector <code>vecIn</code> using this matrix and places the result into <code>
     * vecOut</code>.
     *
     * @param  vecIn   the vector to be transformed
     * @param  vecOut  the vector into which the transformed values are placed
     */
    public void transformVec(final Vector2Int vecIn, final Vector2Int vecOut) {

        vecOut.setVec((this.m00 * vecIn.getVecX()) + (this.m01 * vecIn.getVecY()),
            (this.m10 * vecIn.getVecX()) + (this.m11 * vecIn.getVecY()));
    }
}

package com.srbenoit.geom;

/**
 * A transformation matrix capable representing an affine transformation and acting on vectors. We
 * store only 12 matrix elements rather than 16 since four of the elements are constant.
 */
public final class Transform3 {

    /** the first element of the first row */
    private transient double m00;

    /** the second element of the first row */
    private transient double m01;

    /** the third element of the first row */
    private transient double m02;

    /** the fourth element of the first row */
    private transient double m03;

    /** the first element of the second row */
    private transient double m10;

    /** the second element of the second row */
    private transient double m11;

    /** the third element of the second row */
    private transient double m12;

    /** the fourth element of the second row */
    private transient double m13;

    /** the first element of the third row */
    private transient double m20;

    /** the second element of the third row */
    private transient double m21;

    /** the third element of the third row */
    private transient double m22;

    /** the fourth element of the third row */
    private transient double m23;

    /** working array */
    private transient double[] rowScale;

    /** working array */
    private transient double[] temp;

    /** working array */
    private transient double[] result;

    /** working array */
    private transient int[] rowPerm;

    /**
     * Constructs and initializes a <code>Transform3</code> to the identity transformation.
     */
    public Transform3() {

        this.m00 = 1.0f;
        this.m01 = 0.0f;
        this.m02 = 0.0f;
        this.m03 = 0.0f;

        this.m10 = 0.0f;
        this.m11 = 1.0f;
        this.m12 = 0.0f;
        this.m13 = 0.0f;

        this.m20 = 0.0f;
        this.m21 = 0.0f;
        this.m22 = 1.0f;
```

```
        this.m23 = 0.0f;

        this.rowScale = new double[4];

        this.temp = new double[16];
        this.result = new double[16];
        this.rowPerm = new int[4];
    }

    /**
     * Returns a string that contains the values of this <code>Transform3</code>.
     *
     * @return  the <code>String</code> representation
     */
    @Override public String toString() {

        return this.m00 + ", " + this.m01 + ", " + this.m02 + ", " + this.m03 + "\n" + this.m10
            + ", " + this.m11 + ", " + this.m12 + ", " + this.m13 + "\n" + this.m20 + ", "
            + this.m21 + ", " + this.m22 + ", " + this.m23 + "\n";
    }

    /**
     * Sets the value of an element in the matrix.
     *
     * @param  row     the row
     * @param  col     the column
     * @param  value   the new value to place at that location in the matrix
     */
    public void set(final int row, final int col, final double value) {

        switch (row) {

        case 0:
            switch (col) {

            case 0:
                this.m00 = value;
                break;

            case 1:
                this.m01 = value;
                break;

            case 2:
                this.m02 = value;
                break;

            case 3:
                this.m03 = value;
                break;

            default:
                break;
            }

            break;

        case 1:
            switch (col) {

            case 0:
                this.m10 = value;
                break;

            case 1:
                this.m11 = value;
                break;

            case 2:
                this.m12 = value;
                break;

            case 3:
                this.m13 = value;
                break;

            default:
                break;
            }

            break;

        case 2:
            switch (col) {

            case 0:
                this.m20 = value;
```

```
            break;

        case 1:
            this.m21 = value;
            break;

        case 2:
            this.m22 = value;
            break;

        case 3:
            this.m23 = value;
            break;

        default:
            break;
        }

        break;

    default:

        break;
    }
}

/**
 * Gets the value of an element in the matrix.
 *
 * @param   row  the row
 * @param   col  the column
 * @return  the value at that location in the matrix
 */
public double get(final int row, final int col) {

    double value;

    switch (row) {

    case 0:
        switch (col) {

        case 0:
            value = this.m00;
            break;

        case 1:
            value = this.m01;
            break;

        case 2:
            value = this.m02;
            break;

        case 3:
            value = this.m03;
            break;

        default:
            value = 0;
            break;
        }

        break;

    case 1:
        switch (col) {

        case 0:
            value = this.m10;
            break;

        case 1:
            value = this.m11;
            break;

        case 2:
            value = this.m12;
            break;

        case 3:
            value = this.m13;
            break;

        default:
            value = 0;
            break;
```

```
            }

            break;

        case 2:
            switch (col) {

            case 0:
                value = this.m20;
                break;

            case 1:
                value = this.m21;
                break;

            case 2:
                value = this.m22;
                break;

            case 3:
                value = this.m23;
                break;

            default:
                value = 0;
                break;
            }

            break;

        default:
            value = 0;
            break;
        }

        return value;
    }

    /**
     * Sets the elements of this matrix from another matrix.
     *
     * @param  matrix  the source matrix
     */
    public void set(final Transform3 matrix) {

        this.m00 = matrix.get(0, 0);
        this.m01 = matrix.get(0, 1);
        this.m02 = matrix.get(0, 2);
        this.m03 = matrix.get(0, 3);

        this.m10 = matrix.get(1, 0);
        this.m11 = matrix.get(1, 1);
        this.m12 = matrix.get(1, 2);
        this.m13 = matrix.get(1, 3);

        this.m20 = matrix.get(2, 0);
        this.m21 = matrix.get(2, 1);
        this.m22 = matrix.get(2, 2);
        this.m23 = matrix.get(2, 3);
    }

    /**
     * Right-multiplies this matrix by <code>matrix</code>.
     *
     * @param  matrix  the matrix by which to multiply this matrix
     */
    public void mul(final Transform3 matrix) {

        double c00;
        double c01;
        double c02;
        double c03;
        double c10;
        double c11;
        double c12;
        double c13;
        double c20;
        double c21;
        double c22;
        double c23;

        c00 = (this.m00 * matrix.get(0, 0)) + (this.m01 * matrix.get(1, 0))
            + (this.m02 * matrix.get(2, 0));
        c01 = (this.m00 * matrix.get(0, 1)) + (this.m01 * matrix.get(1, 1))
            + (this.m02 * matrix.get(2, 1));
        c02 = (this.m00 * matrix.get(0, 2)) + (this.m01 * matrix.get(1, 2))
            + (this.m02 * matrix.get(2, 2));
```

```
        c03 = (this.m00 * matrix.get(0, 3)) + (this.m01 * matrix.get(1, 3))
            + (this.m02 * matrix.get(2, 3)) + this.get(0, 3);

        c10 = (this.m10 * matrix.get(0, 0)) + (this.m11 * matrix.get(1, 0))
            + (this.m12 * matrix.get(2, 0));
        c11 = (this.m10 * matrix.get(0, 1)) + (this.m11 * matrix.get(1, 1))
            + (this.m12 * matrix.get(2, 1));
        c12 = (this.m10 * matrix.get(0, 2)) + (this.m11 * matrix.get(1, 2))
            + (this.m12 * matrix.get(2, 2));
        c13 = (this.m10 * matrix.get(0, 3)) + (this.m11 * matrix.get(1, 3))
            + (this.m12 * matrix.get(2, 3)) + this.get(1, 3);

        c20 = (this.m20 * matrix.get(0, 0)) + (this.m21 * matrix.get(1, 0))
            + (this.m22 * matrix.get(2, 0));
        c21 = (this.m20 * matrix.get(0, 1)) + (this.m21 * matrix.get(1, 1))
            + (this.m22 * matrix.get(2, 1));
        c22 = (this.m20 * matrix.get(0, 2)) + (this.m21 * matrix.get(1, 2))
            + (this.m22 * matrix.get(2, 2));
        c23 = (this.m20 * matrix.get(0, 3)) + (this.m21 * matrix.get(1, 3))
            + (this.m22 * matrix.get(2, 3)) + this.get(2, 3);

        this.m00 = c00;
        this.m01 = c01;
        this.m02 = c02;
        this.m03 = c03;
        this.m10 = c10;
        this.m11 = c11;
        this.m12 = c12;
        this.m13 = c13;
        this.m20 = c20;
        this.m21 = c21;
        this.m22 = c22;
        this.m23 = c23;
    }

    /**
     * Inverts the <code>Transform3</code> in place.
     *
     * @throws  SingularMatrixException  if the matrix is singular (not invertible)
     */
    public void invert() throws SingularMatrixException {

        int inx;

        // Use LU decomposition and back-substitution code specifically
        // for 4x4 matrices.

        // Copy source matrix to t1tmp
        this.temp[0] = this.m00;
        this.temp[1] = this.m01;
        this.temp[2] = this.m02;
        this.temp[3] = this.m03;

        this.temp[4] = this.m10;
        this.temp[5] = this.m11;
        this.temp[6] = this.m12;
        this.temp[7] = this.m13;

        this.temp[8] = this.m20;
        this.temp[9] = this.m21;
        this.temp[10] = this.m22;
        this.temp[11] = this.m23;

        this.temp[12] = 0.0;
        this.temp[13] = 0.0;
        this.temp[14] = 0.0;
        this.temp[15] = 1.0;

        // Calculate LU decomposition: Is the matrix singular?
        luDecomposition(this.temp, this.rowPerm);

        // Perform back substitution on the identity matrix
        for (inx = 0; inx < 16; inx++) {
            this.result[inx] = 0.0;
        }

        this.result[0] = 1.0;
        this.result[5] = 1.0;
        this.result[10] = 1.0;
        this.result[15] = 1.0;
        luBacksubstitution(this.temp, this.rowPerm, this.result);

        this.m00 = this.result[0];
        this.m01 = this.result[1];
        this.m02 = this.result[2];
        this.m03 = this.result[3];
```

```
        this.m10 = this.result[4];
        this.m11 = this.result[5];
        this.m12 = this.result[6];
        this.m13 = this.result[7];

        this.m20 = this.result[8];
        this.m21 = this.result[9];
        this.m22 = this.result[10];
        this.m23 = this.result[11];
    }

    /**
     * Given a 4x4 array <code>matrix0</code>, this function replaces it with the LU decomposition
     * of a row-wise permutation of itself. This function is similar to luDecomposition, except
     * that it is tuned specifically for 4x4 matrices.
     *
     * @param   matrix0    the matrix that is to be decomposed, and (on completion), the decomposed
     *                     matrix
     * @param   rowPerms   the row permutations resulting from partial pivoting
     * @throws  SingularMatrixException  if the matrix is singular (not invertible)
     */
    private void luDecomposition(final double[] matrix0, final int[] rowPerms)
        throws SingularMatrixException {

        int row;
        int col;
        int inx;
        double big;
        double tempr;
        int mtx;
        int imax;
        int knx;
        int target;
        int pt1;
        int pt2;
        double sum;

        // For each row ...
        for (row = 0; row < 4; row++) {

            // Find the largest element in the row
            big = 0.0;

            for (col = 0; col < 4; col++) {
                tempr = Math.abs(matrix0[col]);

                if (tempr > big) {
                    big = tempr;
                }
            }

            // Is the matrix singular?
            if (big == 0.0) {
                throw new SingularMatrixException();
            }

            this.rowScale[row] = 1.0 / big;
        }

        mtx = 0;

        // For all columns, execute Crout's method
        for (col = 0; col < 4; col++) {

            // Determine elements of upper diagonal matrix U
            for (inx = 0; inx < col; inx++) {
                target = mtx + (4 * inx) + col;
                sum = matrix0[target];
                pt1 = mtx + (4 * inx);
                pt2 = mtx + col;

                for (knx = 0; knx < inx; knx++) {
                    sum -= matrix0[pt1] * matrix0[pt2];
                    pt1++;
                    pt2 += 4;
                }

                matrix0[target] = sum;
            }

            // Search for largest pivot element and calculate
            // intermediate elements of lower diagonal matrix L.
            big = 0.0;
            imax = -1;

            for (inx = col; inx < 4; inx++) {
                target = mtx + (4 * inx) + col;
```

```
                sum = matrix0[target];
                pt1 = mtx + (4 * inx);
                pt2 = mtx + col;

                for (knx = 0; knx < col; knx++) {
                    sum -= matrix0[pt1] * matrix0[pt2];
                    pt1++;
                    pt2 += 4;
                }

                matrix0[target] = sum;

                // Is this the best pivot so far?
                tempr = this.rowScale[inx] * Math.abs(sum);

                if (tempr >= big) {
                    big = tempr;
                    imax = inx;
                }
            }

            if (imax < 0) {
                throw new SingularMatrixException();
            }

            // Is a row exchange necessary?
            if (col != imax) {

                // Yes: exchange rows
                pt1 = mtx + (4 * imax);
                pt2 = mtx + (4 * col);

                for (knx = 0; knx < 4; knx++) {
                    tempr = matrix0[pt1];
                    matrix0[pt1] = matrix0[pt2];
                    matrix0[pt2] = tempr;
                    pt1++;
                    pt2++;
                }

                // Record change in scale factor
                this.rowScale[imax] = this.rowScale[col];
            }

            // Record row permutation
            rowPerms[col] = imax;

            // Is the matrix singular
            if (matrix0[(mtx + (4 * col) + col)] == 0.0) {
                throw new SingularMatrixException();
            }

            // Divide elements of lower diagonal matrix L by pivot
            if (col != (4 - 1)) {
                tempr = 1.0 / (matrix0[(mtx + (4 * col) + col)]);
                target = mtx + (4 * (col + 1)) + col;

                for (inx = 0; inx < (3 - col); inx++) {
                    matrix0[target] *= tempr;
                    target += 4;
                }
            }
        }
    }

    /**
     * Solves a set of linear equations.
     *
     * @param   matrix1    the matrix produced by <code>luDecomposition</code> (not changed by this
     *                     method)
     * @param   rowPerms   the row permutations resulting from partial pivoting produced by <code>
     *                     luDecomposition</code> (not changed by this method)
     * @param   matrix2    a set of column vectors assembled into a 4x4 matrix of values (the
     *                     procedure takes each column of "matrix2" in turn and treats it as the
     *                     right-hand side of the matrix equation Ax = LUx = b. The solution vector
     *                     replaces the original column of the matrix. If <code>matrix2</code> is the
     *                     identity matrix, the procedure replaces its contents with the inverse of
     *                     the matrix from which <code>matrix1</code> was originally derived)
     */
    private void luBacksubstitution(final double[] matrix1, final int[] rowPerms,
        final double[] matrix2) {

        int prow;
        int perm;
        int jnx;
        double sum;
```

```
            for (int col = 0; col < 4; col++) {

                // Forward substitution
                prow = -1;

                for (int row = 0; row < 4; row++) {
                    perm = rowPerms[row];
                    sum = matrix2[col + (4 * perm)];
                    matrix2[col + (4 * perm)] = matrix2[col + (4 * row)];

                    if (prow >= 0) {

                        for (jnx = prow; jnx <= (row - 1); jnx++) {
                            sum -= matrix1[(row * 4) + jnx] * matrix2[col + (4 * jnx)];
                        }
                    } else if (sum != 0.0) {
                        prow = row;
                    }

                    matrix2[col + (4 * row)] = sum;
                }

                // Back substitution
                matrix2[col + 12] /= matrix1[15];

                matrix2[col + 8] = (matrix2[col + 8] - (matrix1[11] * matrix2[col + 12]))
                    / matrix1[10];

                matrix2[col + 4] = (matrix2[col + 4] - (matrix1[6] * matrix2[col + 8])
                        - (matrix1[7] * matrix2[col + 12])) / matrix1[5];

                matrix2[col] = (matrix2[col] - (matrix1[1] * matrix2[col + 4])
                        - (matrix1[2] * matrix2[col + 8]) - (matrix1[3] * matrix2[col + 12]))
                    / matrix1[0];
            }
        }

        /**
         * Transform the point <code>pointIn</code> using this <code>Transform3</code> and place the
         * result into <code>pointOut</code>. This method permits the same tuple to be used as the
         * source and destination.
         *
         * @param  pointIn   the point to be transformed
         * @param  pointOut  the point into which the transformed values are placed
         */
        public void transformPoint(final Point3Int pointIn, final Point3Int pointOut) {

            pointOut.setPos((this.m00 * pointIn.getPosX()) + (this.m01 * pointIn.getPosY())
                + (this.m02 * pointIn.getPosZ()) + this.m03,
                (this.m10 * pointIn.getPosX()) + (this.m11 * pointIn.getPosY())
                + (this.m12 * pointIn.getPosZ()) + this.m13,
                (this.m20 * pointIn.getPosX()) + (this.m21 * pointIn.getPosY())
                + (this.m22 * pointIn.getPosZ()) + this.m23);
        }

        /**
         * Transform the vector <code>vecIn</code> using this <code>Transform3</code> and place the
         * result into <code>vecOut</code>. This method permits the same tuple to be used as the source
         * and destination.
         *
         * @param  vecIn   the vector to be transformed
         * @param  vecOut  the vector into which the transformed values are placed
         */
        public void transformVec(final Vector3Int vecIn, final Vector3Int vecOut) {

            vecOut.setVec((this.m00 * vecIn.getVecX()) + (this.m01 * vecIn.getVecY())
                + (this.m02 * vecIn.getVecZ()),
                (this.m10 * vecIn.getVecX()) + (this.m11 * vecIn.getVecY())
                + (this.m12 * vecIn.getVecZ()),
                (this.m20 * vecIn.getVecX()) + (this.m21 * vecIn.getVecY())
                + (this.m22 * vecIn.getVecZ()));
        }
    }

    package com.srbenoit.geom;

    import com.srbenoit.log.LoggedObject;

    /**
     * A simple two-dimensional vector.
     */
    public class Vector2 extends LoggedObject implements Vector2Int {

        /** the X component */
        private double vecX;

        /** the Y component */
```

```
private double vecY;

/** the length, computed lazily */
private double len;

/**
 * Constructs a null vector.
 */
public Vector2() {

    this.vecX = 0;
    this.vecY = 0;
    this.len = 0;
}

/**
 * Constructs a vector.
 *
 * @param  xCoord  the X component
 * @param  yCoord  the Y component
 */
public Vector2(final double xCoord, final double yCoord) {

    this.vecX = xCoord;
    this.vecY = yCoord;
    this.len = -1;
}

/**
 * Constructs a vector.
 *
 * @param  source  the vector whose components are to be copied
 */
public Vector2(final Vector2Int source) {

    this.vecX = source.getVecX();
    this.vecY = source.getVecY();
    this.len = source.lazyLength();
}

/**
 * Gets the X component of the vector.
 *
 * @return  the X component
 */
public double getVecX() {

    return this.vecX;
}

/**
 * Sets the X component of the vector.
 *
 * @param  xComp  the X component
 */
public void setVecX(final double xComp) {

    this.vecX = xComp;
    this.len = -1;
}

/**
 * Gets the Y component of the vector.
 *
 * @return  the Y component
 */
public double getVecY() {

    return this.vecY;
}

/**
 * Sets the Y component of the vector.
 *
 * @param  yComp  the Y component
 */
public void setVecY(final double yComp) {

    this.vecY = yComp;
    this.len = -1;
}

/**
 * Sets the coordinates of the vector.
 *
 * @param  xComp  the x component
 * @param  yComp  the y component
```

```
 */
public void setVec(final double xComp, final double yComp) {

    this.vecX = xComp;
    this.vecY = yComp;
    this.len = -1;
}

/**
 * Sets the coordinates of the vector from another vector.
 *
 * @param  source  the vector whose components are to be copied
 */
public void setVec(final Vector2Int source) {

    this.vecX = source.getVecX();
    this.vecY = source.getVecY();
    this.len = source.lazyLength();
}

/**
 * Adds to the components of the vector.
 *
 * @param  xDelta  the change to the x component
 * @param  yDelta  the change to the y component
 */
public void addVec(final double xDelta, final double yDelta) {

    this.vecX += xDelta;
    this.vecY += yDelta;
    this.len = -1;
}

/**
 * Adds a vector to this vector.
 *
 * @param  vector  the vector to add to this vector
 */
public void addVec(final Vector2Int vector) {

    this.vecX += vector.getVecX();
    this.vecY += vector.getVecY();
    this.len = -1;
}

/**
 * Subtracts a vector from this vector.
 *
 * @param  vector  the vector to subtract from this vector
 */
public void subVec(final Vector2Int vector) {

    this.vecX -= vector.getVecX();
    this.vecY -= vector.getVecY();
    this.len = -1;
}

/**
 * Subtracts <code>vector2</code> from <code>vector1</code> and stores the result in this
 * vector.
 *
 * @param  vector1  the vector from which to subtract
 * @param  vector2  the vector to subtract
 */
public void subVec(final Vector2Int vector1, final Vector2Int vector2) {

    this.vecX = vector2.getVecX() - vector1.getVecX();
    this.vecY = vector2.getVecY() - vector1.getVecY();
    this.len = -1;
}

/**
 * Adds a scaled version of a vector to this vector (this = this + scale tuple).
 *
 * @param  scale   the scalar value
 * @param  vector  the vector to be scaled then added
 */
public void addVecScaled(final double scale, final Vector2Int vector) {

    this.vecX += vector.getVecX() * scale;
    this.vecY += vector.getVecY() * scale;
    this.len = -1;
}

/**
 * Sets this vector to the vector from <code>point1</code> to <code>point2</code> (this =
 * point2 - point1).
```

```
     *
     * @param  fromPoint  the first point
     * @param  toPoint    the second point
     */
    public void vectorBetween(final Point2Int fromPoint, final Point2Int toPoint) {

        this.vecX = toPoint.getPosX() - fromPoint.getPosX();
        this.vecY = toPoint.getPosY() - fromPoint.getPosY();
        this.len = -1;
    }

    /**
     * Negates this vector in place.
     */
    public void negateVec() {

        // Length is not affected
        this.vecX = -this.vecX;
        this.vecY = -this.vecY;
    }

    /**
     * Scales this vector by a scalar factor.
     *
     * @param  scale  the scalar factor
     */
    public void scaleVec(final double scale) {

        this.vecX *= scale;
        this.vecY *= scale;

        // if a length is known, the length is scaled by the absolute value of scale
        if (this.len != -1) {

            if (scale < 0) {
                this.len *= -scale;
            } else {
                this.len *= scale;
            }
        }
    }

    /**
     * Sets this vector to a scaled version of another vector.
     *
     * @param  scale  the scalar factor
     * @param  vec    the vector to scale
     */
    public void scaleVec(double scale, Vector2Int vec) {

        this.vecX = vec.getVecX() * scale;
        this.vecY = vec.getVecY() * scale;

        // if a length is known, the length is scaled by the absolute value of scale
        this.len = vec.lazyLength();

        if (vec.lazyLength() != -1) {

            if (scale < 0) {
                this.len *= -scale;
            } else {
                this.len *= scale;
            }
        }
    }

    /**
     * Gets the lazily computed length of the vector.
     *
     * @return  the length of the vector, or -1 if the length has not yet been computed
     */
    public double lazyLength() {

        return this.len;
    }

    /**
     * Gets the squared length of the vector.
     *
     * @return  the squared length of the vector
     */
    public double lengthSquared() {

        double result;

        if (this.len == -1) {
            result = (this.vecX * this.vecX) + (this.vecY * this.vecY);
```

```
        } else {
            result = this.len * this.len;
        }

        return result;
    }

    /**
     * Gets the length of the vector.
     *
     * @return  the length of the vector
     */
    public double length() {

        double result;

        if (this.len == -1) {
            this.len = Math.sqrt((this.vecX * this.vecX) + (this.vecY * this.vecY));
        }

        return this.len;
    }

    /**
     * Computes the dot product of this vector with another vector.
     *
     * @param   vector  the other vector
     * @return  the dot product
     */
    public double dot(final Vector2Int vector) {

        return (this.vecX * vector.getVecX()) + (this.vecY * vector.getVecY());
    }

    /**
     * Normalizes this vector in place. The null vector is normalized to (1,0).
     */
    public void normalize() {

        double before;

        before = length();

        if (before < Double.MIN_NORMAL) {

            // Don't want to divide by that, so consider it zero.
            this.vecX = 1;
            this.vecY = 0;
        } else {
            this.vecX /= before;
            this.vecY /= before;
        }

        this.len = 1;
    }

    /**
     * Generates the string representation of the point.
     *
     * @return  the <code>String</code> representation
     */
    @Override public String toString() {

        StringBuilder str;

        str = new StringBuilder(30);

        str.append('[');
        str.append(this.vecX);
        str.append(", ");
        str.append(this.vecY);
        str.append(']');

        return str.toString();
    }
}

package com.srbenoit.geom;

import com.srbenoit.sparsearray.SparseArray;

/**
 * An array of vectors that supports addition and deletion of vectors, but will not change the
 * index of a vector once added. That is, deleting vectors makes the array sparse, and adding new
 * vectors may fill in gaps left by deleting vectors before appending to the end of the array.
 * Storage is allocated in blocks of fixed size as needed to add vectors.
 */
```

```
public class Vector2Array extends SparseArray<Vector2Int> {

    /**
     * Constructs a new <code>Vector2Array</code> with a default capacity.
     */
    public Vector2Array() {

        this(16);
    }

    /**
     * Constructs a new <code>Vector2Array</code> with capacity for a specified number of vectors.
     * Use this constructor if you know how many vectors will ultimately be added to the array.
     *
     * @param  initialSize  the initial capacity of array to allocate
     */
    public Vector2Array(final int initialSize) {

        super(Vector2Int.class, initialSize);
    }

    /**
     * Transforms the vectors in this list into another list using a given transformation matrix.
     * It is assumed that the target array contains the same number and arrangement of vectors as
     * this list.
     *
     * @param  target     the array into which to transform the vectors
     * @param  transform  the transformation matrix
     */
    public void transformInto(final Vector2Array target, final Transform2 transform) {

        int len;

        len = capacity();

        for (int i = 0; i < len; i++) {

            if (isFilled(i)) {
                transform.transformVec(get(i), target.get(i));
            }
        }
    }
}
```

```
package com.srbenoit.geom;

/**
 * An interface for objects that can be represented as 2-dimensional vectors.
 */
public interface Vector2Int {

    /**
     * Gets the X component of the vector.
     *
     * @return  the X component
     */
    double getVecX();

    /**
     * Sets the X component of the vector.
     *
     * @param  xComp  the X component
     */
    void setVecX(double xComp);

    /**
     * Gets the Y component of the vector.
     *
     * @return  the Y component
     */
    double getVecY();

    /**
     * Sets the Y component of the vector.
     *
     * @param  yComp  the Y component
     */
    void setVecY(double yComp);

    /**
     * Sets the coordinates of the vector.
     *
     * @param  xComp  the x component
     * @param  yComp  the y component
     */
    void setVec(double xComp, double yComp);
```

```
/**
 * Sets the coordinates of the vector from another vector.
 *
 * @param  source  the vector whose components are to be copied
 */
void setVec(Vector2Int source);

/**
 * Adds to the components of the vector.
 *
 * @param  xDelta  the change to the x component
 * @param  yDelta  the change to the y component
 */
void addVec(double xDelta, double yDelta);

/**
 * Adds a vector to this vector.
 *
 * @param  vector  the vector to add to this vector
 */
void addVec(Vector2Int vector);

/**
 * Subtracts a vector from this vector.
 *
 * @param  vector  the vector to subtract from this vector
 */
void subVec(Vector2Int vector);

/**
 * Subtracts <code>vector2</code> from <code>vector1</code> and stores the result in this
 * vector.
 *
 * @param  vector1 the vector from which to subtract
 * @param  vector2 the vector to subtract
 */
void subVec(Vector2Int vector1, Vector2Int vector2);

/**
 * Adds a scaled version of a vector to this vector (this = this + scale tuple).
 *
 * @param  scale   the scalar value
 * @param  vector  the vector to be scaled then added
 */
void addVecScaled(double scale, Vector2Int vector);

/**
 * Sets this vector to the vector from <code>point1</code> to <code>point2</code> (this =
 * point2 - point1).
 *
 * @param  point1  the first point
 * @param  point2  the second point
 */
void vectorBetween(Point2Int point1, Point2Int point2);

/**
 * Negates this vector in place.
 */
void negateVec();

/**
 * Scales this vector by a scalar factor.
 *
 * @param  scale  the scalar factor
 */
void scaleVec(double scale);

/**
 * Sets this vector to a scaled version of another vector.
 *
 * @param  scale  the scalar factor
 * @param  vec the vector to scale
 */
void scaleVec(double scale, Vector2Int vec);

/**
 * Gets the lazily computed length of the vector.
 *
 * @return  the length of the vector, or -1 if the length has not yet been computed
 */
double lazyLength();

/**
 * Gets the squared length of the vector.
 *
 * @return  the squared length of the vector
 */
```

```
    double lengthSquared();

    /**
     * Gets the length of the vector.
     *
     * @return  the length of the vector
     */
    double length();

    /**
     * Computes the dot product of this vector with another vector.
     *
     * @param   vector  the other vector
     * @return  the dot product
     */
    double dot(Vector2Int vector);

    /**
     * Normalizes this vector in place. The null vector is normalized to (1,0).
     */
    void normalize();
}

package com.srbenoit.geom;

import com.srbenoit.log.LoggedObject;

/**
 * A simple three-dimensional vector.
 */
public class Vector3 extends LoggedObject implements Vector3Int {

    /** the X component */
    private double vecX;

    /** the Y component */
    private double vecY;

    /** the Z component */
    private double vecZ;

    /** the length, computed lazily */
    private double len;

    /**
     * Constructs a null vector.
     */
    public Vector3() {

        this.vecX = 0;
        this.vecY = 0;
        this.vecZ = 0;
        this.len = 0;
    }

    /**
     * Constructs a vector.
     *
     * @param  xCoord  the X component
     * @param  yCoord  the Y component
     * @param  zCoord  the Z component
     */
    public Vector3(final double xCoord, final double yCoord, final double zCoord) {

        this.vecX = xCoord;
        this.vecY = yCoord;
        this.vecZ = zCoord;
        this.len = -1;
    }

    /**
     * Constructs a vector.
     *
     * @param  source  the vector whose components are to be copied
     */
    public Vector3(final Vector3Int source) {

        this.vecX = source.getVecX();
        this.vecY = source.getVecY();
        this.vecZ = source.getVecZ();
        this.len = source.lazyLength();
    }

    /**
     * Gets the X component of the vector.
     *
     * @return  the X component
```

```
     */
    public double getVecX() {

        return this.vecX;
    }

    /**
     * Sets the X component of the vector.
     *
     * @param  xComp  the X component
     */
    public void setVecX(final double xComp) {

        this.vecX = xComp;
        this.len = -1;
    }

    /**
     * Gets the Y component of the vector.
     *
     * @return  the Y component
     */
    public double getVecY() {

        return this.vecY;
    }

    /**
     * Sets the Y component of the vector.
     *
     * @param  yComp  the Y component
     */
    public void setVecY(final double yComp) {

        this.vecY = yComp;
        this.len = -1;
    }

    /**
     * Gets the Z component of the vector.
     *
     * @return  the Z component
     */
    public double getVecZ() {

        return this.vecZ;
    }

    /**
     * Sets the Z component of the vector.
     *
     * @param  zComp  the Z component
     */
    public void setVecZ(final double zComp) {

        this.vecZ = zComp;
        this.len = -1;
    }

    /**
     * Sets the coordinates of the vector.
     *
     * @param  xComp  the x component
     * @param  yComp  the y component
     * @param  zComp  the z component
     */
    public void setVec(final double xComp, final double yComp, final double zComp) {

        this.vecX = xComp;
        this.vecY = yComp;
        this.vecZ = zComp;
        this.len = -1;
    }

    /**
     * Sets the coordinates of the vector from another vector.
     *
     * @param  source  the vector whose components are to be copied
     */
    public void setVec(final Vector3Int source) {

        this.vecX = source.getVecX();
        this.vecY = source.getVecY();
        this.vecZ = source.getVecZ();
        this.len = source.lazyLength();
    }
```

```
/**
 * Adds to the components of the vector.
 *
 * @param  xDelta  the change to the x component
 * @param  yDelta  the change to the y component
 * @param  zDelta  the change to the z component
 */
public void addVec(final double xDelta, final double yDelta, final double zDelta) {

    this.vecX += xDelta;
    this.vecY += yDelta;
    this.vecZ += zDelta;
    this.len = -1;
}

/**
 * Adds a vector to this vector.
 *
 * @param  vector  the vector to add to this vector
 */
public void addVec(final Vector3Int vector) {

    this.vecX += vector.getVecX();
    this.vecY += vector.getVecY();
    this.vecZ += vector.getVecZ();
    this.len = -1;
}

/**
 * Subtracts a vector from this vector.
 *
 * @param  vector  the vector to subtract from this vector
 */
public void subVec(final Vector3Int vector) {

    this.vecX -= vector.getVecX();
    this.vecY -= vector.getVecY();
    this.vecZ -= vector.getVecZ();
    this.len = -1;
}

/**
 * Subtracts <code>vector2</code> from <code>vector1</code> and stores the result in this
 * vector.
 *
 * @param  vector1 the vector from which to subtract
 * @param  vector2 the vector to subtract
 */
public void subVec(final Vector3Int vector1, final Vector3Int vector2) {

    this.vecX = vector2.getVecX() - vector1.getVecX();
    this.vecY = vector2.getVecY() - vector1.getVecY();
    this.vecZ = vector2.getVecZ() - vector1.getVecZ();
    this.len = -1;
}

/**
 * Adds a scaled version of a vector to this vector (this = this + scale tuple).
 *
 * @param  scale   the scalar value
 * @param  vector  the vector to be scaled then added
 */
public void addVecScaled(final double scale, final Vector3Int vector) {

    this.vecX += vector.getVecX() * scale;
    this.vecY += vector.getVecY() * scale;
    this.vecZ += vector.getVecZ() * scale;
    this.len = -1;
}

/**
 * Sets this vector to the vector from <code>point1</code> to <code>point2</code> (this =
 * point2 - point1).
 *
 * @param  fromPoint  the first point
 * @param  toPoint    the second point
 */
public void vectorBetween(final Point3Int fromPoint, final Point3Int toPoint) {

    this.vecX = toPoint.getPosX() - fromPoint.getPosX();
    this.vecY = toPoint.getPosY() - fromPoint.getPosY();
    this.vecZ = toPoint.getPosZ() - fromPoint.getPosZ();
    this.len = -1;
}

/**
 * Negates this vector in place.
```

```
 */
public void negateVec() {

    // Length is not affected
    this.vecX = -this.vecX;
    this.vecY = -this.vecY;
    this.vecZ = -this.vecZ;
}

/**
 * Scales this vector by a scalar factor.
 *
 * @param  scale  the scalar factor
 */
public void scaleVec(final double scale) {

    this.vecX *= scale;
    this.vecY *= scale;
    this.vecZ *= scale;

    // if a length is known, the length is scaled by the absolute value of scale
    if (this.len != -1) {

        if (scale < 0) {
            this.len *= -scale;
        } else {
            this.len *= scale;
        }
    }
}

/**
 * Sets this vector to a scaled version of another vector.
 *
 * @param  scale  the scalar factor
 * @param  vec    the vector to scale
 */
public void scaleVec(double scale, Vector3Int vec) {

    this.vecX = vec.getVecX() * scale;
    this.vecY = vec.getVecY() * scale;
    this.vecZ = vec.getVecZ() * scale;

    // if a length is known, the length is scaled by the absolute value of scale
    this.len = vec.lazyLength();

    if (vec.lazyLength() != -1) {

        if (scale < 0) {
            this.len *= -scale;
        } else {
            this.len *= scale;
        }
    }
}

/**
 * Gets the lazily computed length of the vector.
 *
 * @return  the length of the vector, or -1 if the length has not yet been computed
 */
public double lazyLength() {

    return this.len;
}

/**
 * Gets the squared length of the vector.
 *
 * @return  the squared length of the vector
 */
public double lengthSquared() {

    double result;

    if (this.len == -1) {
        result = (this.vecX * this.vecX) + (this.vecY * this.vecY) + (this.vecZ * this.vecZ);
    } else {
        result = this.len * this.len;
    }

    return result;
}

/**
 * Gets the length of the vector.
 *
```

```
     * @return  the length of the vector
     */
    public double length() {

        double result;

        if (this.len == -1) {
            this.len = Math.sqrt((this.vecX * this.vecX) + (this.vecY * this.vecY)
                    + (this.vecZ * this.vecZ));
        }

        return this.len;
    }

    /**
     * Computes the dot product of this vector with another vector.
     *
     * @param   vector  the other vector
     * @return  the dot product
     */
    public double dot(final Vector3Int vector) {

        return (this.vecX * vector.getVecX()) + (this.vecY * vector.getVecY())
            + (this.vecZ * vector.getVecZ());
    }

    /**
     * Computes the cross product of <code>first</code> and <code>second</code> and stores the
     * result in this vector.
     *
     * @param  first   the first tuple in the cross product
     * @param  second  the second tuple in the cross product
     */
    public void cross(final Vector3Int first, final Vector3Int second) {

        this.vecZ = (first.getVecY() * second.getVecZ()) - (first.getVecZ() * second.getVecY());
        this.vecY = (first.getVecZ() * second.getVecX()) - (first.getVecX() * second.getVecZ());
        this.vecZ = (first.getVecX() * second.getVecY()) - (first.getVecY() * second.getVecX());
        this.len = -1;
    }

    /**
     * Normalizes this vector in place. The null vector is normalized to (1,0).
     */
    public void normalize() {

        double before;

        before = length();

        if (before < Double.MIN_NORMAL) {

            // Don't want to divide by that, so consider it zero.
            this.vecX = 1;
            this.vecY = 0;
            this.vecZ = 0;
        } else {
            this.vecX /= before;
            this.vecY /= before;
            this.vecZ /= before;
        }

        this.len = 1;
    }

    /**
     * Generates the string representation of the point.
     *
     * @return  the <code>String</code> representation
     */
    @Override public String toString() {

        StringBuilder str;

        str = new StringBuilder(50);

        str.append('[');
        str.append(this.vecX);
        str.append(", ");
        str.append(this.vecY);
        str.append(", ");
        str.append(this.vecZ);
        str.append(']');

        return str.toString();
    }
}
```

```
package com.srbenoit.geom;

import com.srbenoit.sparsearray.SparseArray;

/**
 * An array of vectors that supports addition and deletion of vectors, but will not change the
 * index of a vector once added. That is, deleting vectors makes the array sparse, and adding new
 * vectors may fill in gaps left by deleting vectors before appending to the end of the array.
 * Storage is allocated in blocks of fixed size as needed to add vectors.
 */
public class Vector3Array extends SparseArray<Vector3Int> {

    /**
     * Constructs a new <code>Vector3Array</code> with a default capacity.
     */
    public Vector3Array() {

        this(16);
    }

    /**
     * Constructs a new <code>Vector3Array</code> with capacity for a specified number of vectors.
     * Use this constructor if you know how many vectors will ultimately be added to the array.
     *
     * @param  initialSize  the initial capacity of array to allocate
     */
    public Vector3Array(final int initialSize) {

        super(Vector3Int.class, initialSize);
    }

    /**
     * Transforms the vectors in this list into another list using a given transformation matrix.
     * It is assumed that the target array contains the same number and arrangement of vectors as
     * this list.
     *
     * @param  target     the array into which to transform the vectors
     * @param  transform  the transformation matrix
     */
    public void transformInto(final Vector3Array target, final Transform3 transform) {

        int len;

        len = capacity();

        for (int i = 0; i < len; i++) {

            if (isFilled(i)) {
                transform.transformVec(get(i), target.get(i));
            }
        }
    }
}
```

```
package com.srbenoit.geom;

/**
 * An interface for objects that can be represented as 3-dimensional vectors.
 */
public interface Vector3Int {

    /**
     * Gets the X component of the vector.
     *
     * @return  the X component
     */
    double getVecX();

    /**
     * Sets the X component of the vector.
     *
     * @param  xComp  the X component
     */
    void setVecX(double xComp);

    /**
     * Gets the Y component of the vector.
     *
     * @return  the Y component
     */
    double getVecY();

    /**
     * Sets the Y component of the vector.
     *
     * @param  yComp  the Y component
```

```
 */
void setVecY(double yComp);

/**
 * Gets the Z component of the vector.
 *
 * @return  the Z component
 */
double getVecZ();

/**
 * Sets the Z component of the vector.
 *
 * @param  zComp  the Z component
 */
void setVecZ(double zComp);

/**
 * Sets the coordinates of the vector.
 *
 * @param  xComp  the x component
 * @param  yComp  the y component
 * @param  zComp  the z component
 */
void setVec(double xComp, double yComp, double zComp);

/**
 * Sets the coordinates of the vector from another vector.
 *
 * @param  source  the vector whose components are to be copied
 */
void setVec(Vector3Int source);

/**
 * Adds to the components of the vector.
 *
 * @param  xDelta  the change to the x component
 * @param  yDelta  the change to the y component
 * @param  zDelta  the change to the z component
 */
void addVec(double xDelta, double yDelta, double zDelta);

/**
 * Adds a vector to this vector.
 *
 * @param  vector  the vector to add to this vector
 */
void addVec(Vector3Int vector);

/**
 * Subtracts a vector from this vector.
 *
 * @param  vector  the vector to subtract from this vector
 */
void subVec(Vector3Int vector);

/**
 * Subtracts <code>vector2</code> from <code>vector1</code> and stores the result in this
 * vector.
 *
 * @param  vector1 the vector from which to subtract
 * @param  vector2 the vector to subtract
 */
void subVec(Vector3Int vector1, Vector3Int vector2);

/**
 * Adds a scaled version of a vector to this vector (this = this + scale tuple).
 *
 * @param  scale   the scalar value
 * @param  vector  the vector to be scaled then added
 */
void addVecScaled(double scale, Vector3Int vector);

/**
 * Sets this vector to the vector from <code>point1</code> to <code>point2</code> (this =
 * point2 - point1).
 *
 * @param  point1  the first point
 * @param  point2  the second point
 */
void vectorBetween(Point3Int point1, Point3Int point2);

/**
 * Negates this vector in place.
 */
void negateVec();
```

```
    /**
     * Scales this vector by a scalar factor.
     *
     * @param  scale  the scalar factor
     */
    void scaleVec(double scale);

    /**
     * Sets this vector to a scaled version of another vector.
     *
     * @param  scale  the scalar factor
     * @param  vec the vector to scale
     */
    void scaleVec(double scale, Vector3Int vec);

    /**
     * Gets the lazily computed length of the vector.
     *
     * @return  the length of the vector, or -1 if the length has not yet been computed
     */
    double lazyLength();

    /**
     * Gets the squared length of the vector.
     *
     * @return  the squared length of the vector
     */
    double lengthSquared();

    /**
     * Gets the length of the vector.
     *
     * @return  the length of the vector
     */
    double length();

    /**
     * Computes the dot product of this vector with another vector.
     *
     * @param   vector  the other vector
     * @return  the dot product
     */
    double dot(Vector3Int vector);

    /**
     * Computes the cross product of <code>first</code> and <code>second</code> and stores the
     * result in this vector.
     *
     * @param  first   the first tuple in the cross product
     * @param  second  the second tuple in the cross product
     */
    void cross(Vector3Int first, Vector3Int second);

    /**
     * Normalizes this vector in place. The null vector is normalized to (1,0).
     */
    void normalize();
}
```

E.1.3 Logging (com.srbenoit.log)

This package extends the Java logging facilities provided under java.util.logging with a custom formatter that presents a much more compact format for log messages, and provides new base classes for Object, Thread and JPanel that have a static Logger named LOG defined and configured with this formatter. By extending one of these classes, logging can be performed by invoking, for example, LOG.info("hello world");.

```
package com.srbenoit.log;
```

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;
import java.util.logging.Formatter;
import java.util.logging.Level;
import java.util.logging.LogRecord;

/**
 * A formatter to create a compact and well-aligned text output for log records.
 */
public class LogFormatter extends Formatter {

    /** the format string for dates used in log entries */
    private static final String DATE_FORMAT = "MM/dd HH:mm:ss.SSS ";

    /** the date of the log record */
    private final transient Date date;

    /** line termination string (\n or \r\n) */
    private final transient String crlf;

    /** formatter for dates */
    private final transient SimpleDateFormat dateFormat;

    /**
     * Constructs a new <code>LogFormatter</code>.
     */
    public LogFormatter() {

        super();

        String linefeed;

        this.dateFormat = new SimpleDateFormat(DATE_FORMAT, Locale.getDefault());
        this.date = new Date();

        linefeed = System.getProperty("line.separator");

        if (linefeed == null) {
            linefeed = "\r\n";
        }

        this.crlf = linefeed;
    }

    /**
     * Formats the given log record and returns the formatted string. The resulting formatted string
     * will include a localized and formated version of the <code>LogRecord</code>'s message field.
     * The <code>Formatter.formatMessage</code> convenience method can (optionally) be used to
     * localize and format the message field.
     *
     * @param   record  the log record to be formatted
     * @return  the formatted log record
     */
    @Override public String format(final LogRecord record) {

        StringBuilder builder;
        int lvl;

        builder = new StringBuilder(200);
        lvl = record.getLevel().intValue();

        // Make FINE/FINER behave like System.out.println, and FINEST like
        // System.out.println.
        if (lvl > Level.FINE.intValue()) {
            this.date.setTime(record.getMillis());
            builder.append(this.dateFormat.format(this.date));
        }

        if (Level.SEVERE.intValue() == lvl) {
            builder.append("S ");
        } else if (Level.WARNING.intValue() == lvl) {
            builder.append("W ");
        } else if (Level.INFO.intValue() == lvl) {
            builder.append("I ");
        } else if (Level.CONFIG.intValue() == lvl) {
            builder.append("C ");
        } else if (lvl > Level.FINE.intValue()) {
            builder.append("  ");
        }

        builder.append(formatMessage(record));

        if (lvl > Level.FINE.intValue()) {
            addSourceInfo(record, builder);
            addExceptionInfo(record, builder);
        }
```

```
        if (lvl > Level.FINEST.intValue()) {
            builder.append(this.crlf);
        }

        return builder.toString();
    }

    /**
     * Appends the source class and method information to the log output.
     *
     * @param  record   the log record
     * @param  builder  the <code>StringBuilder</code> to which to append the throwable information
     */
    private void addSourceInfo(final LogRecord record, final StringBuilder builder) {

        if (record.getSourceClassName() != null) {
            builder.append(" (");
            builder.append(record.getSourceClassName());

            if (record.getSourceMethodName() != null) {
                builder.append('.');
                builder.append(record.getSourceMethodName());
            }

            builder.append(')');
        }
    }

    /**
     * Appends the throwable information to the log output.
     *
     * @param  record   the log record
     * @param  builder  the <code>StringBuilder</code> to which to append the throwable information
     */
    private void addExceptionInfo(final LogRecord record, final StringBuilder builder) {

        Throwable thrown;
        StackTraceElement[] stack;

        thrown = record.getThrown();

        while (thrown != null) {
            builder.append(this.crlf);
            builder.append("----------------------");
            builder.append(thrown.getClass().getSimpleName());

            if (thrown.getLocalizedMessage() != null) {
                builder.append(": ");
                builder.append(thrown.getLocalizedMessage());
            }

            stack = thrown.getStackTrace();

            for (int i = 0; i < stack.length; i++) {
                builder.append(this.crlf);
                builder.append("------------------------");
                builder.append(stack[i].toString());
            }

            thrown = thrown.getCause();

            if (thrown != null) {
                builder.append(this.crlf);
                builder.append("----------------------CAUSED BY:");
            }
        }
    }
}

package com.srbenoit.log;

import java.util.logging.Logger;

/**
 * An object that includes a static logger to which diagnostic messages can be logged.
 */
public class LoggedObject {

    /** a log to which to write diagnostic messages */
    protected static final Logger LOG;

    static {
        LOG = LogMgr.getLogger();
    }

    /**
```

```
     * Constructs a new <code>LoggedObject</code>.
     */
    protected LoggedObject() {

        // No action
    }
}

package com.srbenoit.log;

import java.awt.LayoutManager;
import java.util.logging.Logger;
import javax.swing.JPanel;

/**
 * A panel that includes a static logger to which diagnostic messages can be logged.
 */
public class LoggedPanel extends JPanel {

    /** version number for serialization */
    private static final long serialVersionUID = -9154069273727941925L;

    /** a log to which to write diagnostic messages */
    protected static final Logger LOG;

    static {
        LOG = LogMgr.getLogger();
    }

    /**
     * Constructs a new <code>LoggedPanel</code>.
     */
    protected LoggedPanel() {

        super();
    }

    /**
     * Constructs a new <code>LoggedPanel</code> with the specified layout manager.
     *
     * @param  layout  the <code>LayoutManager</code> to use
     */
    protected LoggedPanel(final LayoutManager layout) {

        super(layout);
    }
}

package com.srbenoit.log;

import java.util.logging.Logger;

/**
 * A thread that includes a static logger to which diagnostic messages can be logged.
 */
public class LoggedThread extends Thread {

    /** a log to which to write diagnostic messages */
    protected static final Logger LOG;

    static {
        LOG = LogMgr.getLogger();
    }

    /**
     * Constructs a new <code>LoggedThread</code>.
     *
     * @param  threadName  the name of the thread
     */
    public LoggedThread(final String threadName) {

        super(threadName);
    }
}

package com.srbenoit.log;

import java.io.File;
import java.io.IOException;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.ResourceBundle;
import java.util.logging.FileHandler;
import java.util.logging.Handler;
import java.util.logging.LogRecord;
import java.util.logging.Logger;
```

```
/**
 * A custom log entry handler that writes log records to a series of files and also has the ability
 * to retain log entries for a period of time, and provide access to the retained entries.
 */
public class LogHandler extends Handler {

    /** the resource bundle containing localized strings */
    private final transient ResourceBundle res;

    /** a file handler used to write log files */
    private transient FileHandler fileHandler = null;

    /** a list of retained log records */
    private final transient List<LogRecord> records;

    /** the maximum number of (the most recent) log records to retain */
    private transient int maxRecsToRetain = 0;

    /**
     * Constructs a new <code>BekenHandler</code>.
     *
     * @param  resources  the resource bundle containing localized strings
     */
    public LogHandler(final ResourceBundle resources) {

        super();

        this.res = resources;
        this.records = new ArrayList<LogRecord>(20);
    }

    /**
     * Configures the handler based on logging properties. This may add a file handler if so
     * configured and the logger has no file handler already.
     *
     * @param  props  the properties from which to get log configuration
     * @param  log    a log to which to write diagnostic messages
     */
    public void configure(final LogProperties props, final Logger log) {

        String path;
        int limit;
        int count;
        boolean append;
        File file;
        String msg;

        if ((this.fileHandler == null) && (props != null)) {
            setLevel(props.getLogLevel());

            path = props.getLogFilePath();
            count = props.getLogFileCount();

            if ((path != null) && (count > 0)) {
                limit = props.getLogFileSizeLimit();
                append = props.isLogFileAppend();

                file = new File(path);
                file.mkdirs();

                try {
                    this.fileHandler = new FileHandler(path, limit, count, append);
                    this.fileHandler.setFormatter(new LogFormatter());
                } catch (IOException e) {
                    msg = this.res.getString(LogRes.CANT_OPEN_LOG);
                    log.warning(MessageFormat.format(msg, path, e.getLocalizedMessage()));
                } catch (IllegalArgumentException e) {
                    msg = this.res.getString(LogRes.BAD_PARAMS);
                    log.warning(MessageFormat.format(msg, e.getLocalizedMessage()));
                }

                msg = this.res.getString(LogRes.LOGGING_STARTED);
                log.info(MessageFormat.format(msg, props.getLogFilePath()));
            }
        }
    }

    /**
     * Determines whether the handler retains log entries or not, and if so, how many log entries
     * are retained.
     *
     * @param  maxToRetain  the maximum number of log entries to retain, or 0 to disable log entry
     *                      retention (the most recent log entries are retained and when this
     *                      retention level is reached, each new log message will cause the oldest
     *                      retained log message to be discarded)
     */
```

```
public void setRetainLogEntries(final int maxToRetain) {

    this.maxRecsToRetain = maxToRetain;

    if (maxToRetain == 0) {
        this.records.clear();
    } else {

        while (this.records.size() > maxToRetain) {
            this.records.remove(maxToRetain);
        }
    }
}

/**
 * Determines the number of retained log entries currently stored in the handler.
 *
 * @return  the number of log entries retained
 */
public int size() {

    return this.records.size();
}

/**
 * Clears all stored log entries, if any.
 */
public void clear() {

    this.records.clear();
}

/**
 * Gets a particular log record from the list of retained records.
 *
 * @param   index  the index of the record to retrieve
 * @return  the log record, or null if the index did not map to a valid log record
 */
public LogRecord get(final int index) {

    LogRecord rec;

    if ((index >= 0) && (index < this.records.size())) {
        rec = this.records.get(index);
    } else {
        rec = null;
    }

    return rec;
}

/**
 * Gets all retained log records.
 *
 * @return  the log records list, or null if records are not being retained
 */
public LogRecord[] getAll() {

    LogRecord[] array;

    array = new LogRecord[this.records.size()];
    this.records.toArray(array);

    return array;
}

/**
 * Closes the handler, releasing all stored resources.
 */
@Override public void close() {

    if (this.fileHandler != null) {
        this.fileHandler.close();
    }

    this.records.clear();
}

/**
 * Flushes any buffered output. Since this class does not output its log entries, this method
 * does nothing.
 */
@Override public void flush() {

    if (this.fileHandler != null) {
        this.fileHandler.flush();
    }
```

```
    }

    /**
     * Publishes a log record. This causes the record to be appended to the list of stored records
     * if this handler is currently configured to retain log entries. Otherwise, the record is
     * ignored.
     *
     * @param  record  the log record to publish
     */
    @Override public void publish(final LogRecord record) {

        if ((this.maxRecsToRetain > 0) && (record != null)) {
            this.records.add(record);

            while (this.records.size() > this.maxRecsToRetain) {
                this.records.remove(this.maxRecsToRetain);
            }
        }

        if (this.fileHandler != null) {
            this.fileHandler.publish(record);
        }
    }
}

package com.srbenoit.log;

import java.util.ResourceBundle;
import java.util.logging.ConsoleHandler;
import java.util.logging.Handler;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * A class with a static method to create a named <code>Logger</code> that has a special handler
 * installed to allow retention of and access to log records.
 */
public final class LogMgr {

    /** a log to which to write diagnostic messages */
    private static final Logger LOG;

    static {

        LogHandler handler;
        ResourceBundle res;

        LOG = Logger.getLogger("com.srbenoit");

        // This named logger may have already been created (and the handler
        // added), so check for an existing BekenHandler.
        if (findHandler(LOG) == null) {
            res = LogRes.getInstance(LOG);
            LOG.setLevel(Level.ALL);
            LOG.setUseParentHandlers(false);
            fixConsole(LOG);
            handler = new LogHandler(res);
            handler.setLevel(Level.ALL);
            LOG.addHandler(handler);
        }
    }

    /**
     * Private constructor to prevent instantiation.
     */
    private LogMgr() {

        // no action
    }

    /**
     * Get a <code>Logger</code> that has a handler that retains log entries installed.
     *
     * @return  the logger
     */
    public static Logger getLogger() {

        return LOG;
    }

    /**
     * Configures a logger based on logging properties. This may start file logging if so configured
     * and not already configured.
     *
     * @param  logger  the logger to be configured
     * @param  props   the properties that hold logging configuration
     */
```

```
public static void configure(final Logger logger, final LogProperties props) {

    LogHandler handler;

    logger.setLevel(props.getLogLevel());

    handler = findHandler(logger);

    if (handler != null) {
        handler.configure(props, logger);
    }
}

/**
 * Given a <code>Logger</code>, scan its list of installed loggers for a <code>
 * LogHandler</code>, and return that handler if found.
 *
 * @param   logger  the <code>Logger</code> to search
 * @return  the <code>LogHandler</code> if found, <code>null</code> if not
 */
public static LogHandler findHandler(final Logger logger) {

    Handler[] list;
    LogHandler handler = null;

    list = logger.getHandlers();

    for (int i = 0; i < list.length; i++) {

        if (list[i] instanceof LogHandler) {
            handler = (LogHandler) list[i];

            break;
        }
    }

    return handler;
}

/**
 * Given a <code>Logger</code>, scans its list of installed loggers for a <code>
 * ConsoleHandler</code>, and changes its output stream from <code>System.err</code> to <code>
 * System.out</code>.
 *
 * @param  logger  the <code>Logger</code> to search
 */
private static void fixConsole(final Logger logger) {

    SysOutConsoleHandler handler;
    Handler[] list;
    boolean hit = false;

    list = logger.getHandlers();

    for (int i = 0; i < list.length; i++) {

        if (list[i] instanceof ConsoleHandler) {
            logger.removeHandler(list[i]);
        } else if (list[i] instanceof SysOutConsoleHandler) {
            hit = true;

            break;
        }
    }

    if (!hit) {
        handler = new SysOutConsoleHandler();
        handler.setLevel(Level.ALL);
        handler.setFormatter(new LogFormatter());
        logger.addHandler(handler);
    }
}

/**
 * Given a <code>Logger</code>, scans its list of installed handlers for <code>
 * ConsoleHandler</code> or <code>SysOutConsoleHandler</code> and removes any such handlers it
 * finds.
 *
 * <p>This is intended for use in headless environments where attempting to log to a console can
 * cause output buffers for consoles to fill, locking up the program.
 *
 * @param  logger  the <code>Logger</code> to check
 */
public static void haltConsoleOutput(final Logger logger) {

    Handler[] list;
```

```
        list = logger.getHandlers();

        for (int i = 0; i < list.length; i++) {

            if (list[i] instanceof ConsoleHandler) {
                logger.removeHandler(list[i]);
            } else if (list[i] instanceof SysOutConsoleHandler) {
                logger.removeHandler(list[i]);
            }
        }
    }

    /**
     * Main method that creates a log and logs several messages to test and demonstrate logging.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        String classname;
        String methodname;
        String[] list;
        Exception cause;
        Exception exc;

        classname = "LogMgr";
        methodname = "main";
        list = new String[] { "item1", "item2", "item3" };
        cause = new IllegalArgumentException("Cause exception");
        exc = new SecurityException("Some exception", cause);

        LOG.config("Config message");
        LOG.entering(classname, methodname);
        LOG.entering(classname, methodname, list[0]);
        LOG.entering(classname, methodname, list);
        LOG.exiting(classname, methodname);
        LOG.exiting(classname, methodname, list[0]);
        LOG.fine("Fine message");
        LOG.finer("Finer message");
        LOG.finest("Finest message");
        LOG.info("Info message");
        LOG.severe("Severe message");
        LOG.throwing(classname, methodname, exc);
        LOG.warning("Warning message");
        LOG.log(Level.CONFIG, "Log message 1");
        LOG.log(Level.INFO, "Log message 2", list[0]);
        LOG.log(Level.WARNING, "Log message 3", list);
        LOG.log(Level.SEVERE, "Log message 4", exc);
    }
}

package com.srbenoit.log;

import java.io.File;
import java.text.MessageFormat;
import java.util.Properties;
import java.util.ResourceBundle;
import java.util.logging.Level;
import java.util.logging.Logger;
import com.srbenoit.util.ResourceLoader;

/**
 * Properties used by the logging classes.
 */
public final class LogProperties {

    /** the key of the log level property */
    public static final String LEVEL_TAG = "log-level";

    /** the key of the log file path property */
    public static final String FILE_PATH_TAG = "log-file-path";

    /** the key of the log file size limit property */
    public static final String FILE_LIMIT_TAG = "log-file-size-limit";

    /** the key of the log file count property */
    public static final String FILE_COUNT_TAG = "log-file-count";

    /** the key of the log file append property */
    public static final String APPEND_TAG = "log-file-append";

    /** the default file limit. */
    private static final int DEF_FILE_LIMIT = 100000;

    /** the default file count. */
    private static final int DEF_FILE_COUNT = 1;
```

```
/** the configured logging level */
private transient Level logLevel;

/** the configured log file path */
private transient final String logFilePath;

/** the configured log file size limit */
private transient int logFileSizeLimit;

/** the configured log file count */
private transient int logFileCount;

/** true to append to existing files; false to overwrite */
private transient boolean logFileAppend;

/**
 * Private constructor so the factory method is always used to load.
 *
 * @param  configDir  the configuration data directory
 * @param  res        the resource bundle containing localized strings
 * @param  props      the loaded properties object
 * @param  log        a log to which to write diagnostic messages
 */
private LogProperties(final File configDir, final ResourceBundle res, final Properties props,
    final Logger log) {

    String str;

    configureLogLevel(res, props, log);
    configureFileLimit(res, props, log);
    configureFileCount(res, props, log);
    configureAppend(res, props, log);

    str = props.getProperty(FILE_PATH_TAG);

    if (configDir == null) {
        this.logFilePath = str;
    } else if (str == null) {
        this.logFilePath = null;
    } else {
        this.logFilePath = str.replace("%dir", configDir.getAbsolutePath());
    }
}

/**
 * Configures the log level from the properties.
 *
 * @param  res    the resource bundle containing localized strings
 * @param  props  the loaded properties object
 * @param  log    a log to which to write diagnostic messages
 */
private void configureLogLevel(final ResourceBundle res, final Properties props,
    final Logger log) {

    String str;
    String msg;

    str = props.getProperty(LEVEL_TAG);

    if (str == null) {
        msg = res.getString(LogRes.MISSING_PARAM);
        log.warning(MessageFormat.format(msg, LEVEL_TAG, "ALL"));
        this.logLevel = Level.ALL;
    } else {

        try {
            this.logLevel = Level.parse(str);
        } catch (IllegalArgumentException e) {
            msg = res.getString(LogRes.BAD_PARAM);
            log.warning(MessageFormat.format(msg, LEVEL_TAG, str, "ALL"));
            this.logLevel = Level.ALL;
        }
    }
}

/**
 * Configures the log file limit from the properties.
 *
 * @param  res    the resource bundle containing localized strings
 * @param  props  the loaded properties object
 * @param  log    a log to which to write diagnostic messages
 */
private void configureFileLimit(final ResourceBundle res, final Properties props,
    final Logger log) {

    String str;
    String msg;
```

```
        str = props.getProperty(FILE_LIMIT_TAG);

        if (str == null) {
            msg = res.getString(LogRes.MISSING_PARAM);
            log.warning(MessageFormat.format(msg, FILE_LIMIT_TAG,
                    Integer.toString(DEF_FILE_LIMIT)));
            this.logFileSizeLimit = DEF_FILE_LIMIT;
        } else {

            try {
                this.logFileSizeLimit = Integer.parseInt(str);
            } catch (NumberFormatException e) {
                msg = res.getString(LogRes.BAD_PARAM);
                log.warning(MessageFormat.format(msg, FILE_LIMIT_TAG, str,
                        Integer.toString(DEF_FILE_LIMIT)));
                this.logFileSizeLimit = DEF_FILE_LIMIT;
            }
        }
    }

    /**
     * Configures the log file limit from the properties.
     *
     * @param  res     the resource bundle containing localized strings
     * @param  props   the loaded properties object
     * @param  log     a log to which to write diagnostic messages
     */
    private void configureFileCount(final ResourceBundle res, final Properties props,
        final Logger log) {

        String str;
        String msg;

        str = props.getProperty(FILE_COUNT_TAG);

        if (str == null) {
            msg = res.getString(LogRes.MISSING_PARAM);
            log.warning(MessageFormat.format(msg, FILE_COUNT_TAG,
                    Integer.toString(DEF_FILE_COUNT)));
            this.logFileCount = DEF_FILE_COUNT;
        } else {

            try {
                this.logFileCount = Integer.parseInt(str);
            } catch (NumberFormatException e) {
                msg = res.getString(LogRes.BAD_PARAM);
                log.warning(MessageFormat.format(msg, FILE_LIMIT_TAG, str,
                        Integer.toString(DEF_FILE_LIMIT)));
                this.logFileCount = DEF_FILE_COUNT;
            }
        }
    }

    /**
     * Configures the log file limit from the properties.
     *
     * @param  res     the resource bundle containing localized strings
     * @param  props   the loaded properties object
     * @param  log     a log to which to write diagnostic messages
     */
    private void configureAppend(final ResourceBundle res, final Properties props,
        final Logger log) {

        String str;
        String msg;

        str = props.getProperty(APPEND_TAG);

        if (str == null) {
            msg = res.getString(LogRes.MISSING_PARAM);
            log.warning(MessageFormat.format(msg, APPEND_TAG, "false"));
            this.logFileAppend = false;
        } else {

            try {
                this.logFileAppend = Boolean.parseBoolean(str);
            } catch (NumberFormatException e) {
                msg = res.getString(LogRes.BAD_PARAM);
                log.warning(MessageFormat.format(msg, APPEND_TAG, str, "false"));
                this.logFileAppend = false;
            }
        }
    }

    /**
     * Gets the log level.
```

```
     *
     * @return  the log level
     */
    public Level getLogLevel() {

        return this.logLevel;
    }

    /**
     * Gets the log file path.
     *
     * @return  the log file path
     */
    public String getLogFilePath() {

        return this.logFilePath;
    }

    /**
     * Gets the log file size limit.
     *
     * @return  the log file size limit
     */
    public int getLogFileSizeLimit() {

        return this.logFileSizeLimit;
    }

    /**
     * Gets the log file count.
     *
     * @return  the log file count
     */
    public int getLogFileCount() {

        return this.logFileCount;
    }

    /**
     * Gets the log file append flag.
     *
     * @return  <code>true</code> if starting logging will append to an existing log file, <code>
     *          false</code> if it will overwrite
     */
    public boolean isLogFileAppend() {

        return this.logFileAppend;
    }

    /**
     * Loads the properties from the configuration directory, or supplies a set of default
     * properties if the properties file cannot be loaded from that location.
     *
     * @param   configDir  the configuration data directory
     * @param   base       the base name (without a language extension) of the properties file to
     *                     load
     * @param   res        the resource bundle containing localized strings
     * @param   log        a log to which to write diagnostic messages
     * @return  the properties (either what we could load, or a set of defaults), but never <code>
     *          null</code>
     */
    public static LogProperties load(final File configDir, final Class<?> base,
        final ResourceBundle res, final Logger log) {

        Properties props;
        String msg;

        if (configDir == null) {
            props = ResourceLoader.loadProperties(base, base.getSimpleName());
        } else {
            props = ResourceLoader.loadProperties(configDir, base.getSimpleName());
        }

        if (props == null) {
            msg = res.getString(LogRes.CANT_LOAD_PROPS);
            log.warning(MessageFormat.format(msg, configDir));
            props = new Properties();
        }

        return new LogProperties(configDir, res, props, log);
    }
}

package com.srbenoit.log;

import java.util.ListResourceBundle;
import java.util.Locale;
```

```
import java.util.ResourceBundle;
import java.util.logging.Logger;

/**
 * Resources used by the logging classes for localization.
 */
public class LogRes extends ListResourceBundle {

    /** message tag */
    public static final String CANT_LOAD_RES = "01";

    /** message tag */
    public static final String CANT_LOAD_PROPS = "02";

    /** message tag */
    public static final String LOGGING_STARTED = "03";

    /** message tag */
    public static final String CANT_OPEN_LOG = "04";

    /** message tag */
    public static final String BAD_PARAMS = "05";

    /** message tag */
    public static final String MISSING_PARAM = "06";

    /** message tag */
    public static final String BAD_PARAM = "07";

    /** message tag */
    public static final String CANT_CREATE_DIR = "08";

    /**
     * Constructs a new <code>ResourceBundle</code> with the localized resources, or supplies a
     * default set of resources if unable to load.
     *
     * @param   log  a log to which to write diagnostic messages
     * @return  the loaded resource bundle
     */
    public static ResourceBundle getInstance(final Logger log) {

        ResourceBundle res;

        // Load localized resources
        res = ResourceBundle.getBundle(LogRes.class.getName(), Locale.getDefault());

        if (res == null) {
            res = new LogRes();
            log.warning(res.getString(LogRes.CANT_LOAD_RES));
        }

        return res;
    }

    /**
     * Returns an array in which each item is a two-<code>Object</code> array, in which the first
     * element is the key, which must be a <code>String</code>, and the second element is the value
     * associated with that key.
     *
     * @return  the content array
     */
    @Override protected Object[][] getContents() {

        Object[][] contents;

        contents = new Object[][] {

                // Log message when resource bundle is not found and default
                // values are being used in place of loaded properties
                { CANT_LOAD_RES, "Unable to load localized resources -- using defaults" },

                // Log message when there is an error loading configuration
                // properties MessageFormat replaces {0} by the directory path
                // from which server attempted to load properties
                { CANT_LOAD_PROPS, "Unable to load properties from \"{0}\"; using defaults." },

                // Log message when file logging is started.
                // MessageFormat replaces {0} with the log file path
                { LOGGING_STARTED, "File logging started in {0}" },

                // Log message when there is an error opening the log file for
                // writing. MessageFormat replaces {0} with the log file path
                // and {1} with the message from the exception that occurred
                // when opening the file
                { CANT_OPEN_LOG, "Unable to open log file \"{0}\": {1}" },

                // Log message when there is an error opening the log file for
```

```
                // writing. MessageFormat replaces {0} with the message of the
                // exception generated by the bad parameter
                { BAD_PARAMS, "Invalid logging parameters: {0}" },

                // Log message when a log properties file is missing a
                // specification for some parameter MessageFormat replaces {0}
                // with the parameter that is missing and {1} with the value
                // that was used as a default
                { MISSING_PARAM, "Missing \"{0}\" value, using \"{1}\"" },

                // Log message when a parameter specified in the properties is
                // invalid MessageFormat replaces {0} with the parameter tag,
                // {1} with the value that was found in the properties file,
                // and {2} with the value that was used as a default
                { BAD_PARAM, "Invalid \"{0}\" ({1}), using \"{2}\"" },

                // Log message when there is an error creating the log
                // directory. MessageFormat replaces {0} with the log file
                // path.
                { CANT_CREATE_DIR, "Unable to create log directory \"{0}\"" },
            };

        return contents;
    }
}

package com.srbenoit.log;

/**
 * Interface implemented by messages.
 */
public interface MessageInt {

    /* Empty */
}

package com.srbenoit.log;

import java.util.ArrayList;
import java.util.List;

/**
 * A list of messages generated by some process that spans classes. Messages are intended for users,
 * not for system diagnostic.
 */
public class MessageList extends LoggedObject {

    /** the list of messages */
    private final transient List<MessageInt> messages;

    /**
     * Constructs a new <code>MessageList</code>.
     */
    public MessageList() {

        super();

        this.messages = new ArrayList<MessageInt>(20);
    }

    /**
     * Gets the number of messages in the list.
     *
     * @return  the number of messages
     */
    public int size() {

        return this.messages.size();
    }

    /**
     * Gets a particular message.
     *
     * @param   index  the index of the message to get
     * @return  the message
     */
    public MessageInt getMessage(final int index) {

        return this.messages.get(index);
    }

    /**
     * Adds a message to the list and logs it at the <code>FINE</code> log level.
     *
     * @param  message  the message to add
     */
    public void add(final MessageInt message) {
```

```
        LOG.fine(message.toString());
        this.messages.add(message);
    }

    /**
     * Generates the string representation of the message list, which contains all of the message
     * strings separated by carriage return / newlines;
     *
     * @return  the string representation
     */
    @Override public String toString() {

        StringBuilder builder;

        builder = new StringBuilder(200);

        for (MessageInt msg : this.messages) {
            builder.append(msg.toString());
            builder.append("\r\n");
        }

        return builder.toString();
    }
}

package com.srbenoit.log;

/**
 * A message that indicates an error in parsing. It includes the range in the source data where the
 * parse error occurred.
 */
public class ParseError implements MessageInt {

    /** the error message */
    private final transient String errorMsg;

    /** the start position */
    private final transient int start;

    /** the end position */
    private final transient int end;

    /**
     * Constructs a new <code>ParseError</code>.
     *
     * @param  msg  the message
     * @param  pos  the error position
     */
    public ParseError(final String msg, final int pos) {

        this.errorMsg = msg;
        this.start = pos;
        this.end = pos;
    }

    /**
     * Constructs a new <code>ParseError</code>.
     *
     * @param  msg       the message
     * @param  startPos  the start position
     * @param  endPos    the end position
     */
    public ParseError(final String msg, final int startPos, final int endPos) {

        this.errorMsg = msg;
        this.start = startPos;
        this.end = endPos;
    }

    /**
     * Gets the error message.
     *
     * @return  the error message
     */
    public String getMessage() {

        return this.errorMsg;
    }

    /**
     * Gets the start position of the parse error.
     *
     * @return  the start position
     */
    public int getStart() {
```

```
        return this.start;
    }

    /**
     * Gets the end position of the parse error.
     *
     * @return  the end position
     */
    public int getEnd() {

        return this.end;
    }

    /**
     * Generates the string representation of the message.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        StringBuilder builder;

        builder = new StringBuilder(50);

        builder.append(this.errorMsg);
        builder.append('(');
        builder.append(this.start);

        if (this.end != this.start) {
            builder.append('-');
            builder.append(this.end);
        }

        builder.append(')');

        return builder.toString();
    }
}

package com.srbenoit.log;

import java.util.logging.Filter;
import java.util.logging.Formatter;
import java.util.logging.Level;
import java.util.logging.LogManager;
import java.util.logging.LogRecord;
import java.util.logging.SimpleFormatter;
import java.util.logging.StreamHandler;

/**
 * A handler to direct output to <code>System.out</code>. The source code for this class is copied
 * from the Java SDK source code for <code>ConsoleHandler</code>, with the exception that it writes
 * to <code>System.out</code> rather than <code>System.err</code>.
 */
public final class SysOutConsoleHandler extends StreamHandler {

    /**
     * Configures the handler.
     */
    private void configure() {

        LogManager manager;
        String cname;
        String val;
        Class<?> clz;

        manager = LogManager.getLogManager();
        cname = getClass().getName();

        val = manager.getProperty(cname + ".level");

        if (val == null) {
            setLevel(Level.INFO);
        } else {
            setLevel(Level.parse(val.trim()));
        }

        val = manager.getProperty(cname + ".filter");

        if (val == null) {
            setFilter(null);
        } else {

            try {
                clz = ClassLoader.getSystemClassLoader().loadClass(val);
                setFilter((Filter) clz.newInstance());
            } catch (Exception e) {
```

```
                setFilter(null);
            }
        }

        val = manager.getProperty(cname + ".formatter");

        if (val != null) {

            try {
                clz = ClassLoader.getSystemClassLoader().loadClass(val);
                setFormatter((Formatter) clz.newInstance());
            } catch (Exception ex1) {
                publish(new LogRecord(Level.SEVERE, "Failed to load formatter class"));
            }
        }

        try {
            val = manager.getProperty(cname + ".encoding");

            if (val == null) {
                setEncoding(null);
            } else {
                setEncoding(val.trim());
            }
        } catch (Exception ex1) {

            try {
                setEncoding(null);
            } catch (Exception ex2) {
                publish(new LogRecord(Level.SEVERE, "Failed to set handler encoding"));
            }
        }
    }

    /**
     * Constructs a new <code>SysOutConsoleHandler</code>.
     */
    public SysOutConsoleHandler() {

        super(System.out, new SimpleFormatter());

        configure();
    }

    /**
     * Publishes a <code>LogRecord</code>.
     *
     * <p>The logging request was made initially to a <code>Logger</code> object, which initialized
     * the <code>LogRecord</code> and forwarded it here.
     *
     * <p>
     *
     * @param  record  description of the log event (a <code>null</code> record is silently ignored
     *                 and is not published)
     */
    @Override public synchronized void publish(final LogRecord record) { // NOPMD

        super.publish(record);
        flush();
    }

    /**
     * Overrides <code>StreamHandler.close</code> to do a flush but not to close the output stream.
     * That is, we do <b>not</b> close <code>System.out</code>.
     */
    @Override public synchronized void close() { // NOPMD

        flush();
    }
}
```

E.1.4 Pooled Object Management (com.srbenoit.pool)

This package provides a generic pool manager and a base class that pooled objects extend. Pools are thread-safe, meaning objects can be safely checked out and checked

back in to pools by multiple threads.

```
package com.srbenoit.pool;

import com.srbenoit.log.LoggedObject;

/**
 * The base class for objects that can be managed by a pool.
 */
public abstract class AbstractPoolObject extends LoggedObject implements Cloneable {

    /** object on which to synchronize access to this object's data */
    protected final transient Object synch;

    /** the pool from which this object was checked out */
    private transient Pool<? extends AbstractPoolObject> fromPool;

    /**
     * Constructs a new <code>AbstractPoolObject</code>.
     */
    public AbstractPoolObject() {

        super();
        this.synch = new Object();
    }

    /**
     * Sets the pool that this object was checked out from.
     *
     * @param  pool  the pool (<code>null</code> if this object was not checked out from a pool)
     */
    public void setFromPool(final Pool<? extends AbstractPoolObject> pool) {

        this.fromPool = pool;
    }

    /**
     * Gets the pool that this object was checked out from.
     *
     * @return  the pool (<code>null</code> if this object was not checked out from a pool)
     */
    public Pool<? extends AbstractPoolObject> getFromPool() {

        return this.fromPool;
    }

    /**
     * Resets this object to a virgin state (used before the object is returned to a pool). This
     * method must reset the object to a state that is indistinguishable from a newly created
     * object.
     */
    public abstract void toVirginState();

    /**
     * Creates a deep copy of the object - used to create new objects when the pool is empty. This
     * method should be implemented using <code>clone</code> to be more efficient than object
     * construction.
     *
     * @return  the copy, which will be in the same state as a newly created object
     */
    public abstract AbstractPoolObject copy();

    /**
     * Informs the pool object that it is being released for garbage collection and should free any
     * internal resources. The object may not be reused after this method is called.
     */
    public abstract void die();

    /**
     * Returns the object to the pool from which it was checked out. This calls <code>
     * checkIn</code> on the pool.
     */
    public void returnToPool() {

        this.fromPool.checkIn(this);
    }
}

package com.srbenoit.pool;

import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * A pool of objects that can be checked out for use, then checked back in when no longer needed.
 * Objects managed by the pool must be subclasses of <code>AbstractPoolObject</code>.
 *
```

```
 * <p>All methods in this class are thread-safe. A request to check out a new object when there are
 * no more available in the pool will result in a new object being created by cloning a prototype
 * object provided in the pool constructor. Therefore, every subclass of <code>
 * AbstractPoolObject</code> must provide a clone method that creates a completely independent
 * copy.
 *
 * <p>If the number of objects in the pool exceeds some threshold amount, some will be released to
 * be garbage collected to bring the total back down to that upper limit.
 *
 * @param  <E>  the type of object this pool manages
 */
public final class Pool<E extends AbstractPoolObject> extends LoggedObject {

    /** a prototype object that we can clone to make new ones */
    private final transient E prototype;

    /** an object on which to synchronize access to this object's data */
    private final transient Object synch;

    /** the maximum size of the pool */
    private final transient int maxPoolSize;

    /** the total number of objects in existence */
    private transient int total;

    /** the total number of available objects */
    private transient int available;

    /** the pool of objects */
    private transient AbstractPoolObject[] objectPool;

    /**
     * Constructs a new <code>Pool</code>.
     *
     * @param  proto         a prototype object that we can clone to make new ones
     * @param  initialSize   the initial number of objects created and placed in the pool
     * @param  maxSize       the maximum number of objects that the pool will store in an available
     *                       state (checked out objects do not count against this maximum)
     */
    public Pool(final E proto, final int initialSize, final int maxSize) {

        super();

        assert (proto != null);
        assert (initialSize >= 0);
        assert (maxSize >= initialSize);

        this.synch = new Object();
        this.prototype = proto;

        // Ensure we're working with a ground-state object as our prototype
        this.prototype.toVirginState();
        this.prototype.setFromPool(null);

        this.maxPoolSize = maxSize;

        this.objectPool = new AbstractPoolObject[initialSize];

        for (int i = 0; i < initialSize; i++) {
            this.objectPool[i] = proto.copy();
        }

        this.total = initialSize;
        this.available = initialSize;
    }

    /**
     * Gets the total number of objects this pool has created but not destroyed.
     *
     * @return  the total number of objects
     */
    public int getTotal() {

        synchronized (this.synch) {
            return this.total;
        }
    }

    /**
     * Gets the number of objects currently available in this pool.
     *
     * @return  the available number of objects
     */
    public int getAvailable() {

        synchronized (this.synch) {
            return this.available;
```

```
    }
}

/**
 * Checks out an object, creating a new one if the pool is empty.
 *
 * @return  the checked out object
 */
@SuppressWarnings("unchecked")
public E checkOut() {

    AbstractPoolObject object;

    synchronized (this.synch) {

        if (this.available > 0) {
            this.available--;
            object = this.objectPool[this.available];
            this.objectPool[this.available] = null;
        } else {
            object = this.prototype.copy();
            this.total++;
        }
    }

    assert object.getFromPool() == null;
    object.setFromPool(this);

    return (E) object;
}

/**
 * Checks in a object.
 *
 * @param  obj  the object to check in
 */
public void checkIn(final AbstractPoolObject obj) {

    int size;
    AbstractPoolObject[] newPool;

    assert obj.getFromPool() == this;
    obj.setFromPool(null);

    synchronized (this.synch) {

        if (this.available == this.maxPoolSize) {

            // Don't put back in pool - allow garbage collection
            obj.die();
            this.total--;
        } else {
            obj.toVirginState();

            if (this.available == this.objectPool.length) {
                size = this.objectPool.length << 1;

                if (size > this.maxPoolSize) {
                    size = this.maxPoolSize;
                }

                newPool = new AbstractPoolObject[size];
                System.arraycopy(this.objectPool, 0, newPool, 0, this.available);
                this.objectPool = newPool;
            }

            this.objectPool[this.available] = obj;
            this.available++;
        }
    }
}

/**
 * Instructs the pool to release all pooled objects for garbage collection, then deallocate its
 * internal objects. The pool may not be reused after this method is called.
 */
public void die() {

    synchronized (this.synch) {

        for (int i = 0; i < this.available; i++) {
            this.objectPool[i].die();
            this.objectPool[i] = null;
        }

        if (this.total > this.available) {
            LOG.log(Level.WARNING,
```

```
                    "Pool of {0} terminated while {1} objects were still checked out",
                    new Object[] {
                        this.prototype.getClass().getName(), this.total - this.available
                    });
            }

            this.prototype.die();
            this.objectPool = null;
            this.available = 0;
        }
    }
}
```

E.1.5 High performance sparse arrays (com.srbenoit.sparsearray)

This package provides classes to manage sparse arrays in which objects that are added retain their array position over their entire lifetime within the array (allowing them to be referenced by index), but the array can become sparse as objects are removed. The arrays track the lowest open position in the array and fill that position first when new elements are added. These arrays are intended to support efficient management of large numbers of objects (like points and vectors, for example), without doing non-essential copying or moving of data.

```
package com.srbenoit.sparsearray;

import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.BitSet;
import java.util.Iterator;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * An array of objects that supports addition and deletion of objects, but will not change the
 * index of a object once added. That is, deleting objects makes the array sparse, and adding new
 * objects may fill in gaps left by deleting objects before appending to the end of the array.
 * Storage is allocated in blocks of fixed size as needed to add objects.
 *
 * @param <E> the type of object this array manages
 */
public class SparseArray<E> extends LoggedObject implements Iterable<E> {

    /** number of objects in an allocated block */
    public static final int BLOCK_SIZE = 64;

    /** the class of objects this array manages */
    private final Class<E> objClass;

    /** index of first available position in array */
    private transient int firstAvailable;

    /** total number of objects in the array */
    private transient int numObjects;

    /** array of objects */
    private transient E[] objects;

    /** listeners registered to be notified with the array structure changes */
    private final List<SparseArrayListener> listeners;

    /**
     * Constructs a new <code>SparseArray</code> with a default capacity.
     *
```

```
     * @param  clazz  the class of object this array will manage
     */
    public SparseArray(final Class<E> clazz) {

        this(clazz, BLOCK_SIZE);
    }

    /**
     * Constructs a new <code>SparseArray</code> with capacity for a specified number of objects.
     * Use this constructor if you know how many objects will ultimately be added to the array.
     *
     * @param  clazz        the class of object this array will manage
     * @param  initialSize  the initial capacity of array to allocate
     */
    @SuppressWarnings({ "unchecked" })
    public SparseArray(final Class<E> clazz, final int initialSize) {

        this.firstAvailable = 0;
        this.numObjects = 0;
        this.objClass = clazz;
        this.objects = (E[]) Array.newInstance(clazz, initialSize);

        this.listeners = new ArrayList<SparseArrayListener>(1);
    }

    /**
     * Adds a listener to the list of those registered to be notified when the sparse array
     * structure changes.
     *
     * @param  listener  the new listener to register
     */
    public void addListener(final SparseArrayListener listener) {

        synchronized (this.listeners) {

            if (!this.listeners.contains(listener)) {
                this.listeners.add(listener);
            }
        }
    }

    /**
     * Removes a listener from the list of those registered to be notified when the sparse array
     * structure changes.
     *
     * @param  listener  the listener to remove
     */
    public void removeListener(final SparseArrayListener listener) {

        synchronized (this.listeners) {

            this.listeners.remove(listener);
        }
    }

    /**
     * Gets the length of the allocated array.
     *
     * @return  the length of the array (includes empty positions)
     */
    public int capacity() {

        return this.objects.length;
    }

    /**
     * Sets the capacity of this array, allocating new objects if needed. If the current array is
     * already large enough to accommodate the request, no changes are made (that is, the capacity
     * after this request may be larger than the requested capacity).
     *
     * @param  newCap  the new capacity
     */
    @SuppressWarnings("unchecked")
    public void setCapacity(final int newCap) {

        int len;
        E[] newArray;

        len = this.objects.length;

        if (newCap > len) {
            newArray = (E[]) Array.newInstance(this.objClass, newCap);
            System.arraycopy(this.objects, 0, newArray, 0, len);
            this.objects = newArray;
        }
    }
```

```
/**
 * Gets the number of objects in this list.
 *
 * @return  the number of objects in the array
 */
public int size() {

    return this.numObjects;
}

/**
 * Tests whether the array contains an object at a particular index.
 *
 * @param   index  the index to test
 * @return  <code>true</code> if the index is filled; <code>false</code> if not
 */
public boolean isFilled(final int index) {

    return this.objects[index] != null;
}

/**
 * Gets the index of the next filled index after (or including) a given starting index.
 *
 * @param   index  the starting index
 * @return  the next filled index, or -1 if no indices are filled from the start index on
 */
public int nextFilled(final int index) {

    int result;

    result = -1;

    for (int i = index; i < this.objects.length; i++) {

        if (this.objects[i] != null) {
            result = i;

            break;
        }
    }

    return result;
}

/**
 * Removes an object from the array.
 *
 * @param   index  the index of the object to remove
 * @return  <code>true</code> if an object was present at the requested index and was removed;
 *          <code>false</code> otherwise
 */
public E remove(final int index) {

    E obj;

    if (this.objects[index] != null) {
        obj = this.objects[index];
        this.objects[index] = null;
        this.numObjects--;

        if (index < this.firstAvailable) {
            this.firstAvailable = index;
        }
    } else {
        obj = null;
    }

    return obj;
}

/**
 * Adds an object to the array, increasing allocated storage if needed.
 *
 * @param   object  the object to add
 * @return  the index of the newly added object in the tuple array
 */
@SuppressWarnings("unchecked")
public int add(final E object) {

    int len;
    int index;
    boolean notify;

    // If we need to allocate more space, do it
    len = this.objects.length;
```

```
        if (this.numObjects == len) {
            setCapacity(len + BLOCK_SIZE);
            index = len;
            this.firstAvailable = len + 1;

            if (this.objects[this.firstAvailable] != null) {
                LOG.log(Level.WARNING, "After expanding sparse array, next item filled",
                    new Exception());
            }

            notify = true;
        } else {
            index = this.firstAvailable;

            this.firstAvailable = len;

            for (int i = index + 1; i < len; i++) {

                if (this.objects[i] == null) {
                    this.firstAvailable = i;

                    break;
                }
            }

            notify = false;
        }

        if (this.objects[index] != null) {
            LOG.log(Level.WARNING,
                "SparseArray of len " + this.objects.length + " stomping entry at " + index,
                new Exception());
        }

        this.objects[index] = object;
        this.numObjects++;

        if (notify) {

            synchronized (this.listeners) {

                for (SparseArrayListener list : this.listeners) {
                    list.capacityChanged(this, capacity());
                }
            }
        }

        return index;
    }

    /**
     * Attempts to add a tuple at a particular index.
     *
     * @param   object  the tuple to add
     * @param   index   the index at which to add the tuple
     * @throws  SparseArrayException  if the supplied index is not valid (either the array does not
     *                                contain the index, or there is already a tuple at the index)
     */
    public void add(final E object, final int index) throws SparseArrayException {

        if (this.objects.length > index) {

            if (this.objects[index] != null) {
                throw new SparseArrayException("Index " + index + " already occupied");
            }

            this.objects[index] = object;
            this.numObjects++;
        } else {
            throw new SparseArrayException("Index " + index + " out of bounds");
        }
    }

    /**
     * Gets an object from the array.
     *
     * @param   index  the index of the object to get
     * @return  the object
     */
    public E get(final int index) {

        return this.objects[index];
    }

    /**
     * Gets an iterator that will iterate over the filled elements in the sparse array.
     *
```

```
     * @return  the iterator
     */
    public Iterator<E> iterator() {

        return new SparseArrayIterator<E>(this);
    }
}

package com.srbenoit.sparsearray;

/**
 * An exception thrown by sparse arrays when invalid operations are attempted.
 */
public final class SparseArrayException extends RuntimeException {

    /** version number for serialization */
    private static final long serialVersionUID = -3949534910942884478L;

    /**
     * Constructs a new <code>SparseArrayException</code> with <code>null</code> as its detail
     * message.
     */
    public SparseArrayException() {

        super();
    }

    /**
     * Constructs a new <code>SparseArrayException</code> with the specified detail message.
     *
     * @param  message  the detail message
     */
    public SparseArrayException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>SparseArrayException</code> with the specified detail message and
     * cause.
     *
     * <p>Note that the detail message associated with <code>cause</code> is <i>not</i>
     * automatically incorporated in this runtime exception's detail message.
     *
     * @param  message  the detail message
     * @param  cause    the cause
     */
    public SparseArrayException(final String message, final Throwable cause) {

        super(message, cause);
    }

    /**
     * Constructs a new <code>SparseArrayException</code> with the specified cause and a detail
     * message of <code>(cause==null ? null : cause.toString())</code> (which typically contains the
     * class and detail message of <code>cause</code> ).
     *
     * @param  cause  the cause
     */
    public SparseArrayException(final Throwable cause) {

        super(cause);
    }
}

package com.srbenoit.sparsearray;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * An iterator that will iterate over a sparse array's elements.
 *
 * @param  <E>  the data type that the sparse array contains
 */
public class SparseArrayIterator<E> implements Iterator<E> {

    /** the source array being iterated */
    private final transient SparseArray<E> source;

    /** the index of the current element, -1 if finished or not yet started */
    private transient int currentElement;

    /** the index of the next element to be returned, -1 if finished */
    private transient int nextElement;

    /**
```

```
     * Constructs a new <code>SparseArrayIterator</code>.
     *
     * @param  sourceArray  the source array being iterated
     */
    public SparseArrayIterator(final SparseArray<E> sourceArray) {

        this.source = sourceArray;
        this.nextElement = sourceArray.nextFilled(0);
        this.currentElement = -1;
    }

    /**
     * Returns <code>true</code> if the iteration has more elements. In other words, returns <code>
     * true</code> if <code>next</code> would return an element rather than throwing an exception.
     *
     * @return  <code>true</code> if the iteration has more elements; <code>false</code> otherwise
     */
    public boolean hasNext() {

        return this.nextElement != -1;
    }

    /**
     * Returns the next element in the iteration.
     *
     * @return  the next element in the iteration
     * @throws  NoSuchElementException  if the iteration has no more elements
     */
    public E next() throws NoSuchElementException {

        E result;

        if (this.nextElement == -1) {
            throw new NoSuchElementException();
        }

        result = this.source.get(this.nextElement);
        this.currentElement = this.nextElement;
        this.nextElement = this.source.nextFilled(this.currentElement + 1);

        return result;
    }

    /**
     * Removes from the underlying collection the last element returned by the iterator (optional
     * operation). This method can be called only once per call to <code>next</code>.
     *
     * <p>The behavior of an iterator is unspecified if the underlying collection is modified while
     * the iteration is in progress in any way other than by calling this method.
     *
     * @throws  IllegalStateException  if the <code>next</code> method has not yet been called, or
     *                                 the <code>remove</code> method has already been called after
     *                                 the last call to the <code>next</code> method
     */
    public void remove() throws IllegalStateException {

        if (this.currentElement == -1) {
            throw new IllegalStateException();
        }

        this.source.remove(this.currentElement);
        this.currentElement = -1;
    }
}

package com.srbenoit.sparsearray;

/**
 * An interface for objects that need to be notified when the internal structure of a sparse array
 * changes.
 */
public interface SparseArrayListener {

    /**
     * Called when the capacity of the sparse array is changed, to allow other programs that need
     * to keep arrays of the same size to adjust their arrays.
     *
     * @param  array        the array whose capacity is changing
     * @param  newCapacity  the new capacity of the sparse array
     */
    void capacityChanged(final SparseArray<?> array, int newCapacity);
}
```

E.1.6 Color Management (com.srbenoit.color)

This package provides management of colors by name, using the color names popularized under the X window system, and also provides classes to manage color gradients.

```
package com.srbenoit.color;

/**
 * A listener interface for classes that wish to be notified when a user has made a selection from
 * a color chooser.
 */
public interface ColorNameChoiceListener {

    /**
     * Called when the user has selected a color name, or canceled their color selection.
     *
     * @param  colorName  the selected color name, or null if selection was canceled
     */
    void colorNameChosen(String colorName);
}

package com.srbenoit.color;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.Popup;
import javax.swing.PopupFactory;
import com.srbenoit.ui.OffscreenImagePanel;

/**
 * A chooser window that presents a palate of colors for the user to select from. The dialog has
 * the option of displaying color information, and may or may not show color names. The size of the
 * color boxes can also be controlled.
 */
public class ColorNameChooser extends MouseAdapter implements ActionListener {

    /** the listener to notify when the user has made a selection */
    private final transient ColorNameChoiceListener listener;

    /** object on which to wait for popup to close */
    private final transient Object waiter;

    /** the width of color boxes */
    private final transient int boxWidth;

    /** the height of color boxes */
    private final transient int boxHeight;

    /** the vertical spacing between color boxes */
    private final transient int boxSpacing;

    /** <code>true</code> to include color names */
    private final transient boolean incNames;

    /** <code>true</code> to include color hex values */
    private final transient boolean incHex;

    /** <code>true</code> to include borders around boxes */
    private final transient boolean incBorders;

    /** the index of the color that is being clicked on */
    private transient int clicked = -1;

    /** the index of the color that has been selected */
    private transient int selected = -1;

    /** an offscreen image with the color boxes & names */
    private transient SwatchImage img = null;
```

```
/** the GUI panel */
private transient JPanel panel = null;

/** the popup that is displaying the color chooser */
private transient Popup popup = null;

/**
 * Constructs a new <code>ColorNameChooser</code> with default settings. The defaults are to
 * have boxes sized at 12x12, with borders and names included, and box spacing set to 2.
 *
 * @param  theListener  the listener that is to be notified when the color choice has been made
 */
public ColorNameChooser(final ColorNameChoiceListener theListener) {

    this(theListener, 12, 12, 2, true, true, true);
}

/**
 * Construct a new <code>ColorNameChooser</code> with particular settings.
 *
 * @param  theListener     the listener that is to be notified when the color choice has been
 *                         made
 * @param  swatchWidth     the width of boxes
 * @param  swatchHeight    the height of boxes
 * @param  swatchSpacing   the number of pixels between boxes
 * @param  includeNames    <code>true</code> to include names, <code>false</code> otherwise
 * @param  includeHex      <code>true</code> to include hex values, <code>false</code>
 *                         otherwise
 * @param  includeBorders  <code>true</code> to include black borders around each box; <code>
 *                         false</code> otherwise
 */
public ColorNameChooser(final ColorNameChoiceListener theListener, final int swatchWidth,
    final int swatchHeight, final int swatchSpacing, final boolean includeNames,
    final boolean includeHex, final boolean includeBorders) {

    super();

    this.listener = theListener;
    this.boxWidth = swatchWidth;
    this.boxHeight = swatchHeight;
    this.boxSpacing = swatchSpacing;
    this.incNames = includeNames;
    this.incHex = includeHex;
    this.incBorders = includeBorders;

    this.waiter = new Object();
}

/**
 * Displays the popup.
 *
 * @param  owner  the owning <code>Component</code>
 * @param  xPos   the X location at which to show the popup
 * @param  yPos   the Y location at which to show the popup
 */
public void show(final Component owner, final int xPos, final int yPos) {

    if (this.popup == null) {
        this.popup = PopupFactory.getSharedInstance().getPopup(owner, buildUI(), xPos, yPos);
    }

    this.popup.show();
}

/**
 * Hides the popup.
 */
private void hide() {

    this.popup.hide();
    this.popup = null;

    synchronized (this.waiter) {
        this.waiter.notifyAll();
    }
}

/**
 * Waits for the user to make a selection and choose "OK", or to choose "Cancel".
 *
 * @return  the index of the selected color, or -1 if canceled
 */
public int waitForSelection() {

    while (this.popup != null) {

        synchronized (this.waiter) {
```

```
            try {
                this.waiter.wait();
            } catch (InterruptedException e) { }
        }
    }

    return this.selected;
}

/**
 * Builds the chooser UI.
 *
 * @return  the constructed panel
 */
private JPanel buildUI() {

    this.img = new SwatchImage(this.boxWidth, this.boxHeight, this.boxSpacing, this.incNames,
            this.incHex, this.incBorders);

    this.panel = new JPanel(new BorderLayout());
    this.panel.setBorder(BorderFactory.createCompoundBorder(
            BorderFactory.createLineBorder(Color.black, 1),
            BorderFactory.createRaisedBevelBorder()));

    buildScroll();
    buildButtonBar();

    return this.panel;
}

/**
 * Constructs the scroll pane that contains the offscreen image with the swatches.
 */
private void buildScroll() {

    OffscreenImagePanel offscreen;
    JScrollPane scroll;
    Dimension pref;

    offscreen = new OffscreenImagePanel(this.img.getImage());
    scroll = new JScrollPane(offscreen);
    offscreen.addMouseListener(this);

    if (this.img.getHeight() > 400) {
        pref = new Dimension(this.img.getWidth() + 24, 300);
    } else {
        pref = new Dimension(this.img.getWidth() + 4, this.img.getHeight() + 4);
    }

    scroll.setPreferredSize(pref);
    scroll.getVerticalScrollBar().setUnitIncrement(36);
    this.panel.add(scroll, BorderLayout.CENTER);
}

/**
 * Constructs the button bar and adds it to the panel.
 */
private void buildButtonBar() {

    JPanel buttonBar;
    JButton button;

    buttonBar = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 0));
    button = new JButton("Ok");
    button.setMargin(new Insets(0, 3, 0, 5));
    button.addActionListener(this);
    buttonBar.add(button);
    button = new JButton("Cancel");
    button.setMargin(new Insets(0, 3, 0, 5));
    button.addActionListener(this);
    buttonBar.add(button);
    this.panel.add(buttonBar, BorderLayout.SOUTH);
}

/**
 * Gets the selected color index, which is an index into the <code>ColorNames.COLORNAMES</code>
 * array of color names.
 *
 * @return  the selected color index, or -1 if nothing selected
 */
public int getSelected() {

    return this.selected;
}

/**
```

```
     * Sets the selected color by index.
     *
     * @param  index  the index of the selected color, which is an index into the <code>
     *                ColorNames.CNAMES</code> array of color names, or -1 to indicate no selection
     */
    public void setSelected(final int index) {

        this.selected = index;

        if (this.img != null) {
            this.img.setSelected(index);
            this.panel.repaint();
        }
    }

    /**
     * Called when the mouse is pressed. Tests whether the click occurs on a color, and if so,
     * records the color under the mouse cursor. If a subsequent release is on the same color, that
     * color will be assumed to have been "clicked".
     *
     * @param  evt  the mouse event
     */
    @Override public void mousePressed(final MouseEvent evt) {

        this.clicked = this.img.getHitIndex(evt.getX(), evt.getY());
    }

    /**
     * Called when the mouse is released. Tests whether the click occurs on a color, and if so,
     * checks whether that color was under the mouse when it was last pressed. If so, that color
     * will be assumed to have been "clicked".
     *
     * @param  evt  the mouse event
     */
    @Override public void mouseReleased(final MouseEvent evt) {

        if ((this.clicked != -1)
                && (this.clicked == this.img.getHitIndex(evt.getX(), evt.getY()))) {
            setSelected(this.clicked);
        }

        this.clicked = -1;
    }

    /**
     * Handler for action events generated when the user clicks the OK or Cancel buttons.
     *
     * @param  evt  the action event
     */
    public void actionPerformed(final ActionEvent evt) {

        String cmd;

        cmd = evt.getActionCommand();

        if ("Cancel".equals(cmd)) {
            this.selected = -1;

            if (this.listener != null) {
                this.listener.colorNameChosen(null);
            }
        } else if (this.selected == -1) {

            if (this.listener != null) {
                this.listener.colorNameChosen(null);
            }
        } else {

            if (this.listener != null) {
                this.listener.colorNameChosen(ColorNames.getInstance().getColorName(
                        this.selected));
            }
        }

        hide();
    }

    /**
     * Main method to bring up an instance of the dialog.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        new ColorNameChooser(null).show(null, 100, 100);
    }
}
```

```
package com.srbenoit.color;

import java.awt.Color;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * A color generator that allows colors to be selected by name, using the X11 color naming scheme,
 * taken from "rgb.txt" in the X11 distribution. Colors can also be specified in the hex format
 * used in HTML, with 2 hex characters per component (e.g. FF0000=bright red, 007700=dark green,
 * etc.).<br>
 * <br>
 * In addition to the standard X11 colors, the standard logo colors for Colorado State University
 * are available under the names "CSU Green" and "CSU Gold".
 */
public final class ColorNames extends LoggedObject {

    /** the singleton instance */
    private static ColorNames instance = new ColorNames();

    /**
     * Private constructor to enforce singleton model.
     */
    private ColorNames() {

        /* Empty */
    }

    /**
     * Get the singleton instance of the color names list.
     *
     * @return  the instance
     */
    public static ColorNames getInstance() {

        return instance;
    }

    /**
     * Tests whether a color name is valid. Valid names can be either the text name of a color, or
     * a hex triplet in the form RRGGBB. Letters in hex values can be either upper-case or
     * lower-case, but the text color names are case sensitive.
     *
     * @param   name  the name of the color
     * @return  <code>true</code> if the name is valid; <code>false</code> otherwise
     */
    public boolean isColorNameValid(final String name) {

        boolean valid;

        // Search for the name in the database
        valid = (indexOfName(name) != -1);

        // If not a preset, see if it's a hex value (RRGGBB)
        if ((!valid) && (name.length() == 6)) {

            try {
                Integer.parseInt(name, 16);
                valid = true;
            } catch (Exception exc) {
                LOG.log(Level.WARNING, "Invalid color: ''{0}''", name);
            }
        }

        return valid;
    }

    /**
     * Finds the name of the color that is the closest match to a given RGB value.
     *
     * @param   red     red (0-255)
     * @param   green   green (0-255)
     * @param   blue    blue (0-255)
     * @return  the closest match color
     */
    public String closestMatch(final int red, final int green, final int blue) {

        int inx;
        int picked;
        int deltar;
        int deltag;
        int deltab;
        int delta;
        int min;

        picked = 0;
        min = 255 * 3;
```

```
        for (inx = 0; inx < RGB.length; inx++) {
            deltar = RGB[inx][0] - red;
            deltag = RGB[inx][1] - green;
            deltab = RGB[inx][2] - blue;

            if (deltar < 0) {
                deltar = -deltar;
            }

            if (deltag < 0) {
                deltag = -deltag;
            }

            if (deltab < 0) {
                deltab = -deltab;
            }

            delta = deltar + deltag + deltab;

            if (delta < min) {
                picked = inx;
                min = delta;
            }
        }

        return CNAMES[picked];
    }

    /**
     * Gets the number of colors served by this class.
     *
     * @return  the number of colors
     */
    public int getNumColors() {

        return CNAMES.length;
    }

    /**
     * Gets a color name.
     *
     * @param   index  the index of the color name
     * @return  the color name
     */
    public String getColorName(final int index) {

        return CNAMES[index];
    }

    /**
     * Gets the red component of a color.
     *
     * @param   index  the index of the color
     * @return  the red component
     */
    public int getRed(final int index) {

        return RGB[index][0];
    }

    /**
     * Gets the green component of a color.
     *
     * @param   index  the index of the color
     * @return  the green component
     */
    public int getGreen(final int index) {

        return RGB[index][1];
    }

    /**
     * Gets the blue component of a color.
     *
     * @param   index  the index of the color
     * @return  the blue component
     */
    public int getBlue(final int index) {

        return RGB[index][2];
    }

    /**
     * Gets a color based on a name or hex triplet, and caches the color so future uses will not
     * create new instances.
     *
```

```
     * @param   index  the index of the color
     * @return  the generated <code>Color</code>
     */
    public Color getColor(final int index) {

        return new Color(RGB[index][0], RGB[index][1], RGB[index][2]);
    }

    /**
     * Gets a color based on a name or hex triplet.
     *
     * @param   name  the name or hex value of the color
     * @return  the generated <code>Color</code>, or <code>Color.BLACK</code> if the name could not
     *          be understood
     */
    public Color getColor(final String name) {

        int index;
        int value;
        Color color;

        index = indexOfName(name);

        if (index >= 0) {
            color = new Color(RGB[index][0], RGB[index][1], RGB[index][2]);
        } else if (name.length() == 6) {

            try {
                value = Integer.parseInt(name, 16);
                color = new Color((value >> 16) & 0xFF, (value >> 8) & 0xFF, (value) & 0xFF);
            } catch (Exception exc) {
                LOG.log(Level.WARNING, "Invalid color: ''{0}''", name);
                color = Color.BLACK;
            }
        } else {
            color = Color.BLACK;
        }

        return color;
    }

    /**
     * Looks up the index of a color name in the X11 database.
     *
     * @param   name  the color name
     * @return  the index, or -1 if the name was not found
     */
    public int indexOfName(final String name) {

        int index = -1;

        for (int i = 0; i < CNAMES.length; i++) {

            if (CNAMES[i].equals(name)) {
                index = i;

                break;
            }
        }

        return index;
    }

    /**
     * Generates a color a shade lighter than a given color.
     *
     * @param   col      the original color
     * @param   percent  the percentage by which to lighten the color
     * @return  the lightened color
     */
    public Color lighten(final Color col, final float percent) {

        int red;
        int grn;
        int blu;

        red = col.getRed();
        grn = col.getGreen();
        blu = col.getBlue();

        red = (int) (red + ((255 - red) * percent));
        grn = (int) (grn + ((255 - grn) * percent));
        blu = (int) (blu + ((255 - blu) * percent));

        return new Color(red, grn, blu);
    }
```

```
/**
 * Generates a color a shade lighter than a given color.
 *
 * @param   col       the original color
 * @param   percent   the percentage by which to darken the color
 * @return  the darkened color
 */
public Color darken(final Color col, final float percent) {

    int red;
    int grn;
    int blu;

    red = col.getRed();
    grn = col.getGreen();
    blu = col.getBlue();

    red = (int) (red - (red * percent));
    grn = (int) (grn - (grn * percent));
    blu = (int) (blu - (blu * percent));

    return new Color(red, grn, blu);
}

/** color names extracted from the X11 RGB database */
private static final String[] CNAMES = {
        "snow", "ghost white", "GhostWhite", "white smoke", "WhiteSmoke", "gainsboro",
        "floral white", "FloralWhite", "old lace", "OldLace", "linen", "antique white",
        "AntiqueWhite", "papaya whip", "PapayaWhip", "blanched almond", "BlanchedAlmond",
        "bisque", "peach puff", "PeachPuff", "navajo white", "NavajoWhite", "moccasin",
        "cornsilk", "ivory", "lemon chiffon", "LemonChiffon", "seashell", "honeydew",
        "mint cream", "MintCream", "azure", "alice blue", "AliceBlue", "lavender",
        "lavender blush", "LavenderBlush", "misty rose", "MistyRose", "white", "black",
        "dark slate gray", "DarkSlateGray", "dark slate grey", "DarkSlateGrey", "dim gray",
        "DimGray", "dim grey", "DimGrey", "slate gray", "SlateGray", "slate grey", "SlateGrey",
        "light slate gray", "LightSlateGray", "light slate grey", "LightSlateGrey", "gray",
        "grey", "light grey", "LightGrey", "light gray", "LightGray", "midnight blue",
        "MidnightBlue", "navy", "navy blue", "NavyBlue", "cornflower blue", "CornflowerBlue",
        "dark slate blue", "DarkSlateBlue", "slate blue", "SlateBlue", "medium slate blue",
        "MediumSlateBlue", "light slate blue", "LightSlateBlue", "medium blue", "MediumBlue",
        "royal blue", "RoyalBlue", "blue", "dodger blue", "DodgerBlue", "deep sky blue",
        "DeepSkyBlue", "sky blue", "SkyBlue", "light sky blue", "LightSkyBlue", "steel blue",
        "SteelBlue", "light steel blue", "LightSteelBlue", "light blue", "LightBlue",
        "powder blue", "PowderBlue", "pale turquoise", "PaleTurquoise", "dark turquoise",
        "DarkTurquoise", "medium turquoise", "MediumTurquoise", "turquoise", "cyan",
        "light cyan", "LightCyan", "cadet blue", "CadetBlue", "medium aquamarine",
        "MediumAquamarine", "aquamarine", "dark green", "DarkGreen", "dark olive green",
        "DarkOliveGreen", "dark sea green", "DarkSeaGreen", "sea green", "SeaGreen",
        "medium sea green", "MediumSeaGreen", "light sea green", "LightSeaGreen", "pale green",
        "PaleGreen", "spring green", "SpringGreen", "lawn green", "LawnGreen", "green",
        "chartreuse", "medium spring green", "MediumSpringGreen", "green yellow",
        "GreenYellow", "lime green", "LimeGreen", "yellow green", "YellowGreen",
        "forest green", "ForestGreen", "olive drab", "OliveDrab", "dark khaki", "DarkKhaki",
        "khaki", "pale goldenrod", "PaleGoldenrod", "light goldenrod yellow",
        "LightGoldenrodYellow", "light yellow", "LightYellow", "yellow", "gold",
        "light goldenrod", "LightGoldenrod", "goldenrod", "dark goldenrod", "DarkGoldenrod",
        "rosy brown", "RosyBrown", "indian red", "IndianRed", "saddle brown", "SaddleBrown",
        "sienna", "peru", "burlywood", "beige", "wheat", "sandy brown", "SandyBrown", "tan",
        "chocolate", "firebrick", "brown", "dark salmon", "DarkSalmon", "salmon",
        "light salmon", "LightSalmon", "orange", "dark orange", "DarkOrange", "coral",
        "light coral", "LightCoral", "tomato", "orange red", "OrangeRed", "red", "hot pink",
        "HotPink", "deep pink", "DeepPink", "pink", "light pink", "LightPink",
        "pale violet red", "PaleVioletRed", "maroon", "medium violet red", "MediumVioletRed",
        "violet red", "VioletRed", "magenta", "violet", "plum", "orchid", "medium orchid",
        "MediumOrchid", "dark orchid", "DarkOrchid", "dark violet", "DarkViolet",
        "blue violet", "BlueViolet", "purple", "medium purple", "MediumPurple", "thistle",
        "snow1", "snow2", "snow3", "snow4", "seashell1", "seashell2", "seashell3", "seashell4",
        "AntiqueWhite1", "AntiqueWhite2", "AntiqueWhite3", "AntiqueWhite4", "bisque1",
        "bisque2", "bisque3", "bisque4", "PeachPuff1", "PeachPuff2", "PeachPuff3",
        "PeachPuff4", "NavajoWhite1", "NavajoWhite2", "NavajoWhite3", "NavajoWhite4",
        "LemonChiffon1", "LemonChiffon2", "LemonChiffon3", "LemonChiffon4", "cornsilk1",
        "cornsilk2", "cornsilk3", "cornsilk4", "ivory1", "ivory2", "ivory3", "ivory4",
        "honeydew1", "honeydew2", "honeydew3", "honeydew4", "LavenderBlush1", "LavenderBlush2",
        "LavenderBlush3", "LavenderBlush4", "MistyRose1", "MistyRose2", "MistyRose3",
        "MistyRose4", "azure1", "azure2", "azure3", "azure4", "SlateBlue1", "SlateBlue2",
        "SlateBlue3", "SlateBlue4", "RoyalBlue1", "RoyalBlue2", "RoyalBlue3", "RoyalBlue4",
        "blue1", "blue2", "blue3", "blue4", "DodgerBlue1", "DodgerBlue2", "DodgerBlue3",
        "DodgerBlue4", "SteelBlue1", "SteelBlue2", "SteelBlue3", "SteelBlue4", "DeepSkyBlue1",
        "DeepSkyBlue2", "DeepSkyBlue3", "DeepSkyBlue4", "SkyBlue1", "SkyBlue2", "SkyBlue3",
        "SkyBlue4", "LightSkyBlue1", "LightSkyBlue2", "LightSkyBlue3", "LightSkyBlue4",
        "SlateGray1", "SlateGray2", "SlateGray3", "SlateGray4", "LightSteelBlue1",
        "LightSteelBlue2", "LightSteelBlue3", "LightSteelBlue4", "LightBlue1", "LightBlue2",
        "LightBlue3", "LightBlue4", "LightCyan1", "LightCyan2", "LightCyan3", "LightCyan4",
        "PaleTurquoise1", "PaleTurquoise2", "PaleTurquoise3", "PaleTurquoise4", "CadetBlue1",
        "CadetBlue2", "CadetBlue3", "CadetBlue4", "turquoise1", "turquoise2", "turquoise3",
        "turquoise4", "cyan1", "cyan2", "cyan3", "cyan4", "DarkSlateGray1", "DarkSlateGray2",
        "DarkSlateGray3", "DarkSlateGray4", "aquamarine1", "aquamarine2", "aquamarine3",
```

```
        "aquamarine4", "DarkSeaGreen1", "DarkSeaGreen2", "DarkSeaGreen3", "DarkSeaGreen4",
        "SeaGreen1", "SeaGreen2", "SeaGreen3", "SeaGreen4", "PaleGreen1", "PaleGreen2",
        "PaleGreen3", "PaleGreen4", "SpringGreen1", "SpringGreen2", "SpringGreen3",
        "SpringGreen4", "green1", "green2", "green3", "green4", "chartreuse1", "chartreuse2",
        "chartreuse3", "chartreuse4", "OliveDrab1", "OliveDrab2", "OliveDrab3", "OliveDrab4",
        "DarkOliveGreen1", "DarkOliveGreen2", "DarkOliveGreen3", "DarkOliveGreen4", "khaki1",
        "khaki2", "khaki3", "khaki4", "LightGoldenrod1", "LightGoldenrod2", "LightGoldenrod3",
        "LightGoldenrod4", "LightYellow1", "LightYellow2", "LightYellow3", "LightYellow4",
        "yellow1", "yellow2", "yellow3", "yellow4", "gold1", "gold2", "gold3", "gold4",
        "goldenrod1", "goldenrod2", "goldenrod3", "goldenrod4", "DarkGoldenrod1",
        "DarkGoldenrod2", "DarkGoldenrod3", "DarkGoldenrod4", "RosyBrown1", "RosyBrown2",
        "RosyBrown3", "RosyBrown4", "IndianRed1", "IndianRed2", "IndianRed3", "IndianRed4",
        "sienna1", "sienna2", "sienna3", "sienna4", "burlywood1", "burlywood2", "burlywood3",
        "burlywood4", "wheat1", "wheat2", "wheat3", "wheat4", "tan1", "tan2", "tan3", "tan4",
        "chocolate1", "chocolate2", "chocolate3", "chocolate4", "firebrick1", "firebrick2",
        "firebrick3", "firebrick4", "brown1", "brown2", "brown3", "brown4", "salmon1",
        "salmon2", "salmon3", "salmon4", "LightSalmon1", "LightSalmon2", "LightSalmon3",
        "LightSalmon4", "orange1", "orange2", "orange3", "orange4", "DarkOrange1",
        "DarkOrange2", "DarkOrange3", "DarkOrange4", "coral1", "coral2", "coral3", "coral4",
        "tomato1", "tomato2", "tomato3", "tomato4", "OrangeRed1", "OrangeRed2", "OrangeRed3",
        "OrangeRed4", "red1", "red2", "red3", "red4", "DeepPink1", "DeepPink2", "DeepPink3",
        "DeepPink4", "HotPink1", "HotPink2", "HotPink3", "HotPink4", "pink1", "pink2", "pink3",
        "pink4", "LightPink1", "LightPink2", "LightPink3", "LightPink4", "PaleVioletRed1",
        "PaleVioletRed2", "PaleVioletRed3", "PaleVioletRed4", "maroon1", "maroon2", "maroon3",
        "maroon4", "VioletRed1", "VioletRed2", "VioletRed3", "VioletRed4", "magenta1",
        "magenta2", "magenta3", "magenta4", "orchid1", "orchid2", "orchid3", "orchid4",
        "plum1", "plum2", "plum3", "plum4", "MediumOrchid1", "MediumOrchid2", "MediumOrchid3",
        "MediumOrchid4", "DarkOrchid1", "DarkOrchid2", "DarkOrchid3", "DarkOrchid4", "purple1",
        "purple2", "purple3", "purple4", "MediumPurple1", "MediumPurple2", "MediumPurple3",
        "MediumPurple4", "thistle1", "thistle2", "thistle3", "thistle4", "gray0", "grey0",
        "gray1", "grey1", "gray2", "grey2", "gray3", "grey3", "gray4", "grey4", "gray5",
        "grey5", "gray6", "grey6", "gray7", "grey7", "gray8", "grey8", "gray9", "grey9",
        "gray10", "grey10", "gray11", "grey11", "gray12", "grey12", "gray13", "grey13",
        "gray14", "grey14", "gray15", "grey15", "gray16", "grey16", "gray17", "grey17",
        "gray18", "grey18", "gray19", "grey19", "gray20", "grey20", "gray21", "grey21",
        "gray22", "grey22", "gray23", "grey23", "gray24", "grey24", "gray25", "grey25",
        "gray26", "grey26", "gray27", "grey27", "gray28", "grey28", "gray29", "grey29",
        "gray30", "grey30", "gray31", "grey31", "gray32", "grey32", "gray33", "grey33",
        "gray34", "grey34", "gray35", "grey35", "gray36", "grey36", "gray37", "grey37",
        "gray38", "grey38", "gray39", "grey39", "gray40", "grey40", "gray41", "grey41",
        "gray42", "grey42", "gray43", "grey43", "gray44", "grey44", "gray45", "grey45",
        "gray46", "grey46", "gray47", "grey47", "gray48", "grey48", "gray49", "grey49",
        "gray50", "grey50", "gray51", "grey51", "gray52", "grey52", "gray53", "grey53",
        "gray54", "grey54", "gray55", "grey55", "gray56", "grey56", "gray57", "grey57",
        "gray58", "grey58", "gray59", "grey59", "gray60", "grey60", "gray61", "grey61",
        "gray62", "grey62", "gray63", "grey63", "gray64", "grey64", "gray65", "grey65",
        "gray66", "grey66", "gray67", "grey67", "gray68", "grey68", "gray69", "grey69",
        "gray70", "grey70", "gray71", "grey71", "gray72", "grey72", "gray73", "grey73",
        "gray74", "grey74", "gray75", "grey75", "gray76", "grey76", "gray77", "grey77",
        "gray78", "grey78", "gray79", "grey79", "gray80", "grey80", "gray81", "grey81",
        "gray82", "grey82", "gray83", "grey83", "gray84", "grey84", "gray85", "grey85",
        "gray86", "grey86", "gray87", "grey87", "gray88", "grey88", "gray89", "grey89",
        "gray90", "grey90", "gray91", "grey91", "gray92", "grey92", "gray93", "grey93",
        "gray94", "grey94", "gray95", "grey95", "gray96", "grey96", "gray97", "grey97",
        "gray98", "grey98", "gray99", "grey99", "gray100", "grey100", "dark_grey", "DarkGrey",
        "dark_gray", "DarkGray", "dark_blue", "DarkBlue", "dark_cyan", "DarkCyan",
        "dark_magenta", "DarkMagenta", "dark_red", "DarkRed", "light_green", "LightGreen",
        "CSU_Green", "CSU_Green_1", "CSU_Green_2", "CSU_Green_3", "CSU_Green_4", "CSU_Gold",
        "CSU_Gold_1", "CSU_Gold_2", "CSU_Gold_3", "CSU_Gold_4", "Light_CSU_Gold"
    };

/** RGB color values corresponding to color names */
private static final int[][] RGB = {
        { 255, 250, 250 },
        { 248, 248, 255 },
        { 248, 248, 255 },
        { 245, 245, 245 },
        { 245, 245, 245 },
        { 220, 220, 220 },
        { 255, 250, 240 },
        { 255, 250, 240 },
        { 253, 245, 230 },
        { 253, 245, 230 },
        { 250, 240, 230 },
        { 250, 235, 215 },
        { 250, 235, 215 },
        { 255, 239, 213 },
        { 255, 239, 213 },
        { 255, 235, 205 },
        { 255, 235, 205 },
        { 255, 228, 196 },
        { 255, 218, 185 },
        { 255, 218, 185 },
        { 255, 222, 173 },
        { 255, 222, 173 },
        { 255, 228, 181 },
        { 255, 248, 220 },
```

```
{ 255, 255, 240 },
{ 255, 250, 205 },
{ 255, 250, 205 },
{ 255, 245, 238 },
{ 240, 255, 240 },
{ 245, 255, 250 },
{ 245, 255, 250 },
{ 240, 255, 255 },
{ 240, 248, 255 },
{ 240, 248, 255 },
{ 230, 230, 250 },
{ 255, 240, 245 },
{ 255, 240, 245 },
{ 255, 228, 225 },
{ 255, 228, 225 },
{ 255, 255, 255 },
{ 0, 0, 0 },
{ 47, 79, 79 },
{ 47, 79, 79 },
{ 47, 79, 79 },
{ 47, 79, 79 },
{ 105, 105, 105 },
{ 105, 105, 105 },
{ 105, 105, 105 },
{ 105, 105, 105 },
{ 112, 128, 144 },
{ 112, 128, 144 },
{ 112, 128, 144 },
{ 112, 128, 144 },
{ 119, 136, 153 },
{ 119, 136, 153 },
{ 119, 136, 153 },
{ 119, 136, 153 },
{ 190, 190, 190 },
{ 190, 190, 190 },
{ 211, 211, 211 },
{ 211, 211, 211 },
{ 211, 211, 211 },
{ 211, 211, 211 },
{ 25, 25, 112 },
{ 25, 25, 112 },
{ 0, 0, 128 },
{ 0, 0, 128 },
{ 0, 0, 128 },
{ 100, 149, 237 },
{ 100, 149, 237 },
{ 72, 61, 139 },
{ 72, 61, 139 },
{ 106, 90, 205 },
{ 106, 90, 205 },
{ 123, 104, 238 },
{ 123, 104, 238 },
{ 132, 112, 255 },
{ 132, 112, 255 },
{ 0, 0, 205 },
{ 0, 0, 205 },
{ 65, 105, 225 },
{ 65, 105, 225 },
{ 0, 0, 255 },
{ 30, 144, 255 },
{ 30, 144, 255 },
{ 0, 191, 255 },
{ 0, 191, 255 },
{ 135, 206, 235 },
{ 135, 206, 235 },
{ 135, 206, 250 },
{ 135, 206, 250 },
{ 70, 130, 180 },
{ 70, 130, 180 },
{ 176, 196, 222 },
{ 176, 196, 222 },
{ 173, 216, 230 },
{ 173, 216, 230 },
{ 176, 224, 230 },
{ 176, 224, 230 },
{ 175, 238, 238 },
{ 175, 238, 238 },
{ 0, 206, 209 },
{ 0, 206, 209 },
{ 72, 209, 204 },
{ 72, 209, 204 },
{ 64, 224, 208 },
{ 0, 255, 255 },
{ 224, 255, 255 },
{ 224, 255, 255 },
{ 95, 158, 160 },
{ 95, 158, 160 },
```

```
{ 102, 205, 170 },
{ 102, 205, 170 },
{ 127, 255, 212 },
{ 0, 100, 0 },
{ 0, 100, 0 },
{ 85, 107, 47 },
{ 85, 107, 47 },
{ 143, 188, 143 },
{ 143, 188, 143 },
{ 46, 139, 87 },
{ 46, 139, 87 },
{ 60, 179, 113 },
{ 60, 179, 113 },
{ 32, 178, 170 },
{ 32, 178, 170 },
{ 152, 251, 152 },
{ 152, 251, 152 },
{ 0, 255, 127 },
{ 0, 255, 127 },
{ 124, 252, 0 },
{ 124, 252, 0 },
{ 0, 255, 0 },
{ 127, 255, 0 },
{ 0, 250, 154 },
{ 0, 250, 154 },
{ 173, 255, 47 },
{ 173, 255, 47 },
{ 50, 205, 50 },
{ 50, 205, 50 },
{ 154, 205, 50 },
{ 154, 205, 50 },
{ 34, 139, 34 },
{ 34, 139, 34 },
{ 107, 142, 35 },
{ 107, 142, 35 },
{ 189, 183, 107 },
{ 189, 183, 107 },
{ 240, 230, 140 },
{ 238, 232, 170 },
{ 238, 232, 170 },
{ 250, 250, 210 },
{ 250, 250, 210 },
{ 255, 255, 224 },
{ 255, 255, 224 },
{ 255, 255, 0 },
{ 255, 215, 0 },
{ 238, 221, 130 },
{ 238, 221, 130 },
{ 218, 165, 32 },
{ 184, 134, 11 },
{ 184, 134, 11 },
{ 188, 143, 143 },
{ 188, 143, 143 },
{ 205, 92, 92 },
{ 205, 92, 92 },
{ 139, 69, 19 },
{ 139, 69, 19 },
{ 160, 82, 45 },
{ 205, 133, 63 },
{ 222, 184, 135 },
{ 245, 245, 220 },
{ 245, 222, 179 },
{ 244, 164, 96 },
{ 244, 164, 96 },
{ 210, 180, 140 },
{ 210, 105, 30 },
{ 178, 34, 34 },
{ 165, 42, 42 },
{ 233, 150, 122 },
{ 233, 150, 122 },
{ 250, 128, 114 },
{ 255, 160, 122 },
{ 255, 160, 122 },
{ 255, 165, 0 },
{ 255, 140, 0 },
{ 255, 140, 0 },
{ 255, 127, 80 },
{ 240, 128, 128 },
{ 240, 128, 128 },
{ 255, 99, 71 },
{ 255, 69, 0 },
{ 255, 69, 0 },
{ 255, 0, 0 },
{ 255, 105, 180 },
{ 255, 105, 180 },
{ 255, 20, 147 },
{ 255, 20, 147 },
```

```
{ 255, 192, 203 },
{ 255, 182, 193 },
{ 255, 182, 193 },
{ 219, 112, 147 },
{ 219, 112, 147 },
{ 176, 48, 96 },
{ 199, 21, 133 },
{ 199, 21, 133 },
{ 208, 32, 144 },
{ 208, 32, 144 },
{ 255, 0, 255 },
{ 238, 130, 238 },
{ 221, 160, 221 },
{ 218, 112, 214 },
{ 186, 85, 211 },
{ 186, 85, 211 },
{ 153, 50, 204 },
{ 153, 50, 204 },
{ 148, 0, 211 },
{ 148, 0, 211 },
{ 138, 43, 226 },
{ 138, 43, 226 },
{ 160, 32, 240 },
{ 147, 112, 219 },
{ 147, 112, 219 },
{ 216, 191, 216 },
{ 255, 250, 250 },
{ 238, 233, 233 },
{ 205, 201, 201 },
{ 139, 137, 137 },
{ 255, 245, 238 },
{ 238, 229, 222 },
{ 205, 197, 191 },
{ 139, 134, 130 },
{ 255, 239, 219 },
{ 238, 223, 204 },
{ 205, 192, 176 },
{ 139, 131, 120 },
{ 255, 228, 196 },
{ 238, 213, 183 },
{ 205, 183, 158 },
{ 139, 125, 107 },
{ 255, 218, 185 },
{ 238, 203, 173 },
{ 205, 175, 149 },
{ 139, 119, 101 },
{ 255, 222, 173 },
{ 238, 207, 161 },
{ 205, 179, 139 },
{ 139, 121, 94 },
{ 255, 250, 205 },
{ 238, 233, 191 },
{ 205, 201, 165 },
{ 139, 137, 112 },
{ 255, 248, 220 },
{ 238, 232, 205 },
{ 205, 200, 177 },
{ 139, 136, 120 },
{ 255, 255, 240 },
{ 238, 238, 224 },
{ 205, 205, 193 },
{ 139, 139, 131 },
{ 240, 255, 240 },
{ 224, 238, 224 },
{ 193, 205, 193 },
{ 131, 139, 131 },
{ 255, 240, 245 },
{ 238, 224, 229 },
{ 205, 193, 197 },
{ 139, 131, 134 },
{ 255, 228, 225 },
{ 238, 213, 210 },
{ 205, 183, 181 },
{ 139, 125, 123 },
{ 240, 255, 255 },
{ 224, 238, 238 },
{ 193, 205, 205 },
{ 131, 139, 139 },
{ 131, 111, 255 },
{ 122, 103, 238 },
{ 105, 89, 205 },
{ 71, 60, 139 },
{ 72, 118, 255 },
{ 67, 110, 238 },
{ 58, 95, 205 },
{ 39, 64, 139 },
{ 0, 0, 255 },
```

```
{ 0, 0, 238 },
{ 0, 0, 205 },
{ 0, 0, 139 },
{ 30, 144, 255 },
{ 28, 134, 238 },
{ 24, 116, 205 },
{ 16, 78, 139 },
{ 99, 184, 255 },
{ 92, 172, 238 },
{ 79, 148, 205 },
{ 54, 100, 139 },
{ 0, 191, 255 },
{ 0, 178, 238 },
{ 0, 154, 205 },
{ 0, 104, 139 },
{ 135, 206, 255 },
{ 126, 192, 238 },
{ 108, 166, 205 },
{ 74, 112, 139 },
{ 176, 226, 255 },
{ 164, 211, 238 },
{ 141, 182, 205 },
{ 96, 123, 139 },
{ 198, 226, 255 },
{ 185, 211, 238 },
{ 159, 182, 205 },
{ 108, 123, 139 },
{ 202, 225, 255 },
{ 188, 210, 238 },
{ 162, 181, 205 },
{ 110, 123, 139 },
{ 191, 239, 255 },
{ 178, 223, 238 },
{ 154, 192, 205 },
{ 104, 131, 139 },
{ 224, 255, 255 },
{ 209, 238, 238 },
{ 180, 205, 205 },
{ 122, 139, 139 },
{ 187, 255, 255 },
{ 174, 238, 238 },
{ 150, 205, 205 },
{ 102, 139, 139 },
{ 152, 245, 255 },
{ 142, 229, 238 },
{ 122, 197, 205 },
{ 83, 134, 139 },
{ 0, 245, 255 },
{ 0, 229, 238 },
{ 0, 197, 205 },
{ 0, 134, 139 },
{ 0, 255, 255 },
{ 0, 238, 238 },
{ 0, 205, 205 },
{ 0, 139, 139 },
{ 151, 255, 255 },
{ 141, 238, 238 },
{ 121, 205, 205 },
{ 82, 139, 139 },
{ 127, 255, 212 },
{ 118, 238, 198 },
{ 102, 205, 170 },
{ 69, 139, 116 },
{ 193, 255, 193 },
{ 180, 238, 180 },
{ 155, 205, 155 },
{ 105, 139, 105 },
{ 84, 255, 159 },
{ 78, 238, 148 },
{ 67, 205, 128 },
{ 46, 139, 87 },
{ 154, 255, 154 },
{ 144, 238, 144 },
{ 124, 205, 124 },
{ 84, 139, 84 },
{ 0, 255, 127 },
{ 0, 238, 118 },
{ 0, 205, 102 },
{ 0, 139, 69 },
{ 0, 255, 0 },
{ 0, 238, 0 },
{ 0, 205, 0 },
{ 0, 139, 0 },
{ 127, 255, 0 },
{ 118, 238, 0 },
{ 102, 205, 0 },
{ 69, 139, 0 },
```

```
{ 192, 255, 62 },
{ 179, 238, 58 },
{ 154, 205, 50 },
{ 105, 139, 34 },
{ 202, 255, 112 },
{ 188, 238, 104 },
{ 162, 205, 90 },
{ 110, 139, 61 },
{ 255, 246, 143 },
{ 238, 230, 133 },
{ 205, 198, 115 },
{ 139, 134, 78 },
{ 255, 236, 139 },
{ 238, 220, 130 },
{ 205, 190, 112 },
{ 139, 129, 76 },
{ 255, 255, 224 },
{ 238, 238, 209 },
{ 205, 205, 180 },
{ 139, 139, 122 },
{ 255, 255, 0 },
{ 238, 238, 0 },
{ 205, 205, 0 },
{ 139, 139, 0 },
{ 255, 215, 0 },
{ 238, 201, 0 },
{ 205, 173, 0 },
{ 139, 117, 0 },
{ 255, 193, 37 },
{ 238, 180, 34 },
{ 205, 155, 29 },
{ 139, 105, 20 },
{ 255, 185, 15 },
{ 238, 173, 14 },
{ 205, 149, 12 },
{ 139, 101, 8 },
{ 255, 193, 193 },
{ 238, 180, 180 },
{ 205, 155, 155 },
{ 139, 105, 105 },
{ 255, 106, 106 },
{ 238, 99, 99 },
{ 205, 85, 85 },
{ 139, 58, 58 },
{ 255, 130, 71 },
{ 238, 121, 66 },
{ 205, 104, 57 },
{ 139, 71, 38 },
{ 255, 211, 155 },
{ 238, 197, 145 },
{ 205, 170, 125 },
{ 139, 115, 85 },
{ 255, 231, 186 },
{ 238, 216, 174 },
{ 205, 186, 150 },
{ 139, 126, 102 },
{ 255, 165, 79 },
{ 238, 154, 73 },
{ 205, 133, 63 },
{ 139, 90, 43 },
{ 255, 127, 36 },
{ 238, 118, 33 },
{ 205, 102, 29 },
{ 139, 69, 19 },
{ 255, 48, 48 },
{ 238, 44, 44 },
{ 205, 38, 38 },
{ 139, 26, 26 },
{ 255, 64, 64 },
{ 238, 59, 59 },
{ 205, 51, 51 },
{ 139, 35, 35 },
{ 255, 140, 105 },
{ 238, 130, 98 },
{ 205, 112, 84 },
{ 139, 76, 57 },
{ 255, 160, 122 },
{ 238, 149, 114 },
{ 205, 129, 98 },
{ 139, 87, 66 },
{ 255, 165, 0 },
{ 238, 154, 0 },
{ 205, 133, 0 },
{ 139, 90, 0 },
{ 255, 127, 0 },
{ 238, 118, 0 },
{ 205, 102, 0 },
```

```
{ 139, 69, 0 },
{ 255, 114, 86 },
{ 238, 106, 80 },
{ 205, 91, 69 },
{ 139, 62, 47 },
{ 255, 99, 71 },
{ 238, 92, 66 },
{ 205, 79, 57 },
{ 139, 54, 38 },
{ 255, 69, 0 },
{ 238, 64, 0 },
{ 205, 55, 0 },
{ 139, 37, 0 },
{ 255, 0, 0 },
{ 238, 0, 0 },
{ 205, 0, 0 },
{ 139, 0, 0 },
{ 255, 20, 147 },
{ 238, 18, 137 },
{ 205, 16, 118 },
{ 139, 10, 80 },
{ 255, 110, 180 },
{ 238, 106, 167 },
{ 205, 96, 144 },
{ 139, 58, 98 },
{ 255, 181, 197 },
{ 238, 169, 184 },
{ 205, 145, 158 },
{ 139, 99, 108 },
{ 255, 174, 185 },
{ 238, 162, 173 },
{ 205, 140, 149 },
{ 139, 95, 101 },
{ 255, 130, 171 },
{ 238, 121, 159 },
{ 205, 104, 137 },
{ 139, 71, 93 },
{ 255, 52, 179 },
{ 238, 48, 167 },
{ 205, 41, 144 },
{ 139, 28, 98 },
{ 255, 62, 150 },
{ 238, 58, 140 },
{ 205, 50, 120 },
{ 139, 34, 82 },
{ 255, 0, 255 },
{ 238, 0, 238 },
{ 205, 0, 205 },
{ 139, 0, 139 },
{ 255, 131, 250 },
{ 238, 122, 233 },
{ 205, 105, 201 },
{ 139, 71, 137 },
{ 255, 187, 255 },
{ 238, 174, 238 },
{ 205, 150, 205 },
{ 139, 102, 139 },
{ 224, 102, 255 },
{ 209, 95, 238 },
{ 180, 82, 205 },
{ 122, 55, 139 },
{ 191, 62, 255 },
{ 178, 58, 238 },
{ 154, 50, 205 },
{ 104, 34, 139 },
{ 155, 48, 255 },
{ 145, 44, 238 },
{ 125, 38, 205 },
{ 85, 26, 139 },
{ 171, 130, 255 },
{ 159, 121, 238 },
{ 137, 104, 205 },
{ 93, 71, 139 },
{ 255, 225, 255 },
{ 238, 210, 238 },
{ 205, 181, 205 },
{ 139, 123, 139 },
{ 0, 0, 0 },
{ 0, 0, 0 },
{ 3, 3, 3 },
{ 3, 3, 3 },
{ 5, 5, 5 },
{ 5, 5, 5 },
{ 8, 8, 8 },
{ 8, 8, 8 },
{ 10, 10, 10 },
{ 10, 10, 10 },
```

```
{ 13, 13, 13 },
{ 13, 13, 13 },
{ 15, 15, 15 },
{ 15, 15, 15 },
{ 18, 18, 18 },
{ 18, 18, 18 },
{ 20, 20, 20 },
{ 20, 20, 20 },
{ 23, 23, 23 },
{ 23, 23, 23 },
{ 26, 26, 26 },
{ 26, 26, 26 },
{ 28, 28, 28 },
{ 28, 28, 28 },
{ 31, 31, 31 },
{ 31, 31, 31 },
{ 33, 33, 33 },
{ 33, 33, 33 },
{ 36, 36, 36 },
{ 36, 36, 36 },
{ 38, 38, 38 },
{ 38, 38, 38 },
{ 41, 41, 41 },
{ 41, 41, 41 },
{ 43, 43, 43 },
{ 43, 43, 43 },
{ 46, 46, 46 },
{ 46, 46, 46 },
{ 48, 48, 48 },
{ 48, 48, 48 },
{ 51, 51, 51 },
{ 51, 51, 51 },
{ 54, 54, 54 },
{ 54, 54, 54 },
{ 56, 56, 56 },
{ 56, 56, 56 },
{ 59, 59, 59 },
{ 59, 59, 59 },
{ 61, 61, 61 },
{ 61, 61, 61 },
{ 64, 64, 64 },
{ 64, 64, 64 },
{ 66, 66, 66 },
{ 66, 66, 66 },
{ 69, 69, 69 },
{ 69, 69, 69 },
{ 71, 71, 71 },
{ 71, 71, 71 },
{ 74, 74, 74 },
{ 74, 74, 74 },
{ 77, 77, 77 },
{ 77, 77, 77 },
{ 79, 79, 79 },
{ 79, 79, 79 },
{ 82, 82, 82 },
{ 82, 82, 82 },
{ 84, 84, 84 },
{ 84, 84, 84 },
{ 87, 87, 87 },
{ 87, 87, 87 },
{ 89, 89, 89 },
{ 89, 89, 89 },
{ 92, 92, 92 },
{ 92, 92, 92 },
{ 94, 94, 94 },
{ 94, 94, 94 },
{ 97, 97, 97 },
{ 97, 97, 97 },
{ 99, 99, 99 },
{ 99, 99, 99 },
{ 102, 102, 102 },
{ 102, 102, 102 },
{ 105, 105, 105 },
{ 105, 105, 105 },
{ 107, 107, 107 },
{ 107, 107, 107 },
{ 110, 110, 110 },
{ 110, 110, 110 },
{ 112, 112, 112 },
{ 112, 112, 112 },
{ 115, 115, 115 },
{ 115, 115, 115 },
{ 117, 117, 117 },
{ 117, 117, 117 },
{ 120, 120, 120 },
{ 120, 120, 120 },
{ 122, 122, 122 },
```

```
{ 122, 122, 122 },
{ 125, 125, 125 },
{ 125, 125, 125 },
{ 127, 127, 127 },
{ 127, 127, 127 },
{ 130, 130, 130 },
{ 130, 130, 130 },
{ 133, 133, 133 },
{ 133, 133, 133 },
{ 135, 135, 135 },
{ 135, 135, 135 },
{ 138, 138, 138 },
{ 138, 138, 138 },
{ 140, 140, 140 },
{ 140, 140, 140 },
{ 143, 143, 143 },
{ 143, 143, 143 },
{ 145, 145, 145 },
{ 145, 145, 145 },
{ 148, 148, 148 },
{ 148, 148, 148 },
{ 150, 150, 150 },
{ 150, 150, 150 },
{ 153, 153, 153 },
{ 153, 153, 153 },
{ 156, 156, 156 },
{ 156, 156, 156 },
{ 158, 158, 158 },
{ 158, 158, 158 },
{ 161, 161, 161 },
{ 161, 161, 161 },
{ 163, 163, 163 },
{ 163, 163, 163 },
{ 166, 166, 166 },
{ 166, 166, 166 },
{ 168, 168, 168 },
{ 168, 168, 168 },
{ 171, 171, 171 },
{ 171, 171, 171 },
{ 173, 173, 173 },
{ 173, 173, 173 },
{ 176, 176, 176 },
{ 176, 176, 176 },
{ 179, 179, 179 },
{ 179, 179, 179 },
{ 181, 181, 181 },
{ 181, 181, 181 },
{ 184, 184, 184 },
{ 184, 184, 184 },
{ 186, 186, 186 },
{ 186, 186, 186 },
{ 189, 189, 189 },
{ 189, 189, 189 },
{ 191, 191, 191 },
{ 191, 191, 191 },
{ 194, 194, 194 },
{ 194, 194, 194 },
{ 196, 196, 196 },
{ 196, 196, 196 },
{ 199, 199, 199 },
{ 199, 199, 199 },
{ 201, 201, 201 },
{ 201, 201, 201 },
{ 204, 204, 204 },
{ 204, 204, 204 },
{ 207, 207, 207 },
{ 207, 207, 207 },
{ 209, 209, 209 },
{ 209, 209, 209 },
{ 212, 212, 212 },
{ 212, 212, 212 },
{ 214, 214, 214 },
{ 214, 214, 214 },
{ 217, 217, 217 },
{ 217, 217, 217 },
{ 219, 219, 219 },
{ 219, 219, 219 },
{ 222, 222, 222 },
{ 222, 222, 222 },
{ 224, 224, 224 },
{ 224, 224, 224 },
{ 227, 227, 227 },
{ 227, 227, 227 },
{ 229, 229, 229 },
{ 229, 229, 229 },
{ 232, 232, 232 },
{ 232, 232, 232 },
```

```
            { 235, 235, 235 },
            { 235, 235, 235 },
            { 237, 237, 237 },
            { 237, 237, 237 },
            { 240, 240, 240 },
            { 240, 240, 240 },
            { 242, 242, 242 },
            { 242, 242, 242 },
            { 245, 245, 245 },
            { 245, 245, 245 },
            { 247, 247, 247 },
            { 247, 247, 247 },
            { 250, 250, 250 },
            { 250, 250, 250 },
            { 252, 252, 252 },
            { 252, 252, 252 },
            { 255, 255, 255 },
            { 255, 255, 255 },
            { 169, 169, 169 },
            { 169, 169, 169 },
            { 169, 169, 169 },
            { 169, 169, 169 },
            { 0, 0, 139 },
            { 0, 0, 139 },
            { 0, 139, 139 },
            { 0, 139, 139 },
            { 139, 0, 139 },
            { 139, 0, 139 },
            { 139, 0, 0 },
            { 139, 0, 0 },
            { 144, 238, 144 },
            { 144, 238, 144 },
            { 0, 107, 84 }, // CSU green
            { 0, 255, 200 }, // CSU green 1
            { 0, 238, 187 }, // CSU green 2
            { 0, 205, 161 }, // CSU green 3
            { 0, 139, 109 }, // CSU green 4
            { 255, 198, 30 }, // CSU gold
            { 255, 198, 30 }, // CSU gold 1
            { 238, 185, 28 }, // CSU gold 2
            { 205, 159, 24 }, // CSU gold 3
            { 139, 108, 16 }, // CSU gold 4
            { 255, 226, 142 }, // light CSU gold
        };
}

package com.srbenoit.color;

import java.awt.Color;

/**
 * A class that generates a sweep of color through a range of hues, and can convert a data value to
 * a color given a value range.
 */
public class Gradient {

    /** the colors */
    private final transient Color[] colors;

    /**
     * Constructs a new <code>Gradient</code> using a given number of colors.
     *
     * @param  numColors  the number of colors
     * @param  numCycles  the number of cycles about hue to use (if this is 1 or less, a single
     *                    cycle with lightness 0.5 will be used; otherwise, we will spiral up from
     *                    black to white through the given number of hue cycles)
     */
    public Gradient(final int numColors, final int numCycles) {

        float fraction;
        float hue;
        float lightness;
        this.colors = new Color[numColors];

        for (int i = 0; i < numColors; i++) {
            hue = 90 + ((360.0f * i * numCycles) / numColors);

            while (hue > 360) {
                hue -= 360;
            }

            fraction = (float) i / numColors;

            if (i == 0) {
                lightness = 0;
            } else {
                lightness = (float) (0.5 - (Math.log((1 / fraction) - 1) / 12.0));
```

```
        }

        this.colors[i] = colorFromHSL(hue, 1.0f, lightness);
    }
}

/**
 * Gets the number of colors.
 *
 * @return  the number of colors
 */
public int getNumColors() {

    return this.colors.length;
}

/**
 * Gets a particular color.
 *
 * @param   index  the index of the color to get
 * @return  the color
 */
public Color getColor(final int index) {

    return this.colors[index];
}

/**
 * Given a hue in the range 0-1 and a lightness in the range 0-1, generates the corresponding
 * <code>Color</code>. Saturation is assumed to be maximum.
 *
 * @param   hue          the hue angle (0-360)
 * @param   saturation   the saturation
 * @param   lightness    the lightness
 * @return  the color
 */
private Color colorFromHSL(final float hue, final float saturation, final float lightness) {

    float temp1;
    float temp2;
    float[] color;

    if (saturation == 0) {
        color = new float[] { lightness, lightness, lightness };
    } else {

        if (lightness < 0.5) {
            temp2 = 2 * (lightness + saturation) * lightness;
        } else {
            temp2 = lightness + saturation - (lightness * saturation);
        }

        temp1 = (2 * lightness) - temp2;

        color = new float[] {
                computeColor(temp1, temp2, hue + 120), computeColor(temp1, temp2, hue),
                computeColor(temp1, temp2, hue - 120)
            };
    }

    return new Color(color[0], color[1], color[2]);
}

/**
 * Computes a color component.
 *
 * @param   temp1  lightness/saturation factor 1
 * @param   temp2  lightness/saturation factor 2
 * @param   hue    the hue component (0-360)
 * @return  the color component
 */
private float computeColor(final float temp1, final float temp2, final float hue) {

    float hue2;
    float color;

    if (hue < 0) {
        hue2 = hue + 360;
    } else if (hue > 360) {
        hue2 = hue - 360;
    } else {
        hue2 = hue;
    }

    if (hue2 < 60) {
        color = temp1 + ((temp2 - temp1) * hue2 / 60);
    } else if (hue2 < 180) {
```

```
            color = temp2;
        } else if (hue2 < 240) {
            color = temp1 + ((temp2 - temp1) * (240 - hue2) / 60);
        } else {
            color = temp1;
        }

        if (color > 1) {
            color = 1;
        } else if (color < 0) {
            color = 0;
        }

        return color;
    }
}

package com.srbenoit.color;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

/**
 * A panel to display gradients with various parameters.
 */
public class GradientPanel extends JPanel {

    /** version number for serialization */
    private static final long serialVersionUID = 5938339262415238099L;

    /** gradient with one cycle */
    private final transient Gradient one;

    /** gradient with two cycles */
    private final transient Gradient two;

    /** gradient with three cycles */
    private final transient Gradient three;

    /** gradient with four cycles */
    private final transient Gradient four;

    /** gradient with five cycles */
    private final transient Gradient five;

    /** gradient with six cycles */
    private final transient Gradient six;

    /**
     * Constructs a new <code>GradientPanel</code>.
     */
    public GradientPanel() {

        super();

        setBackground(Color.BLACK);
        setPreferredSize(new Dimension(1024, 600));

        this.one = new Gradient(1024, 1);
        this.two = new Gradient(1024, 2);
        this.three = new Gradient(1024, 3);
        this.four = new Gradient(1024, 4);
        this.five = new Gradient(1024, 5);
        this.six = new Gradient(1024, 6);
    }

    /**
     * Draws the panel.
     *
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    @Override public void paintComponent(final Graphics grx) {

        super.paintComponent(grx);

        for (int i = 0; i < 1024; i++) {
            grx.setColor(this.one.getColor(i));
            grx.drawLine(i, 10, i, 80);
            grx.setColor(this.two.getColor(i));
            grx.drawLine(i, 110, i, 180);
            grx.setColor(this.three.getColor(i));
            grx.drawLine(i, 210, i, 280);
            grx.setColor(this.four.getColor(i));
            grx.drawLine(i, 310, i, 380);
```

```
            grx.setColor(this.five.getColor(i));
            grx.drawLine(i, 410, i, 480);
            grx.setColor(this.six.getColor(i));
            grx.drawLine(i, 510, i, 580);
        }
    }

    /**
     * Main method to display the panel.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        GradientPanel panel;
        JFrame frame;

        panel = new GradientPanel();

        frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setContentPane(panel);
        frame.pack();
        frame.setLocation(200, 0);
        frame.setVisible(true);
    }
}

package com.srbenoit.color;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.Toolkit;
import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import javax.swing.JFrame;
import com.srbenoit.log.LoggedPanel;
import com.srbenoit.math.linear.DMatrix;

/**
 * Class to load several matrix files, determines the entire range of values, and creates a single
 * color gradient scale for all the images.
 */
public class ScaleUnifier extends LoggedPanel {

    /** version number for serialization */
    private static final long serialVersionUID = 7818251645492851675L;

    /** the list of loaded matrices */
    private final transient DMatrix[] matrices;

    /** screen locations at which to draw each matrix */
    private final transient Point[] locations;

    /** the color gradient */
    private final transient Gradient gradient;

    /** the maximum values for each color range */
    private final transient double[] range;

    /**
     * Constructs a new <code>ScaleUnifier</code>.
     *
     * @param  numColors  the number of colors in the scale
     * @param  numCycles  the number of cycles in the scale
     * @param  files      the list of matrix data files to process
     * @throws  IOException  if there is an error reading a matrix data file
     */
    public ScaleUnifier(final int numColors, final int numCycles, final File[] files)
        throws IOException {

        super();

        this.gradient = new Gradient(numColors, numCycles);
        this.range = new double[numColors];
        this.matrices = new DMatrix[files.length];
        this.locations = new Point[files.length];

        for (int i = 0; i < files.length; i++) {
            this.matrices[i] = DMatrix.load(files[i]);
        }
    }

    /**
```

```
     * Builds a single scale for the data.
     *
     * @param  logBase  the base with which to take logarithms (0 for no logs)
     */
    public void makeScale(final double logBase) {

        if (logBase < 0) {
            makeSmoothScale();
        } else if (logBase == 0) {
            makeLinearScale();
        } else {
            makeLogScale(logBase);
        }
    }

    /**
     * Builds a linear scale for the data.
     */
    private void makeSmoothScale() {

        int size = 0;
        int index = 0;
        DMatrix matrix;
        double[] ranges;

        // Form a combined matrix with all values from the given matrices
        for (DMatrix mat : this.matrices) {
            size += mat.numColumns() * mat.numRows();
        }

        matrix = new DMatrix(1, size);

        for (DMatrix mat : this.matrices) {

            for (int c = 0; c < mat.numColumns(); c++) {

                for (int r = 0; r < mat.numRows(); r++) {
                    matrix.set(0, index, mat.get(r, c));
                    index++;
                }
            }
        }

        // Have the matrix generate the gradient.
        ranges = matrix.colorRanges(this.range.length);
        System.arraycopy(ranges, 0, this.range, 0, this.range.length);
    }

    /**
     * Builds a linear scale for the data.
     */
    private void makeLinearScale() {

        int width;
        int height;
        double min;
        double max;
        double cur;

        // Compute the minimum/maximum values
        min = this.matrices[0].get(0, 0);
        max = min;

        for (int i = 0; i < this.matrices.length; i++) {
            width = this.matrices[i].numColumns();
            height = this.matrices[i].numRows();

            for (int c = 0; c < width; c++) {

                for (int r = 0; r < height; r++) {
                    cur = this.matrices[i].get(r, c);

                    if (cur < min) {
                        min = cur;
                    }

                    if (cur > max) {
                        max = cur;
                    }
                }
            }
        }

        LOG.log(Level.FINE, "Overall range: {0} to {1}", new Object[] { min, max });

        // Now, create a linear gradient of those values.
        for (int i = 0; i < this.range.length; i++) {
```

```
            this.range[i] = min + ((max - min) * i / this.range.length);
        }
    }

    /**
     * Builds a logarithmic scale for the data.
     *
     * @param  logBase  the base with which to take logarithms, using log_a(x) = log(x) / log(a)
     */
    private void makeLogScale(final double logBase) {

        double loga;
        int width;
        int height;
        double min;
        double max;
        double cur;

        loga = Math.log10(logBase);

        // Compute the minimum/maximum values
        cur = Math.log10(this.matrices[0].get(0, 0)) / loga;
        min = cur;
        max = cur;

        for (int i = 0; i < this.matrices.length; i++) {
            width = this.matrices[i].numColumns();
            height = this.matrices[i].numRows();

            for (int c = 0; c < width; c++) {

                for (int r = 0; r < height; r++) {
                    cur = Math.log10(this.matrices[i].get(r, c)) / loga;

                    if (cur < min) {
                        min = cur;
                    }

                    if (cur > max) {
                        max = cur;
                    }
                }
            }
        }

        LOG.log(Level.FINE, "Overall range: {0} to {1}", new Object[] { min, max });

        // Now, create a linear gradient of the logarithm values.
        for (int i = 0; i < this.range.length; i++) {
            cur = min + ((max - min) * i / this.range.length);
            this.range[i] = Math.pow(logBase, cur);
        }
    }

    /**
     * Draws the resulting images.
     */
    private void draw() {

        Dimension screen;
        int xCoord;
        int yCoord;
        int rowHeight;
        int maxWidth;
        JFrame frame;

        screen = Toolkit.getDefaultToolkit().getScreenSize();

        // Compute our size based on sizes of matrices, and compute the draw
        // location of each matrix.
        xCoord = 0;
        yCoord = 0;
        rowHeight = 0;
        maxWidth = 0;

        for (int i = 0; i < this.matrices.length; i++) {

            if ((xCoord + this.matrices[i].numColumns()) > screen.width) {
                xCoord = 0;
                yCoord += rowHeight;
                rowHeight = 0;
            }

            this.locations[i] = new Point(xCoord, yCoord); // NOPMD SRB
            xCoord += this.matrices[i].numColumns();

            if (xCoord > maxWidth) {
```

```
            maxWidth = xCoord;
        }

        if (this.matrices[i].numRows() > rowHeight) {
            rowHeight = this.matrices[i].numRows();
        }
    }

    yCoord += rowHeight;

    setPreferredSize(new Dimension(maxWidth, yCoord));
    setBackground(Color.BLACK);

    // Build a frame and set ourselves as the content panel
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setUndecorated(true);
    frame.setContentPane(this);
    frame.pack();
    frame.setVisible(true);
}

/**
 * Paints the panel.
 *
 * @param  grx  the <code>Graphics</code> to which to draw
 */
@Override public void paintComponent(final Graphics grx) {

    Point loc;
    int width;
    int height;
    double cur;
    int color;

    super.paintComponent(grx);

    for (int i = 0; i < this.matrices.length; i++) {
        loc = this.locations[i];
        width = this.matrices[i].numColumns();
        height = this.matrices[i].numRows();

        for (int c = 0; c < width; c++) {

            for (int r = 0; r < height; r++) {
                cur = this.matrices[i].get(r, c);
                color = getColor(cur);
                grx.setColor(this.gradient.getColor(color));
                grx.drawLine(loc.x + c, loc.y + r, loc.x + c, loc.y + r);
            }
        }
    }
}

/**
 * Use the computed ranges to convert a value to the corresponding color.
 *
 * @param   value  the value
 * @return  the color index
 */
private int getColor(final double value) {

    int color;

    color = this.range.length - 1;

    for (int i = 0; i < this.range.length; i++) {

        if (this.range[i] > value) {
            color = i;

            break;
        }
    }

    return color;
}

/**
 * Main method to run the program.
 *
 * @param  args  command-line arguments
 */
public static void main(final String... args) {

    File[] files;
    ScaleUnifier unifier;
```

```
        files = new File[] {
                new File("/imp/radiusPitchLandscape-1.m3d"),
                new File("/imp/radiusPitchLandscape-2.m3d"),
                new File("/imp/radiusPitchLandscape-3.m3d")
            };

        try {
            unifier = new ScaleUnifier(4096, 1, files);
            unifier.makeScale(-1);
            unifier.draw();
        } catch (IOException e) {
            LOG.throwing("ScaleUnifier", "main", e);
        }
    }
}
```

```
package com.srbenoit.color;

import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import com.srbenoit.font.BundledFontManager;

/**
 * An image containing color swatches with optional color names, that maintains a bounding
 * rectangle for each color and translates points to the color hit.
 */
public class SwatchImage {

    /** the width of color boxes */
    private final transient int boxWidth;

    /** the height of color boxes */
    private final transient int boxHeight;

    /** the vertical spacing between color boxes */
    private final transient int boxSpacing;

    /** <code>true</code> to include color names */
    private final transient boolean incNames;

    /** <code>true</code> to include color hex values */
    private final transient boolean incHex;

    /** <code>true</code> to include borders around boxes */
    private final transient boolean incBorders;

    /** the hit-rectangles of each color box, 4 int's per box, x, y, w, h */
    private transient int[] boxes = null;

    /** the index of the color that has been selected */
    private transient int selected = -1;

    /** an offscreen image with the color boxes & names */
    private transient BufferedImage img = null;

    /** the font to use for drawing color names */
    private transient Font font = null;

    /**
     * Constructs a new <code>SwatchImage</code> with particular settings.
     *
     * @param  swatchWidth     the width of boxes
     * @param  swatchHeight    the height of boxes
     * @param  swatchSpacing   the number of pixels between boxes
     * @param  includeNames    <code>true</code> to include names; <code>false</code> otherwise
     * @param  includeHex      <code>true</code> to include hex values; <code>false</code>
     *                         otherwise
     * @param  includeBorders  <code>true</code> to include black borders around each box; <code>
     *                         false</code> otherwise
     */
    public SwatchImage(final int swatchWidth, final int swatchHeight, final int swatchSpacing,
        final boolean includeNames, final boolean includeHex, final boolean includeBorders) {

        this.boxWidth = swatchWidth;
        this.boxHeight = swatchHeight;
        this.boxSpacing = swatchSpacing;
        this.incNames = includeNames;
        this.incHex = includeHex;
        this.incBorders = includeBorders;

        if (this.incNames || this.incHex) {
            constructImageWithLabels();
```

```
        } else {
            constructImageWithoutLabels();
        }

        drawImage();
    }

    /**
     * Constructs the swatch image with labels on swatches.
     */
    private void constructImageWithLabels() {

        int count;
        int deltay;
        int deltax;
        FontMetrics metrics;
        int max;
        int inx;
        String str;
        int width;
        int height;

        count = ColorNames.getInstance().getNumColors();

        // Get the grid spacing for boxes
        deltax = this.boxWidth + this.boxSpacing;
        deltay = this.boxHeight + this.boxSpacing;

        // this.mRects = new Rectangle[count];
        this.boxes = new int[count * 4];

        // Make a font that will fit the vertical spacing
        metrics = chooseFont(deltay);

        // Compute rectangle positions, in 3 columns, tracking maximum
        // width of mName text. The X position and width of the rectangles
        // are left 0 for now since max sizes are not known.
        max = 0;

        for (inx = 0; inx < count; inx++) {

            if (this.incNames) {

                if (this.incHex) {
                    str = ColorNames.getInstance().getColorName(inx) + "\u00a0(FFFFFF)";
                } else {
                    str = ColorNames.getInstance().getColorName(inx);
                }
            } else {
                str = "FFFFFF";
            }

            width = metrics.stringWidth(str);

            if (max < width) {
                max = width;
            }

            this.boxes[inx * 4] = 0;
            this.boxes[(inx * 4) + 1] = this.boxSpacing + (deltay * (inx / 3));
            this.boxes[(inx * 4) + 2] = 0;
            this.boxes[(inx * 4) + 3] = deltay;
        }

        // Now we know max text size, so we size columns and set X values
        width = deltax + 2 + max + 4; // Box + space + name + big space

        for (inx = 0; inx < count; inx++) {
            this.boxes[inx * 4] = this.boxSpacing + (width * (inx % 3));
            this.boxes[(inx * 4) + 2] = width - 4;
        }

        width *= 3;
        height = this.boxes[((count - 1) * 4) + 1] + deltay + this.boxSpacing;

        // Now we can allocate the offscreen buffer and paint it.
        this.img = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
    }

    /**
     * Determines the largest font for which the ascent + descent is no greater than some delta
     * value.
     *
     * @param   height  the maximum font ascent + descent
     * @return  the best-fit font
     */
    private FontMetrics chooseFont(final int height) {
```

```
        BundledFontManager bfm;
        FontMetrics metrics;

        // Make a font that will fit the vertical spacing
        bfm = BundledFontManager.getInstance();

        this.font = bfm.getFont(BundledFontManager.SANS, height + 5, Font.PLAIN);
        metrics = bfm.getFontMetrics(this.font);

        if ((metrics.getAscent() + metrics.getDescent()) > height) {

            for (int size = height + 4; size > 1; size--) {
                this.font = bfm.getFont(BundledFontManager.SANS, size, Font.PLAIN);
                metrics = bfm.getFontMetrics(this.font);

                if ((metrics.getAscent() + metrics.getDescent()) <= height) {
                    break;
                }
            }
        }

        return metrics;
    }

    /**
     * Constructs the swatch image without labels on swatches.
     */
    private void constructImageWithoutLabels() {

        int count;
        int deltay;
        int deltax;
        int inx;
        int width;
        int height;

        count = ColorNames.getInstance().getNumColors();

        // Get the grid spacing for boxes
        deltax = this.boxWidth + this.boxSpacing;
        deltay = this.boxHeight + this.boxSpacing;

        // this.mRects = new Rectangle[count];
        this.boxes = new int[count * 4];

        // Compute size of offscreen buffer for grid of boxes
        width = (deltax * 32) - this.boxSpacing;
        height = (deltay * ((count + 31) / 32)) - this.boxSpacing;

        for (inx = 0; inx < count; inx++) {
            this.boxes[inx * 4] = deltax * (inx % 32);
            this.boxes[(inx * 4) + 1] = deltay * (inx / 32);
            this.boxes[(inx * 4) + 2] = this.boxWidth;
            this.boxes[(inx * 4) + 3] = this.boxHeight;
        }

        // Now we can allocate the offscreen buffer and paint it.
        this.img = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
    }

    /**
     * Draws the offscreen image. The offscreen image holds the complete display of color boxes and
     * names, and is drawn to the window as needed when the user scrolls through the colors, to
     * avoid having to do the time-consuming computation for positioning and font choice each time
     * the window scrolls.
     */
    private void drawImage() {

        Graphics2D g2d;

        g2d = (Graphics2D) (this.img.getGraphics());
        g2d.setFont(this.font);
        g2d.setColor(Color.WHITE);
        g2d.fillRect(0, 0, this.img.getWidth(), this.img.getHeight());

        if ((this.incNames) || (this.incHex)) {
            drawImageWithLabels(g2d);
        } else {
            drawImageWithoutLabels(g2d);
        }
    }

    /**
     * Draw the offscreen image without any swatch labels.
     *
     * @param  g2d  the <code>Graphics2D</code> to which to draw
```

```
     */
    private void drawImageWithoutLabels(final Graphics2D g2d) {

        for (int i = 0; i < ColorNames.getInstance().getNumColors(); i++) {

            // If selected, highlight the item. If there is spacing between
            // boxes, use that to show the selection. If not, we will
            // change the border color when the border is drawn.
            if ((this.selected == i) && (this.boxSpacing > 0)) {
                g2d.setColor(Color.RED);
                g2d.fillRect(this.boxes[i * 4] - this.boxSpacing,
                    this.boxes[(i * 4) + 1] - this.boxSpacing,
                    this.boxWidth + (this.boxSpacing * 2), this.boxHeight + (this.boxSpacing * 2));
            }

            g2d.setColor(ColorNames.getInstance().getColor(i));
            g2d.fillRect(this.boxes[i * 4], this.boxes[(i * 4) + 1], this.boxWidth,
                this.boxHeight);

            if (this.incBorders) {

                if ((this.boxSpacing == 0) && (this.selected == i)) {
                    g2d.setColor(Color.RED);
                    g2d.drawRect(this.boxes[i * 4] + 1, this.boxes[(i * 4) + 1] + 1,
                        this.boxWidth - 3, this.boxHeight - 3);
                } else {
                    g2d.setColor(Color.BLACK);
                }

                g2d.drawRect(this.boxes[i * 4], this.boxes[(i * 4) + 1], this.boxWidth - 1,
                    this.boxHeight - 1);
            }
        }
    }

    /**
     * Draws the offscreen image with swatch labels.
     *
     * @param  g2d  the <code>Graphics2D</code> to which to draw
     */
    private void drawImageWithLabels(final Graphics2D g2d) {

        for (int i = 0; i < ColorNames.getInstance().getNumColors(); i++) {

            g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

            // If selected, highlight the item
            if (this.selected == i) {
                g2d.setColor(Color.YELLOW);
                g2d.fillRect(this.boxes[i * 4] - this.boxSpacing,
                    this.boxes[(i * 4) + 1] - this.boxSpacing,
                    this.boxes[(i * 4) + 2] + this.boxSpacing,
                    this.boxes[(i * 4) + 3] + this.boxSpacing);
            }

            g2d.setColor(ColorNames.getInstance().getColor(i));
            g2d.fillRect(this.boxes[i * 4], this.boxes[(i * 4) + 1], this.boxWidth,
                this.boxHeight);
            g2d.setColor(Color.BLACK);

            if (this.incBorders) {
                g2d.drawRect(this.boxes[i * 4], this.boxes[(i * 4) + 1], this.boxWidth - 1,
                    this.boxHeight - 1);
            }

            g2d.drawString(makeLabel(i), this.boxes[i * 4] + this.boxWidth + 2,
                this.boxes[(i * 4) + 1] + g2d.getFontMetrics().getAscent());
        }
    }

    /**
     * Constructs the text label for a color swatch.
     *
     * @param   index  the color index
     * @return  the label
     */
    private String makeLabel(final int index) {

        String lbl;

        if (this.incNames) {

            if (this.incHex) {
                lbl = ColorNames.getInstance().getColorName(index) + "(" + colorHex(index) + ")";
            } else {
                lbl = ColorNames.getInstance().getColorName(index);
```

```
            }
        } else {
            lbl = colorHex(index);
        }

        return lbl;
    }

    /**
     * Generates the hex value for a color.
     *
     * @param   index  the index of the color
     * @return  the hex representation
     */
    private static String colorHex(final int index) {

        return (Integer.toHexString(ColorNames.getInstance().getRed(index) / 16)
                + Integer.toHexString(ColorNames.getInstance().getRed(index) % 16)
                + Integer.toHexString(ColorNames.getInstance().getGreen(index) / 16)
                + Integer.toHexString(ColorNames.getInstance().getGreen(index) % 16)
                + Integer.toHexString(ColorNames.getInstance().getBlue(index) / 16)
                + Integer.toHexString(ColorNames.getInstance().getBlue(index) % 16));
    }

    /**
     * Gets the selected color index, which is an index into the <code>ColorNames.COLORNAMES</code>
     * array of color names.
     *
     * @return  the selected color index or -1 if nothing selected
     */
    public int getSelected() {

        return this.selected;
    }

    /**
     * Sets the selected color by index.
     *
     * @param  index  the index of the selected color, which is an index into the <code>
     *                ColorNames.CNAMES</code> array of color names, or -1 to indicate no selection
     */
    public final void setSelected(final int index) {

        this.selected = index;

        if (this.img != null) {
            drawImage();
        }
    }

    /**
     * Gets the image width.
     *
     * @return  the width
     */
    public int getWidth() {

        return this.img.getWidth();
    }

    /**
     * Gets the image height.
     *
     * @return  the height
     */
    public int getHeight() {

        return this.img.getHeight();
    }

    /**
     * Gets the buffered image.
     *
     * @return  the image
     */
    public BufferedImage getImage() {

        return this.img;
    }

    /**
     * Determines which bounding box a point hits, if any.
     *
     * @param   xPos  the point X coordinate
     * @param   yPos  the point Y coordinate
     * @return  the index of the box hit; or -1 if no box was hit
     */
```

```
    public int getHitIndex(final int xPos, final int yPos) {

        int count;
        int inx;
        int hit = -1;

        count = this.boxes.length / 4;

        for (inx = 0; inx < count; inx++) {

            if ((this.boxes[inx * 4] <= xPos) && (this.boxes[(inx * 4) + 1] <= yPos)
                    && ((this.boxes[inx * 4] + this.boxes[(inx * 4) + 2]) >= xPos)
                    && ((this.boxes[(inx * 4) + 1] + this.boxes[(inx * 4) + 3]) >= yPos)) {
                hit = inx;

                break;
            }
        }

        return hit;
    }
}
```

E.1.7 Font Management (com.srbenoit.font)

This package provides tools to manage and view the installed fonts, and to use fonts from a bundle included in a JAR file.

```
package com.srbenoit.font;

import java.awt.Font;
import java.awt.FontFormatException;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.util.ResourceLoader;

/**
 * Provides font management for a packaged set of fonts. This allows fonts to be bundled with an
 * application, without dependence on a set of fonts being installed on a client machine.
 */
public final class BundledFontManager extends LoggedObject {

    /** object on which to synchronize static instance creation */
    private static final Object SYNCH = new Object();

    /** predefined font name for the default serif font */
    public static final String SERIF = "Times New Roman";

    /** predefined font name for the default serif font */
    public static final String SERIF_PREFIX = "Times";

    /** predefined font name for the default sans serif font */
    public static final String SANS = "Arial";

    /** predefined font name for the default sans serif font */
    public static final String SANS_PREFIX = "Arial";

    /** the singleton instance of the font manager */
    private static BundledFontManager instance = null;

    /** the <code>Graphics</code> of the offscreen buffered image */
    private transient Graphics grx;

    /** a map of 1-point fonts as read from the font directory */
    private final transient Map<String, Font> fonts;

    /** storage for error messages generated by the font manager */
    private final transient List<String> reasons;
```

```
/** the names of the installed fonts */
private transient String[] names = null;

/**
 * Constructs a <code>BundledFontManager</code> object.
 */
private BundledFontManager() {

    BufferedImage image;

    this.fonts = new HashMap<String, Font>(20);

    image = new BufferedImage(1, 1, BufferedImage.TYPE_INT_RGB);
    this.grx = image.getGraphics();
    this.reasons = new ArrayList<String>(20);
}

/**
 * Retrieves the singleton instance of the bundled font manager, creating it if it does not yet
 * exist.
 *
 * @return  the <code>BundledFontManager</code> instance
 */
public static BundledFontManager getInstance() {

    synchronized (SYNCH) {

        if (instance == null) {
            instance = new BundledFontManager();
            instance.scanFonts();
        }
    }

    return instance;
}

/**
 * Scans the font directory, importing all fonts found.
 */
private void scanFonts() {

    try {
        addFonts("minfonts2.list");
    } catch (IOException e) {
        logError(e.getLocalizedMessage());
    }
}

/**
 * Given the name of a file that contains a list of fonts, tries loading each font name found in
 * that file.
 *
 * @param   listFileName  the name of the file containing the font name list. This file will be
 *                        loaded as a resource, so it should be in a JAR that is in the
 *                        CLASSPATH
 * @throws  IOException   if there is an error reading the font list file or a font
 */
private void addFonts(final String listFileName) throws IOException {

    Font onePoint;
    InputStream input;
    String[] fontsList;
    String fontName;
    String name;

    fontsList = ResourceLoader.loadFileLines(BundledFontManager.class, listFileName);

    if (fontsList == null) {
        logError(listFileName + "_not_found");
    } else {

        for (int i = 0; i < fontsList.length; i++) {
            fontName = fontsList[i];

            input = ResourceLoader.getInputStream(BundledFontManager.class, fontName);

            if (input != null) {

                if (fontName.toLowerCase(Locale.getDefault()).endsWith(".ttf")) {

                    try {
                        onePoint = Font.createFont(Font.TRUETYPE_FONT, input);
                        name = onePoint.getFontName();

                        if (name.startsWith(SANS_PREFIX)) {
                            name = SANS;
```

```
                        } else if (name.startsWith(SERIF_PREFIX)) {
                            name = SERIF;
                        }

                        this.fonts.put(name, onePoint);
                    } catch (FontFormatException e) {
                        logError(e.getLocalizedMessage());
                    }
                } else if ((fontName.toLowerCase(Locale.getDefault()).endsWith(".pfa"))
                        || (fontName.toLowerCase(Locale.getDefault()).endsWith(".pfb"))) {

                    try {
                        onePoint = Font.createFont(Font.TYPE1_FONT, input);
                        name = onePoint.getFontName();

                        if (name.startsWith(SANS_PREFIX)) {
                            name = SANS;
                        } else if (name.startsWith(SERIF_PREFIX)) {
                            name = SERIF;
                        }

                        this.fonts.put(name, onePoint);
                    } catch (FontFormatException e) {
                        logError(e.getLocalizedMessage());
                    }
                }

                input.close();
            }
        }
    }
}

/**
 * Tests whether a font name is valid.
 *
 * @param   name  the name of the font to test
 * @return  <code>true</code> if the name is valid, <code>false</code> otherwise
 */
public boolean isFontNameValid(final String name) {

    boolean valid;

    if ("SERIF".equals(name) || "SANS".equals(name) || "MONOSPACE".equals(name)) {
        valid = true;
    } else {

        synchronized (this) {
            valid = this.fonts.containsKey(name);
        }
    }

    return valid;
}

/**
 * Generates a list of names of the installed fonts.
 *
 * @return  the list of font names
 */
public String[] fontNames() {

    int inx = 0;
    String[] list;

    synchronized (this) {

        if (this.names == null) {
            this.names = new String[this.fonts.size()];

            for (String elem : this.fonts.keySet()) {
                this.names[inx] = elem;
                inx++;
            }

            Arrays.sort(this.names);
        }

        list = new String[this.names.length];
        System.arraycopy(this.names, 0, list, 0, list.length);
    }

    return list;
}

/**
 * Retrieves a particular font, in a particular size and style.
```

```
     *
     * @param   spec  the specification of the font to retrieve
     * @return  the generated font, or <code>null</code> if the name is not valid
     */
    public Font getFont(final FontSpec spec) {

        return getFont(spec.getFontName(), spec.getFontSize(), spec.getFontStyle());
    }

    /**
     * Retrieves a particular font, in a particular size and style.
     *
     * @param   name   the name of the font face to retrieve
     * @param   size   the point size to retrieve
     * @param   style  the style, as defined in the <code>Font</code> class
     * @return  the generated font, or <code>null</code> if the name is not valid
     */
    public Font getFont(final String name, final float size, final int style) {

        String actual;
        Font onePoint;
        Font derived;
        int style2;

        if ("SANS".equals(name)) {
            actual = SANS;
        } else if ("SERIF".equals(name)) {
            actual = SERIF;
        } else {
            actual = name;
        }

        style2 = style & (Font.BOLD | Font.ITALIC);

        onePoint = this.fonts.get(actual);

        if (onePoint == null) {

            // Emergency fall back
            derived = new Font(SANS, style2, (int) size);
        } else {
            derived = onePoint.deriveFont(style2, size);
        }

        return derived;
    }

    /**
     * Sets the internal <code>Graphics</code> used to generate font metrics.
     *
     * @param  graphics  the <code>Graphics</code> object
     */
    public void setGraphics(final Graphics graphics) {

        this.grx = graphics;
    }

    /**
     * Retrieves a font metrics object for a font, using the <code>Graphics</code> that is
     * associated with the offscreen image.
     *
     * @param   font  the font for which to get metrics
     * @return  the metrics for the font
     */
    public FontMetrics getFontMetrics(final Font font) {

        return this.grx.getFontMetrics(font);
    }

    /**
     * Adds an error message to the error log.
     *
     * @param  err  the error message
     */
    public void logError(final String err) {

        LOG.warning(err);
        this.reasons.add(err);
    }

    /**
     * Gets the list of errors that have been encountered by the font manager since it was
     * instantiated.
     *
     * @return  an array of <code>String</code> error messages
     */
    public String[] errors() {
```

```
        String[] result;

        result = new String[this.reasons.size()];
        this.reasons.toArray(result);

        return result;
    }

    /**
     * Main method for testing.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        BundledFontManager obj;
        String[] names;
        int inx;

        obj = BundledFontManager.getInstance();
        names = obj.fontNames();

        for (inx = 0; inx < names.length; inx++) {
            LOG.info(obj.getFont(names[inx], 15.0f, Font.BOLD).getName());
        }
    }
}

package com.srbenoit.font;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.JTextArea;
import javax.swing.SwingConstants;

/**
 * A frame that displays a tabbed pane with one tab per bundled font. Within each tab are the
 * font's glyphs in various sizes.
 */
public class BundledFontViewer extends JFrame {

    /** version number for serialization */
    private static final long serialVersionUID = 4624550107266295327L;

    /** the font sizes to view */
    private static final int[] SIZES = { 10, 12, 14, 18, 24 };

    /**
     * Constructs a new <code>BundledFontViewer</code>.
     */
    public BundledFontViewer() {

        super("Bundled Fonts Viewer");

        setBackground(new Color(220, 220, 255));
    }

    /**
     * Generates the user interface, which consists of a tabbed pane where each tab is a different
     * font, and within the tab is a set of text boxes showing the font's glyphs in various sizes.
     *
     * @param  mgr  the font manager used to obtain the fonts
     */
    public void createUI(final BundledFontManager mgr) {

        JTabbedPane tabs;
        String[] names;
        JPanel[] panes;
        int inx;

        names = mgr.fontNames();

        tabs = new JTabbedPane(SwingConstants.BOTTOM);
        tabs.setOpaque(true);

        tabs.setBackground(new Color(220, 220, 255));
        tabs.setPreferredSize(new Dimension(740, 500));
        tabs.setFont(new Font("Dialog", Font.PLAIN, 10));
        setContentPane(tabs);
```

```
        panes = new JPanel[names.length];

        for (inx = 0; inx < panes.length; inx++) {
            panes[inx] = new JPanel(); // NOPMD SRB
            tabs.addTab(names[inx], panes[inx]);
            populateTab(mgr, names[inx], panes[inx]);
        }
    }

    /**
     * Generates the contents of a font tab, including text boxes that use the font at varying
     * sizes.
     *
     * @param  mgr   the font manager from which to retrieve fonts
     * @param  name  the name of the font
     * @param  pane  the panel in which to install the text boxes
     */
    private void populateTab(final BundledFontManager mgr, final String name, final JPanel pane) {

        JTextArea area;
        Font fnt;
        int inx;
        int num;
        StringBuilder buf;
        int cnt;
        char chr;

        pane.setLayout(new GridLayout(SIZES.length, 1, 5, 5));
        pane.setBackground(new Color(220, 220, 220));
        pane.setBorder(BorderFactory.createEtchedBorder());

        buf = new StringBuilder(500);

        for (inx = 0; inx < SIZES.length; inx++) {
            area = new JTextArea(); // NOPMD SRB
            fnt = mgr.getFont(name, SIZES[inx], Font.PLAIN);
            area.setFont(fnt);

            // Determine the font glyphs and set that in each text area.
            num = fnt.getNumGlyphs();
            cnt = 0;
            chr = 1;

            buf.setLength(0);

            while ((cnt < num) && (chr < Character.MAX_VALUE)) {

                if (fnt.canDisplay(chr)) {
                    buf.append(chr);
                    cnt++;

                    if ((cnt % ((num / 5) + 1)) == 0) {
                        buf.append('\n');
                    }
                }

                chr++;
            }

            area.setText(buf.toString());
            area.setWrapStyleWord(true);

            pane.add(area);
        }
    }

    /**
     * An implementation of main for testing.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        BundledFontViewer obj;
        BundledFontManager mgr;

        obj = new BundledFontViewer();
        mgr = BundledFontManager.getInstance();

        obj.createUI(mgr);
        obj.pack();
        obj.setVisible(true);
        obj.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

package com.srbenoit.font;
```

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.InputEvent;
import java.io.File;
import java.io.FileInputStream;
import java.util.logging.Level;
import javax.swing.BorderFactory;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.KeyStroke;
import com.srbenoit.log.LoggedObject;

/**
 * A simple application allowing the user to select a TrueType (.TTF) or Type-1 (.PFA/.PFB) font
 * file then displaying a sample of that font's symbols.
 */
public class FontBrowser extends LoggedObject implements ActionListener {

    /** the frame */
    private final transient JFrame frame;

    /** the content panel of the frame */
    private final transient JPanel content;

    /** the currently loaded (one-point) font */
    private transient Font font = null;

    /** the most recently opened font file's parent directory */
    private transient File dir = null;

    /**
     * Creates a new <code>FontBrowser</code>.
     */
    public FontBrowser() {

        JMenuBar bar;
        JMenu menu;
        JMenuItem item;
        FontPanel pnl;

        this.frame = new JFrame("Font Browser");

        bar = new JMenuBar();
        menu = new JMenu("File");
        item = new JMenuItem("Open");

        this.content = new JPanel(new BorderLayout());
        this.frame.setContentPane(this.content);

        item.addActionListener(this);
        item.setAccelerator(KeyStroke.getKeyStroke('O', InputEvent.CTRL_DOWN_MASK));
        menu.add(item);
        bar.add(menu);
        this.frame.setJMenuBar(bar);

        this.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pnl = new FontPanel(this.font);
        this.content.add(pnl, BorderLayout.CENTER);
        this.frame.pack();
        this.frame.setVisible(true);
    }

    /**
     * Gets the frame.
     *
     * @return  the frame
     */
    public JFrame getFrame() {

        return this.frame;
    }

    /**
     * Handles action events.
     *
     * @param  evt  the action event to be processed
```

```
 */
public void actionPerformed(final ActionEvent evt) {

    String cmd;
    JFileChooser jfc;
    File file;
    FileInputStream fis;
    FontPanel pnl;
    String name;

    cmd = evt.getActionCommand();

    if ("Open".equals(cmd)) {
        jfc = new JFileChooser();

        if (this.dir != null) {
            jfc.setCurrentDirectory(this.dir);
        }

        if (jfc.showOpenDialog(this.frame) == JFileChooser.APPROVE_OPTION) {
            file = jfc.getSelectedFile();
            this.dir = file.getParentFile();
            name = file.getName().toLowerCase();

            try {
                fis = new FileInputStream(file);

                if ((name.endsWith(".ttf")) || (name.endsWith(".otf"))) {
                    this.font = Font.createFont(Font.TRUETYPE_FONT, fis);
                } else {
                    this.font = Font.createFont(Font.TYPE1_FONT, fis);
                }

                fis.close();

                this.content.removeAll();
                pnl = new FontPanel(this.font);
                this.content.add(pnl, BorderLayout.CENTER);
                this.frame.pack();

                this.frame.setTitle(file.getAbsolutePath());
            } catch (Exception exc) {
                LOG.log(Level.WARNING, "Failed to open font {0}: {1}",
                    new Object[] { name, exc.getLocalizedMessage() });
            }
        }
    }
}

/**
 * Main method to execute the application.
 *
 * @param  args  command-line arguments
 */
public static void main(final String... args) {

    new FontBrowser().getFrame();
}

/**
 * A panel that displays the selected font in varying sizes.
 */
private static class FontPanel extends JPanel {

    /** version number for serialization */
    private static final long serialVersionUID = -4052269402511508396L;

    /**
     * Construct a new <code>FontPanel</code>.
     *
     * @param  fnt  the font to display in the panel
     */
    protected FontPanel(final Font fnt) {

        super(new BorderLayout(5, 5));

        JPanel grid;
        Font derived;
        JLabel lbl;
        JTextArea area;

        setPreferredSize(new Dimension(800, 800));

        if (fnt != null) {

            // Build the UI.
            grid = new JPanel(new GridLayout(16, 1, 0, 0));
```

```
add(grid, BorderLayout.NORTH);

derived = fnt.deriveFont(Font.PLAIN, 24);
lbl = new JLabel(fnt.getFontName() + " in 24-point plain");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD, 24);
lbl = new JLabel(fnt.getFontName() + " in 24-point bold");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.ITALIC, 24);
lbl = new JLabel(fnt.getFontName() + " in 24-point italic");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD | Font.ITALIC, 24);
lbl = new JLabel(fnt.getFontName() + " in 24-point bold italic");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.PLAIN, 12);
lbl = new JLabel(fnt.getFontName() + " in 12-point plain");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD, 12);
lbl = new JLabel(fnt.getFontName() + " in 12-point bold");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.ITALIC, 12);
lbl = new JLabel(fnt.getFontName() + " in 12-point italic");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD | Font.ITALIC, 12);
lbl = new JLabel(fnt.getFontName() + " in 12-point bold italic");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD | Font.ITALIC, 12);
lbl = new JLabel("ABCDEFGHIJKLMNOPQRSTUVWXYZ" + "abcdefghijklmnopqrstuvwxyz");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD | Font.ITALIC, 12);
lbl = new JLabel("1234567890`~!@#$%^&*()-_=+[{]}\\|;" + ":'\",<.>/?");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.PLAIN, 9);
lbl = new JLabel(fnt.getFontName() + " in 9-point plain");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD, 9);
lbl = new JLabel(fnt.getFontName() + " in 9-point bold");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.ITALIC, 9);
lbl = new JLabel(fnt.getFontName() + " in 9-point italic");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD | Font.ITALIC, 9);
lbl = new JLabel(fnt.getFontName() + " in 9-point bold italic");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD | Font.ITALIC, 9);
lbl = new JLabel("ABCDEFGHIJKLMNOPQRSTUVWXYZ" + "abcdefghijklmnopqrstuvwxyz");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.BOLD | Font.ITALIC, 9);
lbl = new JLabel("1234567890`~!@#$%^&*()-_=+[{]}\\|;:'\",<.>/?");
lbl.setFont(derived);
grid.add(lbl);

derived = fnt.deriveFont(Font.PLAIN, 18);
area = new JTextArea("Type text here to try out the font.");
area.setBorder(BorderFactory.createLoweredBevelBorder());
area.setFont(derived);
add(area, BorderLayout.CENTER);
```

```
            }
        }
    }
}

package com.srbenoit.font;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.io.FileInputStream;
import java.io.InputStream;
import javax.swing.JFrame;
import javax.swing.JPanel;
import com.srbenoit.log.LoggedObject;

/**
 * A class to render some text.
 */
public class FontRender extends LoggedObject {

    /** the path of the font to load */
    private static final String PATH = "/cygwin/home/Steve Benoit/pace.ttf";

    /** a test string to print at small font size */
    private static final String TEST1 = "The quick brown fox jumps over the lazy dog.";

    /** the panel */
    private final transient FontRenderPanel panel;

    /** the font to render */
    private transient Font font;

    /**
     * Constructs a new <code>FontRender</code>.
     */
    public FontRender() {

        InputStream input;

        this.panel = new FontRenderPanel(this);
        this.panel.setPreferredSize(new Dimension(600, 600));
        this.panel.setBackground(Color.WHITE);

        try {
            input = new FileInputStream(PATH);

            try {
                this.font = Font.createFont(Font.TRUETYPE_FONT, input);
                input.close();
            } catch (Exception exc1) {
                LOG.warning("Failed to read " + PATH + ": " + exc1.getLocalizedMessage());
                this.font = new Font("Dialog", Font.PLAIN, 9);
            }
        } catch (Exception exc2) {
            LOG.warning("Failed to open " + PATH + ": " + exc2.getLocalizedMessage());
            this.font = new Font("Dialog", Font.PLAIN, 9);
        }
    }

    /**
     * Gets the panel.
     *
     * @return the panel
     */
    public JPanel getPanel() {

        return this.panel;
    }

    /**
     * Gets the font.
     *
     * @return the font
     */
    public Font getFont() {

        return this.font;
    }

    /**
     * Main method to create the panel and show it.
     *
     * @param args command-line arguments
```

```
 */
public static void main(final String... args) {

    JFrame frame;

    frame = new JFrame("Render Test");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(new FontRender().getPanel());
    frame.pack();
    frame.setLocation(600, 50);
    frame.setVisible(true);
}

/**
 * The panel.
 */
private static class FontRenderPanel extends JPanel {

    /** version number for serialization */
    private static final long serialVersionUID = 50159041185573735L;

    /** the owning <code>FontRender</code> object */
    private final FontRender owner;

    /**
     * Constructs a new <code>FontRenderPanel</code>.
     *
     * @param  owningFontRender  the owning <code>FontRender</code> object
     */
    protected FontRenderPanel(final FontRender owningFontRender) {

        this.owner = owningFontRender;
    }

    /**
     * Renders the panel.
     *
     * @param  grx  the <code>Graphics</code> to which to render
     */
    @Override public void paintComponent(final Graphics grx) {

        Graphics2D g2d;
        Font[] fonts;
        Object[] modes;
        String[] labels;
        int height;
        int yPos;

        g2d = (Graphics2D) grx;

        super.paintComponent(grx);

        // Gather the font sizes to use
        fonts = new Font[4];
        fonts[0] = this.owner.getFont().deriveFont(30.0f);
        fonts[1] = this.owner.getFont().deriveFont(20.0f);
        fonts[2] = this.owner.getFont().deriveFont(10.0f);
        fonts[3] = this.owner.getFont().deriveFont(8.0f);

        // Gather the rendering modes to use and their labels
        modes = new Object[7];
        labels = new String[7];

        modes[0] = RenderingHints.VALUE_TEXT_ANTIALIAS_OFF;
        modes[1] = RenderingHints.VALUE_TEXT_ANTIALIAS_ON;
        modes[2] = RenderingHints.VALUE_TEXT_ANTIALIAS_GASP;
        modes[3] = RenderingHints.VALUE_TEXT_ANTIALIAS_LCD_HBGR;
        modes[4] = RenderingHints.VALUE_TEXT_ANTIALIAS_LCD_HRGB;
        modes[5] = RenderingHints.VALUE_TEXT_ANTIALIAS_LCD_VBGR;
        modes[6] = RenderingHints.VALUE_TEXT_ANTIALIAS_LCD_VRGB;

        labels[0] = "Normal";
        labels[1] = "Antialias";
        labels[2] = "Gasp";
        labels[3] = "HBGR";
        labels[4] = "HRGB";
        labels[5] = "VBGR";
        labels[6] = "VRGB";

        height = grx.getFontMetrics(fonts[0]).getHeight();
        yPos = height;

        for (int i = 0; i < modes.length; i++) {
            g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING, modes[i]);
            drawTestString(g2d, fonts, labels[i], yPos);
            yPos += height;
        }
```

```
            g2d.setFont(fonts[fonts.length - 1]);
            height = g2d.getFontMetrics().getHeight();

            for (int i = 0; i < modes.length; i++) {
                g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING, modes[i]);

                g2d.drawString("This is a test of font rendering at eight points (" + labels[i]
                    + ")", 10, yPos);
                yPos += height;
            }

            g2d.setRenderingHint(RenderingHints.KEY_TEXT_LCD_CONTRAST, Integer.valueOf(100));
            g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_LCD_HRGB);

            for (int i = 0; i < fonts.length; i++) {

                g2d.setFont(fonts[i]);
                yPos += g2d.getFontMetrics().getAscent();
                g2d.drawString("!\"#$%&'()*+,-./ (Using HRGB)", 10, yPos);
                yPos += g2d.getFontMetrics().getDescent();
                yPos += g2d.getFontMetrics().getLeading();
            }
        }

        /**
         * Draws a test string in all font sizes.
         *
         * @param  g2d    the <code>Graphics2D</code> to which to draw
         * @param  fonts  the list of fonts to use to draw the string
         * @param  str    the string to draw
         * @param  yPos   the Y position at which to draw
         */
        private void drawTestString(final Graphics2D g2d, final Font[] fonts, final String str,
            final int yPos) {

            int xPos = 10;

            for (int i = 0; i < fonts.length; i++) {
                g2d.setFont(fonts[i]);
                g2d.drawString(str, xPos, yPos);
                xPos += 150;
            }

            g2d.drawString(TEST1, xPos, yPos);
        }
    }
}

package com.srbenoit.font;

import java.awt.Font;

/**
 * A specification for a font that can be obtained from the <code>BundledFontManager</code>.
 */
public final class FontSpec {

    /** the name of the font, or one of "SANS", "SERIF" or "MONOSPACE" */
    private String fontName = null;

    /** the point size of the font */
    private float fontSize = 1.0f;

    /** a scaling factor to apply to the font size */
    private float fontScale = 1.0f;

    /** the style of the font */
    private int fontStyle = Font.PLAIN;

    /**
     * Gets the font name.
     *
     * @return  the font name
     */
    public String getFontName() {

        return this.fontName;
    }

    /**
     * Sets the font name.
     *
     * @param  theFontName  the font name
     */
    public void setFontName(final String theFontName) {
```

```
        this.fontName = theFontName;
    }

    /**
     * Gets the font size.
     *
     * @return  the font size
     */
    public float getFontSize() {

        return this.fontSize;
    }

    /**
     * Sets the font size.
     *
     * @param  theFontSize  the font size
     */
    public void setFontSize(final float theFontSize) {

        this.fontSize = theFontSize;
    }

    /**
     * Gets the font scale.
     *
     * @return  the font scale
     */
    public float getFontScale() {

        return this.fontScale;
    }

    /**
     * Sets the font scale.
     *
     * @param  theFontScale  the font scale
     */
    public void setFontScale(final float theFontScale) {

        this.fontScale = theFontScale;
    }

    /**
     * Gets the font style.
     *
     * @return  the font style
     */
    public int getFontStyle() {

        return this.fontStyle;
    }

    /**
     * Sets the font style.
     *
     * @param  theFontStyle  the font style
     */
    public void setFontStyle(final int theFontStyle) {

        this.fontStyle = theFontStyle;
    }

    /**
     * Prints the font specification in the format of attributes in an XML tag.
     *
     * @param  builder  the <code>StringBuilder</code> to which to print
     * @param  prefix   a prefix to use on attributes. For example, if the prefix is 'foo', the
     *                  printed attributes will be 'foo-font', 'foo-size' and 'foo-style'
     */
    public void printAsAttributes(final StringBuilder builder, final String prefix) {

        if (this.fontName != null) {
            builder.append(" ");
            builder.append(prefix);
            builder.append("-font='");
            builder.append(this.fontName);
            builder.append("'");
        }

        builder.append(" ");
        builder.append(prefix);
        builder.append("-size='");
        builder.append(Float.toString(this.fontSize));
        builder.append("'");
```

```
        if (this.fontScale != 1.0f) {
            builder.append(" ");
            builder.append(prefix);
            builder.append("-scale='");
            builder.append(Float.toString(this.fontScale));
            builder.append("'");
        }

        builder.append(" ");
        builder.append(prefix);
        builder.append("-style='");

        if (this.fontStyle == Font.PLAIN) {
            builder.append("PLAIN");
        } else if (this.fontStyle == Font.BOLD) {
            builder.append("BOLD");
        } else if (this.fontStyle == Font.ITALIC) {
            builder.append("ITALIC");
        } else if (this.fontStyle == (Font.BOLD | Font.ITALIC)) {
            builder.append("BOLD,ITALIC");
        }

        builder.append("'");
    }
}

package com.srbenoit.font;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.font.FontRenderContext;
import java.awt.image.BufferedImage;
import java.io.File;
import java.util.Iterator;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import javax.imageio.ImageWriter;
import javax.imageio.stream.FileImageOutputStream;
import com.srbenoit.log.LoggedPanel;

/**
 * A panel that renders the glyphs of a font.
 */
public class GlyphPanel extends LoggedPanel {

    /** version number for serialization */
    private static final long serialVersionUID = -3511066856984634391L;

    /** the owning frame */
    private final transient ViewerInt ownerFrame;

    /** the font to be rendered */
    private transient Font font = null;

    /** an offscreen image that is rendered to */
    private transient BufferedImage offscreen = null;

    /** the width of the box for each glyph */
    private transient int boxWidth;

    /** the height of the box for each glyph */
    private transient int boxHeight;

    /** flag to control whether bounds are drawn around characters */
    private transient boolean drawBoxes = false;

    /** flag indicating offscreen image needs to be repainted */
    private transient boolean dirty = false;

    /**
     * Constructs a new <code>GlyphPanel</code>.
     *
     * @param  owner  the viewer that owns this panel
     */
    public GlyphPanel(final ViewerInt owner) {

        super();

        this.ownerFrame = owner;

        setPreferredSize(new Dimension(640, 480));
        setBackground(Color.white);
```

```
}

/**
 * Sets the font that the panel will render.
 *
 * @param  theFont  the font
 */
public void setTheFont(final Font theFont) {

    this.font = theFont;
    this.dirty = true;
}

/**
 * Sets the flag controlling whether or not bounds are drawn.
 *
 * @param  drawBoundsBoxes  <code>true</code> to draw bounds boxes; <code>false</code>
 *                          otherwise
 */
public void setDrawBoundsBoxes(final boolean drawBoundsBoxes) {

    this.drawBoxes = drawBoundsBoxes;
    this.dirty = true;
}

/**
 * Gets the state of the flag controlling whether or not bounds are drawn.
 *
 * @return  <code>true</code> if bounding boxes are being drawn; <code>false</code> otherwise
 */
public boolean isDrawingBoundsBoxes() {

    return this.drawBoxes;
}

/**
 * Redraws the panel. If this is the first time paint has been called since the font was
 * changed, the offscreen glyph image is created.
 *
 * @param  grx  the <code>Graphics</code> object to render to
 */
@Override public void paint(final Graphics grx) {

    grx.setColor(Color.white);
    grx.fillRect(0, 0, getWidth(), getHeight());

    if (this.dirty) {

        if (this.offscreen != null) {
            this.offscreen.flush();
        }

        buildOffscreen((Graphics2D) grx);
        this.dirty = false;
    }

    grx.drawImage(this.offscreen, 0, 0, this);
}

/**
 * Creates an offscreen image with the glyph renderings.
 *
 * @param  grx  the <code>Graphics2D</code> object to render to
 */
private void buildOffscreen(final Graphics2D grx) {

    char[] chars;
    Font labelFont;
    FontMetrics fmLabel;
    FontMetrics fmFont;
    int columns;
    int imgW;
    int imgH;
    int rows;
    Graphics2D g2d;

    // Get the characters the font supports
    chars = getFontSupportedChars(this.font);

    // Create a font to be used for labeling
    labelFont = new Font("Dialog", Font.PLAIN, 9);
    fmLabel = grx.getFontMetrics(labelFont);
    fmFont = grx.getFontMetrics(this.font);

    // Find box width based on max size of label or glyph
    this.boxWidth = fmLabel.stringWidth("9999");
```

```
        if (this.boxWidth < (int) (fmFont.getMaxCharBounds(grx).getWidth() + 0.9)) {
            this.boxWidth = (int) (fmFont.getMaxCharBounds(grx).getWidth() + 0.9);
        }

        this.boxWidth++; // add a pixel per box for left border

        // Using box width, and assuming right border, find boxes per row
        columns = 639 / this.boxWidth;

        // Compute total width of image, including borders
        imgW = (this.boxWidth * columns) + 1; // add a pixel for right border

        // Determine the number of rows
        rows = chars.length / columns; // full rows

        if ((rows * columns) < chars.length) {
            rows++; // partial row
        }

        // Compute box height. Note we don't need descent on label since
        // digits don't extend below baseline, but we add 1 pixel below,
        // along with 1 pixel for an interior border line.
        this.boxHeight = fmFont.getHeight() + fmLabel.getAscent();
        this.boxHeight += 2; // Add top border and border between glyph and

        // label
        imgH = (this.boxHeight * rows) + 1; // add a pixel for bottom border

        // Add space for a line for the font name
        imgH += fmFont.getHeight() + fmFont.getLeading();

        g2d = createOffscreen(imgW, imgH);
        drawGrid(g2d, fmFont);
        g2d.setFont(labelFont);
        drawLabels(g2d, chars, fmFont, fmLabel);

        makeImage(g2d, fmFont, chars);

        setPreferredSize(new Dimension(imgW, imgH));
        this.ownerFrame.updateScroller(this.boxHeight);
    }

    /**
     * Creates the offscreen image and build its graphics object.
     *
     * @param   imgW  the image width
     * @param   imgH  the image height
     * @return  the <code>Graphics2D</code> for the image
     */
    private Graphics2D createOffscreen(final int imgW, final int imgH) {

        Graphics2D g2d;

        this.offscreen = new BufferedImage(imgW, imgH, BufferedImage.TYPE_INT_RGB);

        g2d = (Graphics2D) (this.offscreen.getGraphics());
        g2d.setBackground(Color.white);
        g2d.clearRect(0, 0, imgW, imgH);

        return g2d;
    }

    /**
     * Draws the grid on the image.
     *
     * @param  g2d      the <code>Graphics2D</code> to which to draw
     * @param  fmFont   the metrics of the font being rendered
     */
    private void drawGrid(final Graphics2D g2d, final FontMetrics fmFont) {

        int inx;
        int yPos;
        int rows;
        int columns;

        rows = this.offscreen.getHeight() / this.boxHeight;
        columns = this.offscreen.getWidth() / this.boxWidth;

        yPos = fmFont.getHeight();

        // Draw grid
        for (inx = 0; inx <= rows; inx++) {
            g2d.setColor(Color.lightGray);
            g2d.fillRect(0, yPos + (inx * this.boxHeight) + fmFont.getHeight(),
                this.offscreen.getWidth(), this.boxHeight - fmFont.getHeight());
            g2d.setColor(Color.black);
            g2d.drawLine(0, yPos + (inx * this.boxHeight), this.offscreen.getWidth(),
```

```
                yPos + (inx * this.boxHeight));
            g2d.setColor(Color.gray);
            g2d.drawLine(0, yPos + (inx * this.boxHeight) + fmFont.getHeight(),
                this.offscreen.getWidth(), yPos + (inx * this.boxHeight) + fmFont.getHeight());
        }

        g2d.setColor(Color.BLACK);

        for (inx = 0; inx <= columns; inx++) {
            g2d.drawLine(inx * this.boxWidth, yPos, inx * this.boxWidth,
                yPos + this.offscreen.getHeight() - fmFont.getHeight() - fmFont.getLeading() - 1);
        }
    }

    /**
     * Draws the labels on the image.
     *
     * @param  g2d       the <code>Graphics2D</code> to which to draw
     * @param  chars     the characters supported by the font
     * @param  fmFont    the metrics of the font being rendered
     * @param  fmLabel   the metrics of the label font
     */
    private void drawLabels(final Graphics2D g2d, final char[] chars, final FontMetrics fmFont,
        final FontMetrics fmLabel) {

        int xPos;
        int yPos;
        int inx;
        String str;

        yPos = fmFont.getHeight();
        xPos = 0;

        for (inx = 0; inx < chars.length; inx++) {
            str = Integer.toHexString(chars[inx]);
            g2d.drawString(str, xPos + 1 + ((this.boxWidth - fmLabel.stringWidth(str)) / 2),
                yPos + fmFont.getHeight() + fmLabel.getAscent());

            xPos += this.boxWidth;

            if (xPos >= (this.offscreen.getWidth() - 1)) {
                xPos = 0;
                yPos += this.boxHeight;
            }
        }
    }

    /**
     * Draws the image with a grid with labels and all the glyphs.
     *
     * @param  g2d      the <code>Graphics</code> to which to draw
     * @param  fmFont   the metrics for the font being displayed
     * @param  chars    the characters supported by the font
     */
    private void makeImage(final Graphics2D g2d, final FontMetrics fmFont, final char[] chars) {

        FontRenderContext frc;
        int inx;
        int xPos;
        int yPos;
        String str;
        int pixX;
        int pixY;
        char[] chr;

        try {
            g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_LCD_HRGB);
        } catch (NoSuchFieldError e) {
            g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        }

        // Draw the line of example text
        g2d.setFont(this.font);
        g2d.setColor(Color.black);
        str = this.font.getName() + ", " + this.font.getSize() + " point ";
        g2d.drawString(str, 5, fmFont.getAscent() + (fmFont.getLeading() / 2));
        xPos = 5 + fmFont.stringWidth(str);
        g2d.setFont(this.font.deriveFont(Font.BOLD));
        str = " Bold, ";
        g2d.drawString(str, xPos, fmFont.getAscent() + (fmFont.getLeading() / 2));
        xPos += g2d.getFontMetrics().stringWidth(str);
        g2d.setFont(this.font.deriveFont(Font.ITALIC));
        str = " Italic";
        g2d.drawString(str, xPos, fmFont.getAscent() + (fmFont.getLeading() / 2));
        xPos += g2d.getFontMetrics().stringWidth(str);
```

```
        g2d.setFont(this.font);

        // Draw glyphs
        xPos = 0;
        yPos = fmFont.getHeight();

        g2d.setFont(this.font);
        chr = new char[1];

        for (inx = 0; inx < chars.length; inx++) {
            chr[0] = chars[inx];

            pixX = xPos + 1 + ((this.boxWidth - fmFont.charWidth(chr[0])) / 2);
            pixY = yPos + fmFont.getAscent();

            // Draw a glyph bounds box around the character, if configured
            if (this.drawBoxes) {
                g2d.setColor(Color.LIGHT_GRAY);
                frc = g2d.getFontRenderContext();
                g2d.draw(this.font.createGlyphVector(frc, chr).getPixelBounds(frc, pixX, pixY));
                g2d.setColor(Color.BLACK);
            }

            // Draw the character
            g2d.drawChars(chr, 0, 1, pixX, pixY);

            xPos += this.boxWidth;

            if (xPos >= (this.offscreen.getWidth() - 1)) {
                xPos = 0;
                yPos += this.boxHeight;
            }
        }
    }

    /**
     * Gets the list of characters that a font supports.
     *
     * @param   fnt  the font
     * @return  the list of supported characters
     */
    private char[] getFontSupportedChars(final Font fnt) {

        StringBuilder builder;
        char[] chars;

        builder = new StringBuilder(200);

        for (char c = 0; c < 0xFFFF; c++) {

            if (fnt.canDisplay(c)) {
                builder.append(c);
            }
        }

        chars = builder.toString().toCharArray();

        return chars;
    }

    /**
     * Exports the image as a JPEG file.
     *
     * @param  target  the file to write to
     */
    public void export(final File target) {

        Iterator<ImageWriter> iter;
        ImageWriter writer;
        FileImageOutputStream fios;

        iter = ImageIO.getImageWritersByFormatName("png");

        if (iter.hasNext()) {
            writer = iter.next();

            try {
                fios = new FileImageOutputStream(target);
                writer.setOutput(fios);
                writer.write(this.offscreen);
                fios.close();
            } catch (Exception exc) {
                LOG.log(Level.WARNING, "Failed to write {0}: {1}",
                    new Object[] { target.getAbsolutePath(), exc.getLocalizedMessage() });
            }
        }
```

```
    }
}

package com.srbenoit.font;

import java.awt.Dimension;
import java.awt.Font;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.InputStream;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import javax.swing.ButtonGroup;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JScrollPane;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.util.ResourceLoader;

/**
 * An application to render all glyphs for a font, and a specified point size. The glyph's numeric
 * value is shown below each glyph.
 */
public class GlyphViewer extends LoggedObject implements ActionListener, ViewerInt {

    /** the menu items for the font sizes */
    private static final String[] SIZES = {
            "10 point", "12 point", "14 point", "16 point", "18 point", "20 point", "24 point",
            "30 point", "48 point"
        };

    /** the viewer frame */
    private final transient JFrame frame;

    /** the map of font names to menu names */
    private final transient Properties nameMap;

    /** the font manager from which to gather fonts */
    private final transient BundledFontManager mgr;

    /** the font size menu items */
    private transient JMenuItem[] actions = null;

    /** a scroll pane in which the grid of glyphs is rendered */
    private transient JScrollPane scroll = null;

    /** the point size to use when rendering fonts */
    private transient int size = 0;

    /** the name of the font whose glyphs are being viewed */
    private transient String name = null;

    /** the glyph panel used to render the fonts */
    private transient GlyphPanel panel = null;

    /** the named submenus created so far */
    private final transient Map<String, JMenu> submenus;

    /**
     * Constructs a new <code>GlyphViewer</code>.
     *
     * @param  manager  the font manager from which to retrieve fonts
     */
    public GlyphViewer(final BundledFontManager manager) {

        this.frame = new JFrame("Glyph Viewer");
        this.mgr = manager;
        this.submenus = new HashMap<String, JMenu>(20);
        this.nameMap = new Properties();

        loadNameMap();
        buildUI();
    }

    /**
     * Gets the frame.
     *
     * @return  the frame
     */
```

```
public JFrame getFrame() {

    return this.frame;
}

/**
 * Loads the map from font names to menu names.
 */
private void loadNameMap() {

    InputStream input;

    input = ResourceLoader.getInputStream(GlyphViewer.class, "beken/font/fontnames.properties");

    if (input != null) {

        try {
            this.nameMap.load(input);
            input.close();
        } catch (Exception e) {
            LOG.warning("Failed to load beken/font/fontnames.properties");
            this.nameMap.clear();
        }
    }
}

/**
 * Constructs the user interface, which consists of a menu dropdown of the available fonts, a
 * menu dropdown of point sizes, and a panel that shows the selected font's glyphs in the
 * selected size, with the character codes below each glyph.
 */
private void buildUI() {

    JMenuBar bar;
    JMenu menu;
    String[] names;
    int inx;
    ButtonGroup grp;
    JMenuItem[] fontSizes;
    Dimension screen;

    bar = new JMenuBar();

    menu = new JMenu("Fonts");
    names = this.mgr.fontNames();

    for (inx = 0; inx < names.length; inx++) {
        addFontToMenu(menu, names[inx]);
    }

    bar.add(menu);

    fontSizes = new JMenuItem[SIZES.length];
    menu = new JMenu("Sizes");
    grp = new ButtonGroup();

    for (inx = 0; inx < SIZES.length; inx++) {
        fontSizes[inx] = new JRadioButtonMenuItem(SIZES[inx], false); // NOPMD SRB
        fontSizes[inx].addActionListener(this);
        menu.add(fontSizes[inx]);
        grp.add(fontSizes[inx]);
    }

    fontSizes[3].setSelected(true);
    this.size = 16;
    bar.add(menu);

    this.actions = new JMenuItem[2];
    menu = new JMenu("Actions");
    this.actions[0] = new JMenuItem("Export As Image...");
    this.actions[0].setActionCommand("Export");
    this.actions[0].addActionListener(this);
    this.actions[0].setEnabled(false);
    menu.add(this.actions[0]);
    this.actions[1] = new JMenuItem("Toggle bounds boxes");
    this.actions[1].setActionCommand("ToggleBounds");
    this.actions[1].addActionListener(this);
    menu.add(this.actions[1]);
    bar.add(menu);

    this.frame.setJMenuBar(bar);

    this.panel = new GlyphPanel(this);
    this.scroll = new JScrollPane(this.panel);
    this.scroll.getVerticalScrollBar().setUnitIncrement(36);
    this.scroll.setWheelScrollingEnabled(true);
    this.frame.setContentPane(this.scroll);
```

```
        // Center on screen.
        this.frame.pack();
        screen = Toolkit.getDefaultToolkit().getScreenSize();
        this.frame.setLocation((screen.width - this.frame.getWidth()) / 2,
            (screen.height - this.frame.getHeight()) / 2);
        this.frame.setVisible(true);
    }

    /**
     * Create the menu item for a single font.
     *
     * @param  menu       the menu to which to add the font item
     * @param  fontName   the name of the font to add
     */
    private void addFontToMenu(final JMenu menu, final String fontName) {

        JMenuItem menuItem;
        String menuName;
        JMenu submenu;
        Enumeration<?> names;
        String key;

        menuItem = new JRadioButtonMenuItem(fontName, false);
        menuItem.addActionListener(this);

        submenu = menu;
        menuName = fontName;

        names = this.nameMap.propertyNames();

        while (names.hasMoreElements()) {
            key = (String) names.nextElement();

            if (fontName.startsWith(key)) {
                menuName = this.nameMap.getProperty(key);

                // See if we already have this menu
                submenu = this.submenus.get(menuName);

                break;
            }
        }

        if (submenu == null) {
            submenu = new JMenu(menuName);
            this.submenus.put(menuName, submenu);
            menu.add(submenu);
        }

        submenu.add(menuItem);
    }

    /**
     * Handler for menu item selections.
     *
     * @param  evt  the action event
     */
    public void actionPerformed(final ActionEvent evt) {

        String cmd;
        String num;
        int pickedSize;
        JFileChooser chooser;
        File file;

        cmd = evt.getActionCommand();

        if (cmd.endsWith("␣point")) {
            num = cmd.substring(0, cmd.length() - 6);
            pickedSize = Integer.parseInt(num);

            if (this.size != pickedSize) {
                this.size = pickedSize;

                if (this.name != null) {
                    rebuildFont();
                }
            }
        } else if ("Export".equals(cmd)) {
            chooser = new JFileChooser();
            chooser.setMultiSelectionEnabled(false);
            chooser.setFileSelectionMode(JFileChooser.FILES_ONLY);

            file = new File(chooser.getCurrentDirectory(), this.name + ".png");
            chooser.setSelectedFile(file);
```

```
            if (chooser.showSaveDialog(this.frame) == JFileChooser.APPROVE_OPTION) {
                this.panel.export(chooser.getSelectedFile());
            }
        } else if ("ToggleBounds".equals(cmd)) {
            this.panel.setDrawBoundsBoxes(!this.panel.isDrawingBoundsBoxes());
        } else {
            this.name = cmd;

            if (this.size != 0) {
                rebuildFont();
            }
        }
    }

    /**
     * Regenerates the font and glyphs display.
     */
    private void rebuildFont() {

        Font font;

        font = this.mgr.getFont(this.name, this.size, Font.PLAIN);
        this.panel.setTheFont(font);

        // Enable export as image
        this.actions[0].setEnabled(true);
    }

    /**
     * Tells the scroll pane that something inside it has changed.
     *
     * @param  jump  the vertical size of boxes
     */
    public void updateScroller(final int jump) {

        this.scroll.getVerticalScrollBar().setUnitIncrement(jump);
        this.scroll.revalidate();

        this.frame.repaint();
    }

    /**
     * Main method to launch the application.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        GlyphViewer viewer;

        viewer = new GlyphViewer(BundledFontManager.getInstance());
        viewer.getFrame().setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```
package com.srbenoit.font;

/**
 * An interface for viewers that have a scroll pane.
 */
public interface ViewerInt {

    /**
     * Tells the scroll pane that something inside it has changed.
     *
     * @param  jump  the vertical size of boxes
     */
    void updateScroller(int jump);
}
```

E.1.8 XML File Management (com.srbenoit.xml)

This package provides lightweight classes to read and parse a simple subset of the XML format, useful for many situations where XML is used but the full aparatus of the generalized protocol is overkill.

```
package com.srbenoit.xml;

/**
 * A CDATA node, used to represent any characters that are not part of tags. Whitespace is retained
 * in these nodes exactly as supplied in the source XML.
 */
public class CData implements Node {

    /** the start position of the block */
    public final transient int start;

    /** the end position of the block */
    public final transient int end;

    /**
     * Constructs a new <code>CData</code>.
     *
     * @param  startPos  the start position of the block
     * @param  endPos    the end position of the block
     */
    public CData(final int startPos, final int endPos) {

        this.start = startPos;
        this.end = endPos;
    }

    /**
     * Generates the print representation of the node (may recursively call this method on child
     * nodes).
     *
     * @param   xml  the source XML
     * @return  the print representation
     */
    public String print(final String xml) {

        return xml.substring(this.start, this.end);
    }
}

package com.srbenoit.xml;

import java.io.UnsupportedEncodingException;
import java.net.URLDecoder;
import java.net.URLEncoder;
import java.util.TreeMap;

/**
 * The base class for XML elements.
 */
public class ElementBase extends TreeMap<String, String> {

    /** version number for serialization */
    private static final long serialVersionUID = -5830831427588246195L;

    /** the line end character */
    protected static final String CRLF;

    /** the character encoding to use for URL encode/decode */
    protected static final String UTF8 = "UTF-8";

    /** the parsed tag name */
    public final transient String tagName;

    static {
        String crlf;

        crlf = System.getProperty("line.separator");
        CRLF = (crlf == null) ? "\r\n" : crlf;
    }

    /**
     * Constructs a new <code>AbstractElement</code>.
     *
     * @param  tag  the parsed tag name
     */
    public ElementBase(final String tag) {

        super();

        this.tagName = tag;
    }

    /**
     * Encode a string for inclusion in XML.
     *
     * @param   str  the string to encode
     * @return  the encoded string
```

```
     */
    public static String encode(final String str) {

        String result;

        try {
            result = URLEncoder.encode(str, UTF8);
        } catch (UnsupportedEncodingException e) {
            result = str;
        }

        return result;
    }

    /**
     * Decode a string from its XML escaped representation.
     *
     * @param   str  the string to decode
     * @return  the decoded string
     */
    public static String decode(final String str) {

        String result;

        try {
            result = URLDecoder.decode(str, UTF8);
        } catch (UnsupportedEncodingException e) {
            result = str;
        }

        return result;
    }
}

package com.srbenoit.xml;

/**
 * An empty element, characterized by a tag of the form &lt; ... /&gt;.
 */
public class EmptyElement extends ElementBase implements Node {

    /** version number for serialization */
    private static final long serialVersionUID = 2526807668030298751L;

    /** the tag span of the element */
    public final transient TagSpan tagSpan;

    /**
     * Constructs a new <code>EmptyElement</code>. /** Constructs a new <code>Element</code>.
     *
     * @param  tag   the parsed tag name
     * @param  span  the tag span for the element's tag
     */
    public EmptyElement(final String tag, final TagSpan span) {

        super(tag);

        this.tagSpan = span;
    }

    /**
     * Generates the print representation of the node (may recursively call this method on child
     * nodes).
     *
     * @param   xml  the source XML
     * @return  the print representation
     */
    public String print(final String xml) {

        StringBuilder str;
        String key;
        String value;

        str = new StringBuilder(200);

        str.append('<');
        str.append(this.tagName);

        for (String attr : keySet()) {

            key = encode(attr);
            value = encode(get(attr));

            str.append(' ');
            str.append(key);
            str.append('=');
            str.append(value.contains("'") ? "\"" : "'");
```

```
            str.append(value);
            str.append(value.contains("'") ? "\"" : "'");
        }

        str.append("/>");

        return str.toString();
    }
}

package com.srbenoit.xml;

/**
 * The interface implemented by elements, empty elements, and CData spans.
 */
public interface Node {

    /**
     * Generates the print representation of the node.
     *
     * @param   xml  the source XML
     * @return  the print representation
     */
    String print(String xml);
}

package com.srbenoit.xml;

import java.util.ArrayList;
import java.util.List;

/**
 * An element, characterized by a pair of tags of the form &lt; ... &gt; and &lt;/ ... &gt;.
 */
public class NonemptyElement extends ElementBase implements Node {

    /** version number for serialization */
    private static final long serialVersionUID = -7401017410745873815L;

    /** the opening tag span of the element */
    public final transient TagSpan openTagSpan;

    /** the list of children of this element */
    public final transient List<Node> children;

    /**
     * Constructs a new <code>Element</code>.
     *
     * @param  tag   the parsed tag name
     * @param  span  the tag span for the open tag
     */
    public NonemptyElement(final String tag, final TagSpan span) {

        super(tag);

        this.openTagSpan = span;
        this.children = new ArrayList<Node>(4);
    }

    /**
     * Generates the print representation of the node (may recursively call this method on child
     * nodes).
     *
     * @param   xml  the source XML
     * @return  the print representation
     */
    public String print(final String xml) {

        StringBuilder str;
        String key;
        String value;

        str = new StringBuilder(200);

        str.append('<');
        str.append(this.tagName);

        for (String attr : keySet()) {
            key = encode(attr);
            value = encode(get(attr));

            str.append(' ');
            str.append(key);
            str.append('=');
            str.append(value.contains("'") ? "\"" : "'");
            str.append(value);
            str.append(value.contains("'") ? "\"" : "'");
```

```
        }

        str.append(">");

        // Append children
        for (Node child : this.children) {
            str.append(child.print(xml));
        }

        str.append("</");
        str.append(this.tagName);
        str.append(">");

        return str.toString();
    }
}

package com.srbenoit.xml;

/**
 * A generic span that covers a tag.
 */
public class TagSpan {

    /** the index of the start of the tag (the &lt; character) */
    public final transient int start;

    /** the index of the end of the tag (the &gt; character) */
    public final transient int end;

    /** <code>true</code> if this is an open (or empty) tag */
    public final transient boolean isOpen;

    /** <code>true</code> if this is a close (or empty) tag */
    public final transient boolean isClose;

    /** the index of the start of the tag name */
    public final transient int nameStart;

    /** the index of the end of the tag name */
    public final transient int nameEnd;

    /**
     * Constructs a new <code>TagSpan</code>.
     *
     * @param  startPos       the start position of the tag
     * @param  endPos         the end position of the tag
     * @param  open           <code>true</code> if this is an open (or empty) tag
     * @param  close          <code>true</code> if this is a close (or empty) tag
     * @param  nameStartPos   the start position of the tag name
     * @param  nameEndPos     the end position of the tag name
     */
    public TagSpan(final int startPos, final int endPos, final boolean open, final boolean close,
        final int nameStartPos, final int nameEndPos) {

        this.start = startPos;
        this.end = endPos;
        this.isOpen = open;
        this.isClose = close;
        this.nameStart = nameStartPos;
        this.nameEnd = nameEndPos;
    }
}

package com.srbenoit.xml;

import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * A utility class to escape strings for use in XML (as an attribute value, for example) and
 * recover the original strings from such escaped string values.
 */
public final class XmlEscaper extends LoggedObject {

    /** a string builder used to construct output strings */
    private static final StringBuilder STR;

    static {
        STR = new StringBuilder(100);
    }

    /**
     * Private constructor to prevent instantiation.
     */
    private XmlEscaper() { }
```

```
/**
 * Escapes a string for use in an XML attribute.
 *
 * @param   orig  the original string to escape
 * @return  the escaped string
 */
public static String escape(final String orig) {

    char[] chars;

    chars = orig.toCharArray();

    synchronized (STR) {
        STR.setLength(0);

        for (char chr : chars) {

            switch (chr) {

            case '\"':
                STR.append("&quot;");
                break;

            case '\'':
                STR.append("&apos;");
                break;

            case '<':
                STR.append("&lt;");
                break;

            case '>':
                STR.append("&gt;");
                break;

            case '&':
                STR.append("&amp;");
                break;

            default:
                STR.append(chr);
                break;
            }
        }

        return STR.toString();
    }
}

/**
 * Unescapes a string that was processed by <code>escape</code>.
 *
 * @param   escaped  the string to unescape
 * @return  the original (unescaped) string
 */
public static String unescape(final String escaped) {

    char[] chars;
    int len;
    int pos;

    chars = escaped.toCharArray();
    len = chars.length;

    synchronized (STR) {
        STR.setLength(0);

        pos = 0;

        while (pos < len) {

            if (chars[pos] == '&') {

                if ((pos < (len - 5)) && (chars[pos + 1] == 'q') && (chars[pos + 2] == 'u')
                        && (chars[pos + 3] == 'o') && (chars[pos + 4] == 't')
                        && (chars[pos + 5] == ';')) {
                    STR.append('\"');
                    pos += 6;
                } else if ((pos < (len - 5)) && (chars[pos + 1] == 'a')
                        && (chars[pos + 2] == 'p') && (chars[pos + 3] == 'o')
                        && (chars[pos + 4] == 's') && (chars[pos + 5] == ';')) {
                    STR.append('\'');
                    pos += 6;
                } else if ((pos < (len - 3)) && (chars[pos + 1] == 'l')
                        && (chars[pos + 2] == 't') && (chars[pos + 3] == ';')) {
                    STR.append('<');
                    pos += 4;
```

```
                    } else if ((pos < (len - 3)) && (chars[pos + 1] == 'g')
                            && (chars[pos + 2] == 't') && (chars[pos + 3] == ';')) {
                        STR.append('>');
                        pos += 4;
                    } else if ((pos < (len - 4)) && (chars[pos + 1] == 'a')
                            && (chars[pos + 2] == 'm') && (chars[pos + 3] == 'p')
                            && (chars[pos + 4] == ';')) {
                        STR.append('&');
                        pos += 5;
                    } else {
                        LOG.warning("Invalid escape sequence detected");
                        STR.append(chars[pos]);
                        pos++;
                    }
                } else {
                    STR.append(chars[pos]);
                    pos++;
                }
            }

            return STR.toString();
        }
    }

    /**
     * Main method to exercise the methods in this class.
     *
     * @param  args  command-line arguments (ignored)
     */
    public static void main(final String... args) {

        String orig;
        String escaped;
        String test;

        orig = "\"'&<>&'><&>\"\"\"'''\"";
        escaped = XmlEscaper.escape(orig);
        test = XmlEscaper.unescape(escaped);

        LOG.log(Level.FINE, "Original:   [{0}]", orig);
        LOG.log(Level.FINE, "Escaped:    [{0}]", escaped);
        LOG.log(Level.FINE, "Recovered:  [{0}]", test);
    }
}
```

```
package com.srbenoit.xml;

import java.io.File;
import javax.swing.filechooser.FileFilter;

/**
 * A filter to limit file views to only files with ".xml" extensions (not case sensitive).
 */
public class XmlFileFilter extends FileFilter implements java.io.FileFilter {

    /**
     * Tests whether or not the specified abstract pathname should be included in a pathname list.
     *
     * @param    pathname  The abstract pathname to be tested
     * @return  <code>true</code> if and only if <code>pathname</code> should be included
     */
    @Override public boolean accept(final File pathname) {

        return pathname.getName().toLowerCase().endsWith(".xml");
    }

    /**
     * Gets the description of this filter.
     *
     * @return  the description
     */
    @Override public String getDescription() {

        return "XML files (.xml)";
    }
}
```

```
package com.srbenoit.xml;

import java.text.ParseException;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.List;
import com.srbenoit.log.LoggedObject;

/**
```

```
 * A very basic parser that can extract structural information from XML and can detect some XML
 * syntax problems.
 */
public class XmlParser extends LoggedObject {

    /** the list of tag spans */
    private final transient List<TagSpan> tags;

    /** a stack of open tags (must be matched by close tags in proper order) */
    private final transient Deque<NonemptyElement> openTags;

    /**
     * Constructs a new <code>XmlParser</code>.
     */
    public XmlParser() {

        super();

        this.tags = new ArrayList<TagSpan>(20);
        this.openTags = new ArrayDeque<NonemptyElement>(20);
    }

    /**
     * Parses some XML into an object tree.
     *
     * @param   xml           the XML to parse
     * @param   ignoreCData   <code>true</code> to ignore any CData in the XML, resulting in a node
     *                        tree containing only <code>EmptyElement</code> and <code>
     *                        NonemptyElement</code> nodes.
     * @return  the list of top-level nodes, of which any <code>Element</code> nodes can contain
     *          child nodes
     * @throws  ParseException  if an error is detected in the XML
     */
    public List<Node> parse(final String xml, final boolean ignoreCData) throws ParseException {

        List<Node> nodes;
        int len;
        int pos;
        int index;
        TagSpan span;
        String tag;

        // Construct the list of tags (open, close, and empty) in the XML
        this.tags.clear();
        buildSpansList(xml);

        nodes = new ArrayList<Node>(20);

        // Build the set of nodes from the list of spans
        len = xml.length();
        pos = 0;
        index = 0;

        while ((pos < len) && (index < this.tags.size())) {

            // Get the next tag
            span = this.tags.get(index);
            index++;

            // If there are characters before the tag, add a CData node
            if (pos < span.start) {

                if (!ignoreCData) {
                    makeCdata(pos, span.start, nodes);
                }

                pos = span.start;
            }

            if (span.isOpen) {

                if (span.isClose) {
                    makeEmpty(xml, span, nodes);
                } else {
                    makeNonempty(xml, span, nodes);
                }
            } else {

                if (this.openTags.isEmpty()) {
                    throw new ParseException("Unmatched close tag: '" + getTag(xml, span) + "'",
                        span.start);
                }

                // This is a non-empty element's close tag
                tag = getTag(xml, span);

                if (!this.openTags.pop().tagName.equals(tag)) {
```

```
                throw new ParseException("Mismatched close tag: '" + tag + "'", span.start);
            }
        }

        pos = span.end + 1;
    }

    if ((pos < len) && (!ignoreCData)) {
        makeCdata(pos, len, nodes);
    }

    return nodes;
}

/**
 * Builds a CDATA element and adds it to the node list.
 *
 * @param  start  the offset of the start of the data
 * @param  end    the offset of the end of the data
 * @param  nodes  the list of nodes to which to add the element
 */
private void makeCdata(final int start, final int end, final List<Node> nodes) {

    CData cdata;

    cdata = new CData(start, end);

    if (this.openTags.isEmpty()) {
        nodes.add(cdata);
    } else {
        this.openTags.getFirst().children.add(cdata);
    }
}

/**
 * Builds an empty element and adds it to the node list.
 *
 * @param   xml     the source XML
 * @param   span    the tag span describing the empty element
 * @param   nodes   the list of nodes to which to add the element
 * @throws  ParseException  if an error is detected in the XML
 */
private void makeEmpty(final String xml, final TagSpan span, final List<Node> nodes)
    throws ParseException {

    String tag;
    EmptyElement empty;

    tag = getTag(xml, span);
    empty = new EmptyElement(tag, span);
    extractAttribtues(xml, span, empty);

    if (this.openTags.isEmpty()) {
        nodes.add(empty);
    } else {
        this.openTags.getFirst().children.add(empty);
    }
}

/**
 * Builds a nonempty element and adds it to the node list.
 *
 * @param   xml     the source XML
 * @param   span    the tag span describing the empty element
 * @param   nodes   the list of nodes to which to add the element
 * @throws  ParseException  if an error is detected in the XML
 */
private void makeNonempty(final String xml, final TagSpan span, final List<Node> nodes)
    throws ParseException {

    String tag;
    NonemptyElement nonempty;

    // This is a non-empty element's open tag
    tag = getTag(xml, span);
    nonempty = new NonemptyElement(tag, span);
    extractAttribtues(xml, span, nonempty);

    if (this.openTags.isEmpty()) {
        nodes.add(nonempty);
    } else {
        this.openTags.getFirst().children.add(nonempty);
    }

    this.openTags.push(nonempty);
}
```

```
/**
 * Scans through the XML looking for '&lt;' and '&gt;' characters and assembles them into an
 * ordered list of spans.
 *
 * @param   xml  the XML to parse
 * @throws  ParseException  if there is an invalid or unterminated tag
 */
private void buildSpansList(final String xml) throws ParseException {

    int start;
    int end;

    // Scan through the XML accumulating a list of '<', '>', '</', '/>'
    // and assembling them into tag spans
    start = xml.indexOf('<');

    while (start >= 0) {

        end = xml.indexOf('>', start + 1);

        if (end == -1) {
            throw new ParseException("Unterminated tag (missing '>')", start);
        }

        // Detect pathological "<>" and "</>" cases.
        if (end == (start + 1)) {
            throw new ParseException("Invalid tag (<>)", start);
        }

        if ((end == (start + 2)) && (xml.charAt(start + 1) == '/')) {
            throw new ParseException("Invalid tag (</>)", start);
        }

        validateTag(xml, start, end);

        start = xml.indexOf('<', end + 1);
    }
}

/**
 * Examine a tag delimited by a start and end position, and add it to the list of tags.
 *
 * @param   xml     the source XML
 * @param   start   the start position of the tag
 * @param   end     the end position of the tag
 * @throws  ParseException  if there is an invalid or unterminated tag
 */
private void validateTag(final String xml, final int start, final int end)
    throws ParseException {

    TagSpan span;
    char ch1;
    char ch2;
    int nameStart;
    int nameEnd;

    ch1 = xml.charAt(start + 1);
    ch2 = xml.charAt(end - 1);

    // Start of tag name is first non-whitespace after tag start
    nameStart = scanWhitespace(xml, (ch1 == '/') ? (start + 2) : (start + 1), end, false);

    // Test for no tag name, as in "< >" or "< />"
    if ((nameStart == end) || ((nameStart == (end - 1)) && (ch2 == '/'))) {
        throw new ParseException("Missing tag name", start);
    }

    // End of tag name is first whitespace after tag name start,
    // excepting the possible '/' at the end of an empty tag
    nameEnd = scanWhitespace(xml, nameStart + 1, end, true);

    if ((nameEnd == end) && (ch2 == '/')) {
        nameEnd--;
    }

    span = new TagSpan(start, end, (ch1 != '/'), ((ch1 == '/') || (ch2 == '/')), nameStart,
            nameEnd);
    this.tags.add(span);
}

/**
 * Scans the source XML for the next character that is either whitespace or not whitespace,
 * starting at a specified starting index and limiting the search to a specified ending index.
 *
 * @param   xml             the source XML
 * @param   start           the index at which to begin searching
 * @param   end             the index at which to end searching
```

```
     * @param   isWhitespace  <code>true</code> to search for the next whitespace, <code>
     *                        false</code> to search for the next non-whitespace
     * @return  the index of the first matching character in the designated range, or the end index
     *          if no non-whitespace was found
     */
    private int scanWhitespace(final String xml, final int start, final int end,
        final boolean isWhitespace) {

        int pos;

        for (pos = start; pos < end; pos++) {

            if (Character.isWhitespace(xml.charAt(pos)) == isWhitespace) {
                break;
            }
        }

        return pos;
    }

    /**
     * Returns the tag name from a tag span.
     *
     * @param   xml   the source XML
     * @param   span  the tag span
     * @return  the extracted tag name
     */
    private String getTag(final String xml, final TagSpan span) {

        String raw;
        String name;

        raw = xml.substring(span.nameStart, span.nameEnd);
        name = XmlEscaper.unescape(raw);

        return name;
    }

    /**
     * Extracts a list of attributes from a tag.
     *
     * @param   xml       the source XML
     * @param   span      the tag span defining the tag from which to extract attributes
     * @param   element   the element to which to add extracted attributes
     * @throws  ParseException  if there is an error parsing attributes
     */
    private void extractAttribtues(final String xml, final TagSpan span, final ElementBase element)
        throws ParseException {

        char chr;
        int end;
        int pos;

        // Figure out where tag content ends
        end = (xml.charAt(span.end - 1) == '/') ? (span.end - 1) : span.end;

        // Scan for attribute name='value' pairs
        pos = span.nameEnd;

        while (pos < end) {
            chr = xml.charAt(pos);

            // Skip whitespace before attribute name
            if (Character.isWhitespace(chr)) {
                pos++;
            } else if (chr == '=') {

                // Leading '=' not valid as attribute name
                throw new ParseException("Missing attribute name", pos);
            } else {
                pos = processAttribute(xml, pos, end, element);
            }
        }
    }

    /**
     * Extracts a single attribute from the XML stream.
     *
     * @param   xml       the source XMl
     * @param   start     the start position of the attribute
     * @param   end       the end of the current tag
     * @param   element   the element to which to add the attribute
     * @return  the position immediately following the end of the attribute
     * @throws  ParseException  if there is an error parsing the attribute
     */
    private int processAttribute(final String xml, final int start, final int end,
        final ElementBase element) throws ParseException {
```

```
        int pos;
        char chr;
        int attrEnd;
        int equals;
        int valueEnd;
        char match;

        // Find the end of the attribute name and the equals sign
        for (attrEnd = start; attrEnd < end; attrEnd++) {
            chr = xml.charAt(attrEnd);

            if (Character.isWhitespace(chr) || (chr == '=')) {
                break;
            }
        }

        equals = scanWhitespace(xml, attrEnd, end, false);

        if ((equals == end) || (xml.charAt(equals) != '=')) {
            throw new ParseException("Invalid attribute", start);
        }

        pos = scanWhitespace(xml, equals + 1, end, false);

        // Attribute must be wrapped in ' or "
        match = xml.charAt(pos);

        if ("'\"".indexOf(match) == -1) {
            throw new ParseException("Invalid attribute value", pos);
        }

        valueEnd = xml.indexOf(match, pos + 1);

        if ((valueEnd == -1) || (valueEnd >= end)) {
            throw new ParseException("Unterminated attribute value", pos);
        }

        addAttr(xml, start, attrEnd, pos + 1, valueEnd, element);

        return valueEnd + 1;
    }

    /**
     * Decodes an attribute name and value and adds the attribute to an element.
     *
     * @param  xml          the source XML
     * @param  attrStart    the index of the start of the attribute name
     * @param  attrEnd      the index of the end of the attribute name
     * @param  valueStart   the index of the start of the attribute value
     * @param  valueEnd     the index of the end of the attribute value
     * @param  element      the element to which to add the attribute
     */
    private void addAttr(final String xml, final int attrStart, final int attrEnd,
        final int valueStart, final int valueEnd, final ElementBase element) {

        String raw;
        String name;
        String value;

        raw = xml.substring(attrStart, attrEnd).trim();
        name = XmlEscaper.unescape(raw);

        raw = xml.substring(valueStart, valueEnd).trim();
        value = XmlEscaper.unescape(raw);

        element.put(name, value);
    }
}
```

E.1.9 User Interface Utilities (com.srbenoit.ui)

This package provides utilities to build and manage user interfaces. This class relies on the Substance UI library. The location where this library is hosted is very dynamic. If you cannot find "substance.jar" and "trident.jar", just don't request that this class

change to the substance UI.

```
package com.srbenoit.ui;

import java.lang.reflect.InvocationTargetException;
import java.util.logging.Level;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import com.srbenoit.log.LoggedObject;
import org.pushingpixels.substance.api.SubstanceLookAndFeel;
import org.pushingpixels.substance.api.skin.BusinessSkin;

/**
 * A convenience class to house the static code to change to the Nimbus UI.
 */
public final class ChangeUI extends LoggedObject implements Runnable {

    /** the class name of the look and feel to use */
    public static final String NIMBUS = "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel";

    /** the class name of the look and feel to use */
    public static final String SUBSTANCE = "org.pushingpixels.substance.api.SubstanceLookAndFeel";

    /**
     * Private constructor to prevent instantiation.
     */
    private ChangeUI() {

        super();
    }

    /**
     * Sets the UI to the Nimbus UI.
     */
    static public void changeUI() {

        try {
            UIManager.setLookAndFeel(NIMBUS);
        } catch (Exception e) {
            LOG.log(Level.WARNING, "Failed to set look-and-feel", e);
        }
    }

    /**
     * Sets the UI to the Substance UI.
     */
    static public void changeUISubstance() {

        try {
            SwingUtilities.invokeAndWait(new ChangeUI());
        } catch (InterruptedException e) {
            LOG.log(Level.WARNING, "Failed to set look-and-feel", e);
        } catch (InvocationTargetException e) {
            LOG.log(Level.WARNING, "Failed to set look-and-feel", e);
        }
    }

    /**
     */
    public void run() {

        SubstanceLookAndFeel.setSkin(new BusinessSkin());
    }
}

package com.srbenoit.ui;

import javax.swing.JFrame;

/**
 * An interface implemented by classes that need to construct a GUI in the AWT event thread.
 */
public interface GuiBuilder {

    /**
     * Constructs the GUI, to be called in the AWT event thread.
     *
     * @param  frame  the frame to which to add menus if needed
     */
    void buildUI(JFrame frame);
}

package com.srbenoit.ui;

import java.util.logging.Level;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
```

```
import com.srbenoit.log.LoggedObject;

/**
 * A class to execute a <code>GuiBuilder</code> from the AWT event thread.
 */
public final class GuiBuilderRunner extends LoggedObject implements Runnable {

    /** the <code>GuiBuilder</code> whose build method is to be called */
    private final transient GuiBuilder owner;

    /** the frame to which to add menus if needed */
    private transient JFrame frame;

    /**
     * Constructs a new <code>GuiBuilderRunner</code>.
     *
     * @param  theOwner  the <code>GuiBuilder</code> whose build method is to be called
     */
    public GuiBuilderRunner(final GuiBuilder theOwner) {

        super();
        this.owner = theOwner;
    }

    /**
     * Execute the <code>buildUI</code> method on the owner object in the AWT event thread, waiting
     * for that to complete before returning.
     *
     * @param  menuFrame  the frame to which to add menus if needed
     */
    public void buildUI(final JFrame menuFrame) {

        this.frame = menuFrame;

        if (SwingUtilities.isEventDispatchThread()) {
            run();
        } else {

            try {
                SwingUtilities.invokeAndWait(this);
            } catch (Exception ex1) {
                LOG.log(Level.WARNING, "Failed to invoke UI builder: ", ex1);
            }
        }
    }

    /**
     * Runnable method to call the owner's <code>buildUI</code> method from within the AWT event
     * dispatcher thread.
     */
    public void run() {

        this.owner.buildUI(this.frame);
    }
}

package com.srbenoit.ui;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Insets;
import javax.swing.JPanel;
import com.srbenoit.math.Histogram;

/**
 * A panel that displays a histogram of data.
 */
public class HistogramPanel extends JPanel {

    /** version number for serialization */
    private static final long serialVersionUID = -1112693135693520783L;

    /** the histogram data */
    private final transient Histogram data;

    /** the labels for the planes of the histogram */
    private final transient String[] labels;

    /** the line color */
    private final transient Color lines;

    /** the point color */
    private final transient Color points;
```

```
/** the line accent color */
private final transient Color lines2;

/** the point accent color */
private final transient Color points2;

/** the grid color */
private final transient Color grid;

/** the background color */
private final transient Color back;

/** a boldface font */
protected final transient Font font;

/** the phase of the color switch */
private transient boolean phase;

/**
 * Constructs a new <code>HistogramPanel</code>.
 *
 * @param  hist              the histogram to display
 * @param  histogramLabels   the labels for the histogram planes
 * @param  backgroundColor   the background color
 * @param  linesColor        the lines color
 * @param  pointsColor       the points color
 * @param  gridColor         the grid color
 * @param  backColor         the background color
 */
public HistogramPanel(final Histogram hist, final String[] histogramLabels,
    final Color backgroundColor, final Color linesColor, final Color pointsColor,
    final Color gridColor, final Color backColor) {

    super();

    Insets insets;
    int width;

    if (histogramLabels.length != hist.getNumPlanes()) {
        throw new IllegalArgumentException("Label length mismatch");
    }

    this.data = hist;
    this.labels = histogramLabels.clone();
    this.lines = linesColor;
    this.lines2 = new Color((int) (linesColor.getRed() * 0.8),
            (int) (linesColor.getGreen() * 0.8), (int) (linesColor.getBlue() * 0.8));
    this.points = pointsColor;
    this.points2 = new Color((int) (pointsColor.getRed() * 0.8),
            (int) (pointsColor.getGreen() * 0.8), (int) (pointsColor.getBlue() * 0.8));
    this.grid = gridColor;
    this.back = backColor;

    this.font = getFont().deriveFont(Font.BOLD, 11.0f);
    setFont(this.font);

    setBackground(backgroundColor);

    // Compute the preferred width
    insets = getInsets();
    width = insets.left + insets.right + 12 + hist.getNumTimes();
    setPreferredSize(new Dimension(width, 50 * hist.getNumPlanes()));
}

/**
 * Gets the histogram that this panel is displaying.
 *
 * @return  the histogram.
 */
public Histogram getData() {

    return this.data;
}

/**
 * Registers a tick from the owning panel, which shifts the histogram to the left one notch, and
 * initializes the newly opened slot to a particular value.
 */
public void tick() {

    this.data.shiftHigher();
    this.phase ^= true; // NOT operation
    invalidate();
    repaint();
}

/**
```

```
     * Draws an activity histogram.
     *
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    @Override public void paintComponent(final Graphics grx) {

        super.paintComponent(grx);

        if (isEnabled()) {

            synchronized (this.data) {
                paintContents(grx);
            }
        }
    }

    /**
     * Draws an activity histogram.
     *
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    private void paintContents(final Graphics grx) {

        Insets insets;
        int height;
        int heightPerPlane;
        int top;
        FontMetrics met;
        int xPos;
        int max;
        int scale;
        int xPix;
        int yPix;
        int barHeight;
        boolean toggle;

        super.paintComponent(grx);

        insets = getInsets();
        height = getHeight() - insets.top - insets.bottom;
        heightPerPlane = height / this.data.getNumPlanes();

        met = grx.getFontMetrics();
        barHeight = heightPerPlane - met.getHeight();

        // Don't draw lines unless our panel is tall enough
        top = insets.top;
        xPos = insets.left + 3;

        for (int plane = 0; plane < this.data.getNumPlanes(); plane++) {
            scale = 1;
            if (barHeight > 1) {
                max = maxValue(plane);

                // Construct a scale to use when drawing data
                while (max > barHeight) {
                    scale++;
                    max = max * (scale - 1) / scale;
                }

                // Draw the label with optional scale indicator.
                drawLabel(grx, scale, top + met.getAscent(), plane);

                // Draw a background
                grx.setColor(this.back);
                grx.fillRect(xPos, top + heightPerPlane - barHeight, this.data.getNumTimes() + 1,
                    barHeight + 1);

                // Draw some grid lines, and labels
                grx.setColor(this.grid);

                for (int i = 0; i < barHeight; i += 10) {
                    yPix = top + heightPerPlane - i;
                    grx.drawLine(xPos, yPix, xPos + this.data.getNumTimes(), yPix);
                }

                // Draw the lines
                xPix = xPos + this.data.getNumTimes();

                for (int i = 0; i < this.data.getNumTimes(); i++) {
                    toggle = ((i & 0x01) == 0x01) ^ this.phase;
                    yPix = top + heightPerPlane - (this.data.getValue(plane, i) / scale);
                    grx.setColor(toggle ? this.lines : this.lines2);
                    grx.drawLine(xPix, top + heightPerPlane, xPix, yPix);
                    grx.setColor(toggle ? this.points : this.points2);
                    grx.drawLine(xPix, yPix, xPix, yPix);
                    xPix--;
```

```
                }
            }

            top += heightPerPlane;
        }
    }

    /**
     * Finds the maximum data value for a plane.
     *
     * @param    plane
     * @return   the maximum value
     */
    private int maxValue(final int plane) {

        int max;
        int value;

        max = this.data.getValue(plane, 0);

        for (int i = 1; i < this.data.getNumTimes(); i++) {
            value = this.data.getValue(plane, i);

            if (value > max) {
                max = value;
            }
        }

        return max;
    }

    /**
     * Draw a label at the top of the histogram.
     *
     * @param  grx     the <code>Graphics</code> to which to draw
     * @param  scale   the scale
     * @param  yPos    the Y position at which to draw
     * @param  plane   the plane whose label to draw
     */
    private void drawLabel(final Graphics grx, final int scale, final int yPos, final int plane) {

        StringBuilder str;

        str = new StringBuilder(100);

        str.append(this.labels[plane]);
        str.append(" (");
        str.append(10 * scale);
        str.append(" per gridline)");
        str.append(":");

        grx.setColor(this.points2);
        grx.drawString(str.toString(), 4, yPos);
    }
}

package com.srbenoit.ui;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import javax.swing.JPanel;

/**
 * A panel that simply presents an offscreen image. It is intended to be placed in a scroll pane.
 */
public class OffscreenImagePanel extends JPanel {

    /** version number for serialization */
    private static final long serialVersionUID = -8490317779957133123L;

    /** the offscreen image to present */
    private final transient BufferedImage offscreen;

    /**
     * Constructs a new <code>OffscreenImagePanel</code>.
     *
     * @param  image  the buffered image to present.
     */
    public OffscreenImagePanel(final BufferedImage image) {

        super();

        this.offscreen = image;
        setPreferredSize(new Dimension(image.getWidth(), image.getHeight()));
    }
```

```
    /**
     * Draws the image to the screen.
     *
     * @param  grx  the <code>Graphics</code> object to draw to
     */
    @Override public void paintComponent(final Graphics grx) {

        grx.drawImage(this.offscreen, 0, 0, null);
    }
}

package com.srbenoit.ui;

import java.awt.Component;
import java.awt.Dimension;
import java.awt.Graphics2D;
import java.awt.GraphicsEnvironment;
import java.awt.Toolkit;
import java.awt.image.BufferedImage;

/**
 * A set of utilities for user interface construction.
 */
public final class UIUtilities {

    /**
     * Private constructor to prevent instantiation.
     */
    private UIUtilities() {

        // No action
    }

    /**
     * Gets a <code>Graphics2D</code> object that is compatible with the current desktop.
     *
     * @return  the <code>Graphics2D</code>
     */
    public static Graphics2D getGraphics() {

        GraphicsEnvironment env;
        BufferedImage img;

        env = GraphicsEnvironment.getLocalGraphicsEnvironment();

        img = env.getDefaultScreenDevice().getDefaultConfiguration().createCompatibleImage(1, 1);

        return env.createGraphics(img);
    }

    /**
     * Places a component (typically a frame or dialog) at a particular position within the desktop.
     *
     * @param  frame  the frame to position
     * @param  xPos   the x position (0.0 is left, 1.0 is right, 0.5 is centered)
     * @param  yPos   the y position (0.0 is top, 1.0 is bottom, 0.5 is centered)
     */
    public static void positionFrame(final Component frame, final double xPos, final double yPos) {

        Dimension screen;
        Dimension size;
        int xPixel;
        int yPixel;

        screen = Toolkit.getDefaultToolkit().getScreenSize();
        size = frame.getSize();

        xPixel = (int) ((screen.width - size.width) * xPos);
        yPixel = (int) ((screen.height - size.height) * yPos);

        frame.setLocation(xPixel, yPixel);
    }
}
```

E.1.10 General utilities (com.srbenoit.util)

This package collects many one-off general utilities to simplify common functions or provide functionality missing in the JDK.

```java
package com.srbenoit.util;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;
import java.util.jar.JarInputStream;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * A class with a static method that can locate the set of classes installed under a given package.
 * To use, pass the package name to <code>scanClasses</code>, which returns a list of the matching
 * classes.
 */
public final class ClassList extends LoggedObject {

    /**
     * Private constructor to prevent instantiation.
     */
    private ClassList() {

        super();
    }

    /**
     * Scans for all classes installed under a particular path.
     *
     * @param   root  the root path in which to search for classes
     * @return  the list of classes found, or <code>null</code> on any error
     */
    public static List<Class<?>> scanClasses(final String root) {

        List<Class<?>> list;
        ClassLoader loader;

        list = new ArrayList<Class<?>>(20);
        loader = Thread.currentThread().getContextClassLoader();

        if (loader instanceof java.net.URLClassLoader) {

            for (URL url : ((java.net.URLClassLoader) loader).getURLs()) {

                if (url.toString().startsWith("file:")) {
                    scanFilename(url.getFile(), root, list, loader);
                } else {
                    scanUrl(url, root, list, loader);
                }
            }
        } else {

            for (String path
                : System.getProperty("java.class.path", "").split(
                    (File.separatorChar == '\\') ? "\\\\" : File.separator)) {
                scanFilename(path, root, list, loader);
            }
        }

        return list;
    }

    /**
     * Scans for all classes installed under a particular URL.
     *
     * @param  url     the URL to scan
     * @param  path    the path in which to search for classes
     * @param  list    the list to which to accumulate results
     * @param  loader  the class loader to use to locate resources
     */
    private static void scanUrl(final URL url, final String path, final List<Class<?>> list,
        final ClassLoader loader) {

        byte[] buf;
        ByteArrayOutputStream baos;
        InputStream input;
        int size;
        JarInputStream jis;
        JarEntry entry;

        try {
```

```
            // Download the JAR file
            buf = new byte[1024];
            baos = new ByteArrayOutputStream();
            input = url.openStream();
            size = input.read(buf);

            while (size != -1) {
                baos.write(buf, 0, size);
                size = input.read(buf);
            }

            input.close();

            jis = new JarInputStream(new ByteArrayInputStream(baos.toByteArray()));

            entry = jis.getNextJarEntry();

            while (entry != null) {

                if (entry.getName().endsWith(".class")) {
                    processClass(entry.getName(), path, loader, list);
                }

                entry = jis.getNextJarEntry();
            }
        } catch (IOException e1) {
            LOG.log(Level.WARNING, "Could not download the JAR file ''{0}'': {1}",
                new Object[] { url, e1.getMessage() });
        }
    }

    /**
     * Scans for all classes installed under a particular path in a single JAR file.
     *
     * @param  fname    the filename to scan
     * @param  path     the path in which to search for classes
     * @param  list     the list to which to accumulate results
     * @param  loader   the class loader to use to locate resources
     */
    private static void scanFilename(final String fname, final String path,
        final List<Class<?>> list, final ClassLoader loader) {

        File file;

        if (fname.endsWith(".jar")) {
            scanJarFile(fname, path, loader, list);
        } else {
            file = new File(fname);

            if (file.isDirectory()) {
                recurseDirectory(file.getAbsolutePath(), file, path, loader, list);
            }
        }
    }

    /**
     * Recurses a directory searching for files whose names end in ".class".
     *
     * @param  root     the root of the directory being searched
     * @param  dir      the directory to search
     * @param  path     the path being searched for
     * @param  loader   the ClassLoader to use
     * @param  list     the list to which to add matches
     */
    private static void recurseDirectory(final String root, final File dir, final String path,
        final ClassLoader loader, final List<Class<?>> list) {

        String absPath;

        for (File f : dir.listFiles()) {

            if (f.isDirectory()) {
                recurseDirectory(root, f, path, loader, list);
            } else {
                absPath = f.getAbsolutePath();

                if (absPath.startsWith(root)) {
                    absPath = absPath.substring(root.length() + 1);
                }

                processClass(absPath.replace(File.separatorChar, '.'), path, loader, list);
            }
        }
    }

    /**
```

```
     * Scans a single JAR file for class entries matching the search path.
     *
     * @param  fname    the name of the Jar file to search
     * @param  path     the path being searched for
     * @param  loader   the ClassLoader to use
     * @param  list     the list to which to add matches
     */
    private static void scanJarFile(final String fname, final String path,
        final ClassLoader loader, final List<Class<?>> list) {

        JarFile jar;
        Enumeration<JarEntry> entries;
        String entry;

        try {
            jar = new JarFile(fname.replace("%20", " "));
            entries = jar.entries();

            while (entries.hasMoreElements()) {
                entry = entries.nextElement().getName();

                if (entry.endsWith(".class")) {
                    processClass(entry, path, loader, list);
                }
            }

            try {
                jar.close();
            } catch (IOException e) {
                LOG.log(Level.WARNING,
                    "The module jar file ''{0}'' could not be closed. Error: {1}",
                    new Object[] { fname, e.getMessage() });
            }
        } catch (Exception e) {
            LOG.log(Level.WARNING,
                "jar file ''{0}'' could not be instantiated from file path. Error: {1}",
                new Object[] { fname, e.getMessage() });
        }
    }

    /**
     * Given the filename of a single class, tries to load that class and checks it against the
     * search path.
     *
     * @param  fname    the class filename
     * @param  path     the path being searched for
     * @param  loader   the ClassLoader to use
     * @param  list     the list to which to add matches
     */
    private static void processClass(final String fname, final String path,
        final ClassLoader loader, final List<Class<?>> list) {

        String name;
        Class<?> theClass;
        int pos;

        name = fname.replace('/', '.').substring(0, fname.length() - 6);

        // Classes not in the search path are ignored.
        pos = name.indexOf(path);

        if (pos != -1) {

            // Ignore classes deeper under the search path
            pos = name.indexOf('.', pos + path.length() + 1);

            if (pos == -1) {

                try {
                    theClass = Class.forName(name, false, loader);

                    if (!theClass.isInterface()) {
                        list.add(theClass);
                    }
                } catch (ClassNotFoundException nfe) {
                    LOG.log(Level.WARNING, "Skipping class ''{0}'' for reason {1}",
                        new Object[] { name, nfe.getMessage() });
                } catch (NoClassDefFoundError e) {
                    LOG.log(Level.WARNING, "Skipping class ''{0}'' for reason {1}",
                        new Object[] { name, e.getMessage() });
                }
            }
        }
    }

    /**
     * Main method to exercise the <code>ClassList</code> class.
```

```
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        List<Class<?>> cls;

        cls = ClassList.scanClasses("com.bekenlearning.db.data");

        for (Class<?> clazz : cls) {
            LOG.info(clazz.getSimpleName());
        }
    }
}

package com.srbenoit.util;

import java.io.File;
import javax.swing.filechooser.FileFilter;

/**
 * A file filter that allows only directories.
 */
public class DirectoryFilter extends FileFilter {

    /**
     * Gets the description of the filter.
     *
     * @return  the description of the filter
     */
    @Override public String getDescription() {

        return "Directories";
    }

    /**
     * The filter function, which accepts only directories.
     *
     * @param   file  the file being tested
     * @return  <code>true</code> if the file is a directory; <code>false</code> otherwise
     */
    @Override public boolean accept(final File file) {

        return file.isDirectory();
    }
}

package com.srbenoit.util;

/**
 * An exception during evaluation.
 */
public class EvaluationException extends Exception {

    /** version number for serialization */
    private static final long serialVersionUID = 2880651736185522023L;

    /**
     * Constructs a new <code>EvaluationException</code>.
     *
     * @param  message  the exception message
     */
    public EvaluationException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>EvaluationException</code>.
     *
     * @param  message  the exception message
     * @param  cause    the exception that led to this exception
     */
    public EvaluationException(final String message, final Throwable cause) {

        super(message, cause);
    }
}

package com.srbenoit.util;

/**
 * A set of fields storing a local date and time (does not store a Java timestamp or a time zone).
 */
public class LocalTime {

    /** error message when time string is invalid */
```

```
private static final String ERR = "Invlalid LocalTime string";

/** an integer that indicates a field has no value */
public static final int NO_VALUE = -1;

/** the year (range 1-9999), -1 means no value */
private int year = NO_VALUE;

/** the month, where January is 1 (range 1-12), -1 means no value */
private int month = NO_VALUE;

/** the day of the month (range 1-31), -1 means no value */
private int day = NO_VALUE;

/** the hour of the day (range 0-23), -1 means no value */
private int hour = NO_VALUE;

/** the minute of the hour (range 0-59), -1 means no value */
private int minute = NO_VALUE;

/** the second (range 0-59), -1 means no value */
private int second = NO_VALUE;

/** milliseconds (range 0-999), -1 means no value */
private int millis = NO_VALUE;

/**
 * Sets the year value.
 *
 * @param   newYear  the new year value, in the range 0000 to 9999, or NO_VALUE to clear the
 *                   year field
 * @throws  IllegalArgumentException  if the supplied year value is not either <code>
 *                                    NO_VALUE</code> or in the range 0-9999.
 */
public void setYear(final int newYear) {

    if ((newYear == NO_VALUE) || ((newYear >= 0) && (newYear <= 9999))) {
        this.year = newYear;
    } else {
        throw new IllegalArgumentException("year value out of range");
    }
}

/**
 * Gets the current year value.
 *
 * @return  the current year value, which may be <code>NO_VALUE</code>
 */
public int getYear() {

    return this.year;
}

/**
 * Sets the month value.
 *
 * @param   newMonth  the new month value, in the range 1 to 12, or NO_VALUE to clear the month
 *                    field
 * @throws  IllegalArgumentException  if the supplied month value is not either <code>
 *                                    NO_VALUE</code> or in the range 1-12.
 */
public void setMonth(final int newMonth) {

    if ((newMonth == NO_VALUE) || ((newMonth >= 1) && (newMonth <= 12))) {
        this.month = newMonth;
    } else {
        throw new IllegalArgumentException("month value out of range");
    }
}

/**
 * Gets the current month value.
 *
 * @return  the current month value, which may be <code>NO_VALUE</code>
 */
public int getMonth() {

    return this.month;
}

/**
 * Sets the day value.
 *
 * @param   newDay  the new day value, or NO_VALUE to clear the day field
 * @throws  IllegalArgumentException  if the supplied day value is not either <code>
 *                                    NO_VALUE</code> or in the range 1-31.
 */
```

```
public void setDay(final int newDay) {

    if ((newDay == NO_VALUE) || ((newDay >= 1) && (newDay <= 31))) {
        this.day = newDay;
    } else {
        throw new IllegalArgumentException("day_value_out_of_range");
    }
}

/**
 * Gets the current day value.
 *
 * @return  the current day value, which may be <code>NO_VALUE</code>
 */
public int getDay() {

    return this.day;
}

/**
 * Sets the hour value.
 *
 * @param   newHour  the new hour value, or NO_VALUE to clear the hour field
 * @throws  IllegalArgumentException  if the supplied hour value is not either <code>
 *                                    NO_VALUE</code> or in the range 0-23.
 */
public void setHour(final int newHour) {

    if ((newHour == NO_VALUE) || ((newHour >= 0) && (newHour <= 23))) {
        this.hour = newHour;
    } else {
        throw new IllegalArgumentException("hour_value_out_of_range");
    }
}

/**
 * Gets the current hour value.
 *
 * @return  the current hour value, which may be <code>NO_VALUE</code>
 */
public int getHour() {

    return this.year;
}

/**
 * Sets the minute value.
 *
 * @param   newMinute  the new minute value, or NO_VALUE to clear the minute field
 * @throws  IllegalArgumentException  if the supplied minute value is not either <code>
 *                                    NO_VALUE</code> or in the range 0-59.
 */
public void setMinute(final int newMinute) {

    if ((newMinute == NO_VALUE) || ((newMinute >= 0) && (newMinute <= 59))) {
        this.minute = newMinute;
    } else {
        throw new IllegalArgumentException("minute_value_out_of_range");
    }
}

/**
 * Gets the current minute value.
 *
 * @return  the current minute value, which may be <code>NO_VALUE</code>
 */
public int getMinute() {

    return this.minute;
}

/**
 * Sets the second value.
 *
 * @param   newSecond  the new second value, or NO_VALUE to clear the second field
 * @throws  IllegalArgumentException  if the supplied second value is not either <code>
 *                                    NO_VALUE</code> or in the range 0-59.
 */
public void setSecond(final int newSecond) {

    if ((newSecond == NO_VALUE) || ((newSecond >= 0) && (newSecond <= 59))) {
        this.second = newSecond;
    } else {
        throw new IllegalArgumentException("second_value_out_of_range");
    }
}
```

```
/**
 * Gets the current second value.
 *
 * @return  the current second value, which may be <code>NO_VALUE</code>
 */
public int getSecond() {

    return this.second;
}

/**
 * Sets the milliseconds value.
 *
 * @param   newMillis  the new milliseconds value, or NO_VALUE to clear the milliseconds field
 * @throws  IllegalArgumentException  if the supplied milliseconds value is not either <code>
 *                                    NO_VALUE</code> or in the range 0-999.
 */
public void setMillis(final int newMillis) {

    if ((newMillis == NO_VALUE) || ((newMillis >= 0) && (newMillis <= 999))) {
        this.millis = newMillis;
    } else {
        throw new IllegalArgumentException("year_value_out_of_range");
    }
}

/**
 * Gets the current milliseconds value.
 *
 * @return  the current milliseconds value, which may be <code>NO_VALUE</code>
 */
public int getMillis() {

    return this.millis;
}

/**
 * Tests whether the date is completely set (that is, if the month, day, and year fields all
 * have values other than <code>No_VALUE</code>.
 *
 * @return  <code>true</code> if the date is completely set; <code>false</code> otherwise
 */
public boolean isDateSet() {

    return (this.year != NO_VALUE) && (this.month != NO_VALUE) && (this.day != NO_VALUE);
}

/**
 * Tests whether the time is completely set (that is, if the hour, minute, and second fields
 * all have values other than <code>No_VALUE</code>.
 *
 * @return  <code>true</code> if the time is completely set; <code>false</code> otherwise
 */
public boolean isTimeSet() {

    return (this.hour != NO_VALUE) && (this.minute != NO_VALUE) && (this.second != NO_VALUE);
}

/**
 * Generates the string representation of the local time. There are several permitted formats
 * for the <code>String</code>:
 *
 * <p>If neither date nor time is populated, returns 'never'.
 *
 * <p>If the date is complete, but time is not, the output will be a 10-character string in the
 * format 'MM/DD/YYYY'.
 *
 * <p>If the time is complete but the date is not, the output will be either an 8-character
 * string of format 'hh:mm:ss' (if milliseconds field is not set), or a 12-character string of
 * format 'hh:mm:ss.uuu' (if milliseconds field is set).
 *
 * <p>If date and time are both complete, the output will be either a 19-character string of
 * format 'MM/DD/YYYY hh:mm:ss' (if milliseconds field is not set), or a 23-character string of
 * format 'MM/DD/YYYY hh:mm:ss.uuu' (if milliseconds field is sets).
 *
 * @return  the string representation
 */
@Override public String toString() {

    StringBuilder str;

    str = new StringBuilder(30);

    if (isDateSet()) {

        if (isTimeSet()) {
            appendDate(str);
```

```
            str.append('␣');
            appendTime(str);
        } else {
            appendDate(str);
        }
    } else if (isTimeSet()) {
        appendTime(str);
    } else {
        str.append("never");
    }

    return str.toString();
}

/**
 * Appends the date to a <code>StringBuilder</code>, in the format 'MM/DD/YYYY'.
 *
 * @param  str  the <code>StringBuilder</code> to which to append
 */
private void appendDate(final StringBuilder str) {

    str.append(Integer.toString((this.day / 10) % 10));
    str.append(Integer.toString(this.day % 10));
    str.append('/');
    str.append(Integer.toString((this.month / 10) % 10));
    str.append(Integer.toString(this.month % 10));
    str.append('/');
    str.append(Integer.toString((this.year / 1000) % 10));
    str.append(Integer.toString((this.year / 100) % 10));
    str.append(Integer.toString((this.year / 10) % 10));
    str.append(Integer.toString(this.year % 10));
}

/**
 * Appends the time to a <code>StringBuilder</code>, in the format 'hh:mm:ss' or
 * 'hh:mm:ss.uuu'.
 *
 * @param  str  the <code>StringBuilder</code> to which to append
 */
private void appendTime(final StringBuilder str) {

    str.append(Integer.toString((this.hour / 10) % 10));
    str.append(Integer.toString(this.hour % 10));
    str.append(':');
    str.append(Integer.toString((this.minute / 10) % 10));
    str.append(Integer.toString(this.minute % 10));
    str.append(':');
    str.append(Integer.toString((this.second / 10) % 10));
    str.append(Integer.toString(this.second % 10));

    if (this.millis != NO_VALUE) {
        str.append('.');
        str.append(Integer.toString((this.millis / 100) % 10));
        str.append(Integer.toString((this.millis / 10) % 10));
        str.append(Integer.toString(this.millis % 10));
    }
}

/**
 * Parses a <code>LocalTime</code> from a String representation.
 *
 * @param   str  the string to parse
 * @return  the parsed <code>LocalTime</code> object.
 * @throws  IllegalArgumentException  if the strung could not be parsed.
 */
public static LocalTime parse(final String str) throws IllegalArgumentException {

    LocalTime obj;

    if ("never".equals(str)) {
        obj = new LocalTime();
    } else {

        switch (str.length()) {

        case 8: // hh:mm:ss
            if ((str.charAt(2) == ':') && (str.charAt(5) == ':')) {
                obj = new LocalTime();
                parseHMS(obj, str, 0);
            } else {
                throw new IllegalArgumentException(ERR);
            }

            break;

        case 10: // MM/DD/YYYY
            if ((str.charAt(2) == '/') && (str.charAt(5) == '/')) {
```

```
                obj = new LocalTime();
                parseMDY(obj, str, 0);
            } else {
                throw new IllegalArgumentException(ERR);
            }

            break;

        case 12: // hh:mm:ss.uuuu
            if ((str.charAt(2) == ':') && (str.charAt(5) == ':') && (str.charAt(8) == '.')) {
                obj = new LocalTime();
                parseHMSU(obj, str, 0);
            } else {
                throw new IllegalArgumentException(ERR);
            }

            break;

        case 19: // MM/DD/YYYY hh:mm:ss
            if ((str.charAt(2) == '/') && (str.charAt(5) == '/') && (str.charAt(10) == ' ')
                    && (str.charAt(13) == ':') && (str.charAt(16) == ':')) {
                obj = new LocalTime();
                parseMDY(obj, str, 0);
                parseHMS(obj, str, 11);
            } else {
                throw new IllegalArgumentException(ERR);
            }

            break;

        case 23: // MM/DD/YYYY hh:mm:ss.uuuu
            if ((str.charAt(2) == '/') && (str.charAt(5) == '/') && (str.charAt(10) == ' ')
                    && (str.charAt(13) == ':') && (str.charAt(16) == ':')
                    && (str.charAt(19) == '.')) {
                obj = new LocalTime();
                parseMDY(obj, str, 0);
                parseHMSU(obj, str, 11);
            } else {
                throw new IllegalArgumentException(ERR);
            }

            break;

        default:
            throw new IllegalArgumentException(ERR);
        }
    }

    return obj;
}

/**
 * Parses an hour-minute-second value from a string of the format 'hh:mm:ss'.
 *
 * @param   obj     the object into which to place parsed values
 * @param   str     the string to parse
 * @param   index   the index of the start of the value in the string
 * @throws  IllegalArgumentException  if the value cannot be parsed
 */
private static void parseHMS(final LocalTime obj, final String str, final int index)
    throws IllegalArgumentException {

    try {
        obj.setHour(Integer.parseInt(str.substring(index, index + 2)));
        obj.setMinute(Integer.parseInt(str.substring(index + 3, index + 5)));
        obj.setSecond(Integer.parseInt(str.substring(index + 6, index + 8)));
    } catch (NumberFormatException e) {
        throw new IllegalArgumentException(ERR, e);
    }
}

/**
 * Parses an hour-minute-second-microsecond value from a string of the format 'hh:mm:ss.uuuu'.
 *
 * @param   obj     the object into which to place parsed values
 * @param   str     the string to parse
 * @param   index   the index of the start of the value in the string
 * @throws  IllegalArgumentException  if the value cannot be parsed
 */
private static void parseHMSU(final LocalTime obj, final String str, final int index)
    throws IllegalArgumentException {

    try {
        obj.setHour(Integer.parseInt(str.substring(index, index + 2)));
        obj.setMinute(Integer.parseInt(str.substring(index + 3, index + 5)));
        obj.setSecond(Integer.parseInt(str.substring(index + 6, index + 8)));
        obj.setMillis(Integer.parseInt(str.substring(index + 9, index + 12)));
```

```
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(ERR, e);
        }
    }

    /**
     * Parses an month-day-year value from a string of the format 'MM/DD/YYYY'.
     *
     * @param    obj     the object into which to place parsed values
     * @param    str     the string to parse
     * @param    index   the index of the start of the value in the string
     * @throws   IllegalArgumentException   if the value cannot be parsed
     */
    private static void parseMDY(final LocalTime obj, final String str, final int index)
        throws IllegalArgumentException {

        try {
            obj.setMonth(Integer.parseInt(str.substring(index, index + 2)));
            obj.setDay(Integer.parseInt(str.substring(index + 3, index + 5)));
            obj.setYear(Integer.parseInt(str.substring(index + 6, index + 10)));
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException(ERR, e);
        }
    }
}

package com.srbenoit.util;

import java.util.MissingResourceException;
import java.util.ResourceBundle;

/**
 * Provides general access to a set of localized resources for a class. This class assumes that
 * corresponding to a class "foo.bar.Baz" there is a localized resource bundle named
 * "foo.bar.Baz.messages".
 */
public class Messages {

    /** the class for which messages will be retrieved */
    private final transient Class<?> ownerClass;

    /** the loaded resource bundle */
    private final transient ResourceBundle resBundle;

    /**
     * Construct a <code>Messages</code>.
     *
     * @param  clazz  the class of the caller. The resource name is constructed by appending
     *                ".messages" to the class's owning package. Therefore a file called
     *                "messages.properties" (or any localized variant) in the class's package will
     *                be loaded.
     */
    public Messages(final Class<?> clazz) {

        String name;

        this.ownerClass = clazz;

        name = clazz.getPackage().getName() + ".messages";

        this.resBundle = ResourceBundle.getBundle(name);
    }

    /**
     * Tests whether a named string is defined.
     *
     * @param   key  the message key
     * @return  <code>true</code> if a message is defined for the given class and key; <code>
     *          false</code> otherwise
     */
    public boolean exists(final String key) {

        String msgKey;
        boolean found;

        msgKey = this.ownerClass.getSimpleName() + '.' + key;

        try {
            this.resBundle.getString(msgKey);
            found = true;
        } catch (MissingResourceException e) {
            found = false;
        }

        return found;
    }
```

```
    /**
     * Gets a named string.
     *
     * @param   key  the message key
     * @return  the <code>String</code>, or a filler <code>String</code> if there was no <code>
     *          String</code> defined with the specified key
     */
    public String getString(final int key) {

        String msgKey;
        String value;

        msgKey = this.ownerClass.getSimpleName() + '.' + key;

        try {
            value = this.resBundle.getString(msgKey);
        } catch (MissingResourceException e) {
            value = '!' + msgKey + '!';
        }

        return value;
    }

    /**
     * Gets a named array of strings.
     *
     * @param   key  the message key
     * @return  the <code>String</code> array, or an empty array if there was not at least one
     *          <code>String</code> defined with the specified key
     */
    public String[] getStrings(final int key) {

        int count;
        String msgKey;
        String[] values;

        // Count the number of indexed strings found
        count = 0;

        for (int i = 1; i < 999; i++) {
            msgKey = this.ownerClass.getSimpleName() + '.' + key + '.' + Integer.toString(i);

            if (exists(msgKey)) {
                count++;
            }
        }

        if (count == 0) {
            msgKey = this.ownerClass.getSimpleName() + '.' + key;

            if (exists(msgKey)) {
                values = new String[1];

                try {
                    values[0] = this.resBundle.getString(msgKey);
                } catch (MissingResourceException e) {
                    values[0] = '!' + msgKey + '!';
                }
            } else {
                values = new String[0];
            }
        } else {
            values = new String[count];

            for (int i = 1; i <= count; i++) {
                msgKey = this.ownerClass.getSimpleName() + '.' + key + '.' + Integer.toString(i);

                try {
                    values[i - 1] = this.resBundle.getString(msgKey);
                } catch (MissingResourceException e) {
                    values[i - 1] = '!' + msgKey + '!';
                }
            }
        }

        return values;
    }
}

package com.srbenoit.util;

import java.awt.image.BufferedImage;
import java.io.BufferedReader;
import java.io.ByteArrayOutputStream;
import java.io.Closeable;
import java.io.File;
import java.io.FileInputStream;
```

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Locale;
import java.util.Properties;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import com.srbenoit.log.LoggedObject;

/**
 * Utility class to load resources, line-oriented files, properties files, and images. This class
 * should be able to load from a local file or a file in a JAR in the classpath, using a given
 * class a the base of the package in which to search for the resource.
 */
public final class ResourceLoader extends LoggedObject {

    /** the file extension for properties files */
    private static final String PROPS_EXTENSION = ".properties";

    /**
     * Private constructor to prevent instantiation.
     */
    private ResourceLoader() {

        super();
    }

    /**
     * Loads a text file, storing the file contents in a <code>String</code>. Lines in the returned
     * file are separated by single '\n' characters regardless of the line terminator in the source
     * file. The last line will end with a '\n' character, whether or not there was one in the
     * input file.
     *
     * @param   file  the file to read
     * @return  the loaded file contents, or <code>null</code> if unable to load
     */
    public static String loadFile(final File file) {

        InputStream input;
        BufferedReader reader;
        String line;
        StringBuilder builder;
        String result = null;

        try {
            input = new FileInputStream(file);

            try {
                reader = new BufferedReader(new InputStreamReader(input));

                builder = new StringBuilder((int) file.length());

                try {
                    line = reader.readLine();

                    while (line != null) {
                        builder.append(line);
                        builder.append("\n");
                        line = reader.readLine();
                    }

                    result = builder.toString();
                } catch (Exception e) {
                    logException("load file 1", file.getName(), e);
                }

                close(reader, file.getName());
            } catch (Exception e) {
                logException("load file 2", file.getName(), e);
            } finally {
                close(input, file.getName());
            }
        } catch (Exception e) {
            logException("load file 3", file.getName(), e);
        }

        return result;
    }

    /**
     * Loads a text file, storing the file contents in a <code>String</code>. Lines in the returned
     * file are separated by single '\n' characters regardless of the line terminator in the source
     * file. The last line will end with a '\n' character, whether or not there was one in the
     * input file.
     *
```

```
     * @param   caller  the class of the object making the call, so that relative resource paths
     *                  are based on the caller's position in the source tree
     * @param   name    the name of the file to read
     * @return  the loaded file contents, or <code>null</code> if unable to load
     */
    public static String loadFile(final Class<?> caller, final String name) {

        InputStream input;
        BufferedReader reader;
        String line;
        StringBuilder builder;
        String result = null;

        input = openInputStream(caller, name, true);

        if (input != null) {

            try {
                reader = new BufferedReader(new InputStreamReader(input));

                builder = new StringBuilder(1024);

                try {
                    line = reader.readLine();

                    while (line != null) {
                        builder.append(line);
                        builder.append("\n");
                        line = reader.readLine();
                    }

                    result = builder.toString();
                } catch (Exception e) {
                    logException("load file 4", name, e);
                }

                close(reader, name);
            } catch (Exception e) {
                logException("load file 5", name, e);
            } finally {
                close(input, name);
            }
        }

        return result;
    }

    /**
     * Loads a text file, storing the resulting lines of text in a <code>String</code> array.
     *
     * @param   caller  the class of the object making the call, so that relative resource paths
     *                  are based on the caller's position in the source tree
     * @param   name    the name of the file to read
     * @return  the loaded file contents, or <code>null</code> if unable to load
     */
    public static String[] loadFileLines(final Class<?> caller, final String name) {

        InputStream input;
        BufferedReader reader;
        String line;
        ArrayList<String> lines;
        String[] result;

        input = getInputStream(caller, name);

        if (input == null) {
            result = null;
        } else {

            try {
                reader = new BufferedReader(new InputStreamReader(input));

                lines = new ArrayList<String>(100);

                try {
                    line = reader.readLine();

                    while (line != null) {
                        lines.add(line);
                        line = reader.readLine();
                    }

                    result = new String[lines.size()];
                    lines.toArray(result);
                } catch (Exception e) {
                    logException("read from", name, e);
                    result = null;
```

```
            } finally {
                close(reader, name);
            }
        } catch (Exception e) {
            logException("create reader for", name, e);
            result = null;
        } finally {
            close(input, name);
        }
    }

    return result;
}

/**
 * Loads a binary file, storing the resulting data in as<code>byte</code> array.
 *
 * @param   caller  the class of the object making the call, so that relative resource paths
 *                  are based on the caller's position in the source tree
 * @param   name    the name of the file to read
 * @return  the loaded file contents, or <code>null</code> if unable to load
 */
public static byte[] loadFileBytes(final Class<?> caller, final String name) {

    InputStream input;
    ByteArrayOutputStream baos;
    byte[] buffer;
    int count;
    byte[] result;

    input = getInputStream(caller, name);

    if (input == null) {
        result = null;
    } else {
        buffer = new byte[1024];
        baos = new ByteArrayOutputStream();

        try {
            count = input.read(buffer);

            while (count != -1) {
                baos.write(buffer, 0, count);
                count = input.read(buffer);
            }

            baos.close();
            result = baos.toByteArray();
        } catch (Exception e) {
            logException("read from", name, e);
            result = null;
        }

        close(input, name);
    }

    return result;
}

/**
 * Reads a single image file into a buffered image.
 *
 * @param   caller  the class of the object making the call, so that relative resource paths
 *                  are based on the caller's position in the source tree
 * @param   path    the resource path of the image file
 * @return  the loaded buffered image
 */
public static BufferedImage loadImage(final Class<?> caller, final String path) {

    InputStream input;
    BufferedImage img = null;

    input = openInputStream(caller, path, true);

    if (input != null) {

        try {
            img = ImageIO.read(input);
        } catch (Exception e) {
            logException("load image", path, e);
        } finally {
            close(input, path);
        }
    }

    return img;
}
```

```
/**
 * Finds and opens the appropriate resource bundle for a given locale. This locale may change
 * if the user selects different languages from the interface, in which case the GUI will be
 * rebuilt with the new settings.
 *
 * <p>If the resource bundle cannot be loaded for any reason, a set of default settings will be
 * generated and returned.
 *
 * @param   dir   the directory in which to find the properties file
 * @param   base  the base name (without language extension) of the properties file
 * @return  the opened resources, as a <code>Properties</code> object
 */
public static Properties loadProperties(final File dir, final String base) {

    Locale locale;
    String path;
    InputStream input;
    Properties res = null;

    locale = Locale.getDefault();

    // Now, look for a file qualified with the locale name, then for a file
    // with no qualifications.
    path = base + "_" + locale.getLanguage() + PROPS_EXTENSION;
    input = openInputStream(dir, path, false);

    if (input == null) {
        path = base + PROPS_EXTENSION;
        input = openInputStream(dir, path, true);
    }

    if (input != null) {
        res = new Properties();

        try {
            res.load(input);
        } catch (IOException e) {
            logException("load_properties", path, e);
        } finally {
            close(input, path);
        }
    }

    return res;
}

/**
 * Finds and opens the appropriate resource bundle for a given locale. This locale may change
 * if the user selects different languages from the interface, in which case the GUI will be
 * rebuilt with the new settings.
 *
 * <p>If the resource bundle cannot be loaded for any reason, a set of default settings will be
 * generated and returned.
 *
 * @param   caller  the class of the object making the call, so that relative resource paths
 *                  are based on the caller's position in the source tree
 * @param   base    the base name (without language extension) of the resource bundle
 * @return  the opened resources, as a <code>Properties</code> object
 */
public static Properties loadProperties(final Class<?> caller, final String base) {

    Locale locale;
    String path;
    InputStream input;
    Properties res = null;

    locale = Locale.getDefault();

    // Now, look for a file qualified with the locale name, then for a file
    // with no qualifications.
    path = base + "_" + locale.getLanguage() + PROPS_EXTENSION;
    input = openInputStream(caller, path, false);

    if (input == null) {
        path = base + PROPS_EXTENSION;
        input = openInputStream(caller, path, true);
    }

    if (input != null) {
        res = new Properties();

        try {
            res.load(input);
        } catch (IOException e) {
            logException("load_properties", path, e);
        } finally {
```

```
            close(input, path);
        }
    }

    return res;
}

/**
 * Finds and opens the appropriate resource bundle for a given locale. This locale may change
 * if the user selects different languages from the interface, in which case the GUI will be
 * rebuilt with the new settings.
 *
 * <p>If the resource bundle cannot be loaded for any reason, a set of default settings will be
 * generated and returned.
 *
 * @param    caller  the class of the object making the call, so that relative resource paths
 *                   are based on the caller's position in the source tree
 * @param    base    the base name (without language extension) of the resource bundle
 * @param    def     a set of defaults in case the resources could not be found
 * @return   the opened resources, as a <code>Properties</code> object
 */
public static Properties loadProperties(final Class<?> caller, final String base,
    final Properties def) {

    Locale locale;
    String path;
    InputStream input;
    Properties res = null;

    locale = Locale.getDefault();

    // Now, look for a file qualified with the locale name, then for a file
    // with no qualifications.
    path = base + "_" + locale.getLanguage() + PROPS_EXTENSION;
    input = openInputStream(caller, path, false);

    if (input == null) {
        path = base + PROPS_EXTENSION;
        input = openInputStream(caller, path, true);
    }

    if (input != null) {
        res = new Properties(def);

        try {
            res.load(input);
        } catch (IOException e) {
            logException("load_properties", path, e);
        } finally {
            close(input, path);
        }
    }

    if (res == null) {
        res = def; // Use defaults if loading failed
    }

    return res;
}

/**
 * Obtains an input stream for a particular resource.
 *
 * @param    caller  the class of the object making the call, so that relative resource paths
 *                   are based on the caller's position in the source tree
 * @param    name    the name of the resource to read
 * @return   the input stream, or null if unable to open
 */
public static InputStream getInputStream(final Class<?> caller, final String name) {

    return openInputStream(caller, name, true);
}

/**
 * Obtains an input stream for a particular resource.
 *
 * @param    caller      the class of the object making the call, so that relative resource paths
 *                       are based on the caller's position in the source tree
 * @param    name        the name of the resource to read
 * @param    logErrors   true to log errors; false otherwise
 * @return   the input stream, or null if unable to open
 */
public static InputStream getInputStream(final Class<?> caller, final String name,
    final boolean logErrors) {

    return openInputStream(caller, name, logErrors);
}
```

```
/**
 * Obtains an input stream for a particular resource.
 *
 * @param    caller      the class of the object making the call, so that relative resource paths
 *                       are based on the caller's position in the source tree
 * @param    name        the name of the resource to read
 * @param    logErrors   true to log errors; false otherwise
 * @return   the input stream, or null if unable to open
 */
private static InputStream openInputStream(final Class<?> caller, final String name,
    final boolean logErrors) {

    InputStream input;
    File file;

    input = caller.getResourceAsStream(name);

    if (input == null) {
        input = Thread.currentThread().getContextClassLoader().getResourceAsStream(name);
    }

    if (input == null) {
        input = ClassLoader.getSystemResourceAsStream(name);
    }

    if (input == null) {
        file = new File(System.getProperty("user.dir"));

        try {
            input = new FileInputStream(new File(file, name));
        } catch (FileNotFoundException e) {
            input = null;
        }
    }

    if ((input == null) && logErrors) {
        LOG.log(Level.WARNING, "Failed to load: {0}", name);
    }

    return input;
}

/**
 * Obtains an input stream for a particular resource.
 *
 * @param    dir         the directory in which to locate the file to be opened
 * @param    name        the name of the resource to read
 * @param    logErrors   true to log errors; false otherwise
 * @return   the input stream, or null if unable to open
 */
private static InputStream openInputStream(final File dir, final String name,
    final boolean logErrors) {

    File file;
    InputStream input = null;

    file = new File(dir, name);

    try {
        input = new FileInputStream(file);
    } catch (FileNotFoundException e) {

        if (logErrors) {
            LOG.log(Level.WARNING, "Failed to load: {0}", file.getAbsolutePath());
        }
    }

    return input;
}

/**
 * Closes an input stream.
 *
 * @param  input   the stream to close
 * @param  name    the name of the resource being closed
 */
private static void close(final Closeable input, final String name) {

    try {
        input.close();
    } catch (IOException e) {
        logException("close", name, e);
    }
}

/**
```

```
     * Logs an exception encountered by the <code>ResourceLoader</code>.
     *
     * @param  operation  the operation that was being attempted
     * @param  name       the name of the resource
     * @param  exc        the exception
     */
    private static void logException(final String operation, final String name,
        final Exception exc) {

        LOG.log(Level.WARNING, "ResourceLoader failed to {0} ''{1}'': {2}",
            new Object[] { operation, name, exc.getLocalizedMessage() });
    }
}

package com.srbenoit.util;

import java.util.Calendar;

/**
 * A thread-safe singleton class to perform calendar operations. This is needed as the <code>
 * Calendar</code> class is not thread safe.
 */
public final class SharedCalendar {

    /** the calendar */
    private final static Calendar CAL;

    static {
        CAL = Calendar.getInstance();
    }

    /**
     * Private constructor to enforce singleton model.
     */
    private SharedCalendar() {

        // No action
    }

    /**
     * Gets the year of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the year
     */
    public static int yearOf(final long millis) {

        return get(Calendar.YEAR, millis);
    }

    /**
     * Gets the month of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the month
     */
    public static int monthOf(final long millis) {

        return get(Calendar.MONTH, millis) + 1;
    }

    /**
     * Gets the day of the month of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the day of the month
     */
    public static int dayOfMonthOf(final long millis) {

        return get(Calendar.DAY_OF_MONTH, millis);
    }

    /**
     * Gets the day of the year of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the day of the week
     */
    public static int dayOfYearOf(final long millis) {

        return get(Calendar.DAY_OF_YEAR, millis);
    }

    /**
     * Gets the day of the week of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
```

```
     * @return  the day of the week
     */
    public static int dayOfWeekOf(final long millis) {

        return get(Calendar.DAY_OF_WEEK, millis);
    }

    /**
     * Gets the hour of the day of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the hour of the day
     */
    public static int hourOfDayOf(final long millis) {

        return get(Calendar.HOUR_OF_DAY, millis);
    }

    /**
     * Gets the hour of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the hour (0-11)
     */
    public static int hourOf(final long millis) {

        return get(Calendar.HOUR, millis);
    }

    /**
     * Gets the minute of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the minute
     */
    public static int minuteOf(final long millis) {

        return get(Calendar.MINUTE, millis);
    }

    /**
     * Gets the second of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the second
     */
    public static int secondOf(final long millis) {

        return get(Calendar.SECOND, millis);
    }

    /**
     * Gets the millisecond of a particular timestamp.
     *
     * @param   millis  the timestamp in milliseconds
     * @return  the millisecond
     */
    public static int millisecondOf(final long millis) {

        return get(Calendar.MILLISECOND, millis);
    }

    /**
     * Gets a particular calendar field for a particular timestamp.
     *
     * @param   field   the calendar field to get
     * @param   millis  the timestamp in milliseconds
     * @return  the value of that field
     */
    private static int get(final int field, final long millis) {

        synchronized (CAL) {
            CAL.setTimeInMillis(millis);

            return CAL.get(field);
        }
    }
}

package com.srbenoit.util;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

/**
 * A thread-safe singleton class to perform date to string conversion. This is needed as the <code>
```

```
 * SimpleDateFormat</code> class is not thread safe.
 */
public final class SharedDateFormat {

    /** formatter for dates */
    private final static SimpleDateFormat DATEFORMAT;

    static {
        DATEFORMAT = new SimpleDateFormat("MM/dd/yy HH:mm:ss.SSS", Locale.getDefault());
    }

    /**
     * Private constructor to enforce singleton model.
     */
    private SharedDateFormat() {

        // No action
    }

    /**
     * Generates the string representation of a date value.
     *
     * @param   date  the date
     * @return  the String representation
     */
    public static String format(final Date date) {

        synchronized (DATEFORMAT) {
            return DATEFORMAT.format(date);
        }
    }

    /**
     * Generates the string representation of a timestamp value.
     *
     * @param   timestamp  the timestamp
     * @return  the String representation
     */
    public static String format(final long timestamp) {

        return format(new Date(timestamp));
    }
}

package com.srbenoit.util;

/**
 * Utilities for working with strings.
 */
public final class StringUtils {

    /**
     * Private construct to prevent instantiation.
     */
    private StringUtils() {

        // No action
    }

    /**
     * Tests whether a string contains only whitespace characters.
     *
     * @param   str  the string to test
     * @return  <code>true</code> if the string contains only whitespace characters (or is empty);
     *          <code>false</code> if it contains non-whitespace characters
     */
    public static boolean isOnlyWhitespace(final String str) {

        int len;
        boolean isWhitespace = true;

        len = str.length();

        for (int i = 0; i < len; i++) {

            if (Character.isWhitespace(str.charAt(i))) {
                continue;
            }

            isWhitespace = false;

            break;
        }

        return isWhitespace;
    }
}
```

```java
package com.srbenoit.util;

import java.util.Calendar;
import java.util.Date;

/**
 * A thread-safe static class to perform calendar operations. This is needed as the <code>
 * Calendar</code> class is not thread safe.
 */
public final class TimeConverter {

    /** the default calendar */
    private final static Calendar CAL;

    static {
        CAL = Calendar.getInstance();
    }

    /**
     * Private constructor to enforce singleton model.
     */
    private TimeConverter() {

        /* No action. */
    }

    /**
     * Takes only the time component of a timestamp and converts it to the same time on January 1,
     * 1970 (the "epoch").
     *
     * @param   timestamp  the timestamp whose time is to be extracted
     * @return  the extracted timestamp
     */
    public static long compressToFirstDayOfEpoch(final long timestamp) {

        long time;

        synchronized (CAL) {
            CAL.setTimeInMillis(timestamp);
            CAL.set(Calendar.YEAR, 1970);
            CAL.set(Calendar.DAY_OF_YEAR, 1);
            time = CAL.getTimeInMillis();
        }

        return time;
    }

    /**
     * Takes only the time component of a timestamp and converts it to the same time on January 1,
     * 1970 (the "epoch").
     *
     * @param   timestamp  the timestamp whose time is to be extracted
     * @return  the extracted timestamp
     */
    public static Long compressToFirstDayOfEpoch(final Long timestamp) {

        Long time;

        if (timestamp == null) {
            time = null;
        } else {
            time = Long.valueOf(compressToFirstDayOfEpoch(timestamp.longValue()));
        }

        return time;
    }

    /**
     * Takes only the time component of a timestamp and converts it to the same time on January 1,
     * 1970 (the "epoch").
     *
     * @param   timestamp  the timestamp whose time is to be extracted
     * @return  the extracted timestamp
     */
    public static Long compressToFirstDayOfEpoch(final Date timestamp) {

        Long time;

        if (timestamp == null) {
            time = null;
        } else {
            time = Long.valueOf(compressToFirstDayOfEpoch(timestamp.getTime()));
        }

        return time;
    }
```

```
/**
 * Given a year, month (1-12) and day of month (1-31), gets the timestamp in milliseconds of
 * midnight at the beginning of that day.
 *
 * @param    year         the year
 * @param    month        the month
 * @param    dayOfMonth   the day of the month
 * @return   the millisecond value
 */
public static long toMillis(final int year, final int month, final int dayOfMonth) {

    return toMillis(year, month, dayOfMonth, 0, 0, 0, 0);
}

/**
 * Given the hours, minutes, and seconds, gets the timestamp in milliseconds of that time on
 * January 1, 1970.
 *
 * @param    hourOfDay   the hour of the day
 * @param    minute      the minute
 * @param    second      the second
 * @param    millis      the milliseconds within the second
 * @return   the millisecond value
 */
public static long toMillis(final int hourOfDay, final int minute, final int second,
    final int millis) {

    return toMillis(1970, 1, 1, hourOfDay, minute, second, millis);
}

/**
 * Given a year, month (1-12), day of month (1-31), hours, minutes, and seconds, gets the
 * timestamp in milliseconds.
 *
 * @param    year         the year
 * @param    month        the month
 * @param    dayOfMonth   the day of the month
 * @param    hourOfDay    the hour of the day
 * @param    minute       the minute
 * @param    second       the second
 * @param    millis       the milliseconds within the second
 * @return   the millisecond value
 */
public static long toMillis(final int year, final int month, final int dayOfMonth,
    final int hourOfDay, final int minute, final int second, final int millis) {

    synchronized (CAL) {
        CAL.set(year, month - 1, dayOfMonth, hourOfDay, minute, second);
        CAL.set(Calendar.MILLISECOND, millis);

        return CAL.getTimeInMillis();
    }
}

/**
 * Converts a timestamp (in milliseconds) into an integer that represents the year * 1000 + the
 * day (this is an integer value that will be different for every day and is monotonically
 * increasing; useful for comparing two timestamps to see if they were on the same day or if
 * not, which occurred later).
 *
 * @param    timestamp   the timestamp to convert
 * @return   the integer day value
 */
public static int toIntegerDay(final long timestamp) {

    int day;

    synchronized (CAL) {
        CAL.setTimeInMillis(timestamp);
        day = (1000 * CAL.get(Calendar.YEAR)) + CAL.get(Calendar.DAY_OF_YEAR);
    }

    return day;
}

/**
 * Converts a date into an integer that represents the year * 1000 + the day (this is an
 * integer value that will be different for every day and is monotonically increasing; useful
 * for comparing two timestamps to see if they were on the same day or if not, which occurred
 * later).
 *
 * @param    date   the date to convert
 * @return   the integer day value
 */
public static int toIntegerDay(final Date date) {
```

```
        int day;

        synchronized (CAL) {
            CAL.setTimeInMillis(date.getTime());
            day = (1000 * CAL.get(Calendar.YEAR)) + CAL.get(Calendar.DAY_OF_YEAR);
        }

        return day;
    }

    /**
     * Converts a timestamp (in milliseconds) into an integer that represents the minute of the day
     * (0 for the first minute after midnight, and 1439 during the last minute before midnight).
     *
     * @param   timestamp  the timestamp to convert
     * @return  the integer minute value
     */
    public static int toMinuteOfDay(final long timestamp) {

        int minute;

        synchronized (CAL) {
            CAL.setTimeInMillis(timestamp);
            minute = (60 * CAL.get(Calendar.HOUR_OF_DAY)) + CAL.get(Calendar.MINUTE);
        }

        return minute;
    }

    /**
     * Converts a date into an integer that represents the minute of the day (0 for the first
     * minute after midnight, and 1439 during the last minute before midnight).
     *
     * @param   date  the date to convert
     * @return  the integer minute value
     */
    public static int toMinuteOfDay(final Date date) {

        int minute;

        synchronized (CAL) {
            CAL.setTimeInMillis(date.getTime());
            minute = (60 * CAL.get(Calendar.HOUR_OF_DAY)) + CAL.get(Calendar.MINUTE);
        }

        return minute;
    }
}
```

E.2 Media Management

E.2.1 QuickTime Movie Creation (com.srbenoit.media.movie)

This package provides classes to convert series of files or sets of buffered images into QuickTime movies. Almost every class in this package relies on the Java Media Framwork libraries. To obtain these, download and install "JMF 2.1.1e" from the Java media Framework downloads web page.

```
package com.srbenoit.media.movie;

import java.awt.image.BufferedImage;
import java.util.List;
import javax.media.MediaLocator;
import javax.media.Time;
import javax.media.protocol.ContentDescriptor;
import javax.media.protocol.PullBufferDataSource;
import javax.media.protocol.PullBufferStream;

/**
```

```
 * A <code>DataSource</code> to read from a list of JPEG image files and turn that into a stream of
 * JMF buffers. The <code>DataSource</code> is not seekable or positionable.
 */
public class BufferedImageDataSource extends PullBufferDataSource {

    /** the list of input streams */
    private final transient BufferedImageSourceStream[] streams;

    /**
     * Constructs a new <code>BufferedImageDataSource</code>.
     *
     * @param  width       the target frame width
     * @param  height      the target frame height
     * @param  frameRate   the target frame rate
     * @param  images      the list of image names
     */
    public BufferedImageDataSource(final int width, final int height, final int frameRate,
        final List<BufferedImage> images) {

        super();

        this.streams = new BufferedImageSourceStream[1];
        this.streams[0] = new BufferedImageSourceStream(width, height, frameRate, images);
    }

    /**
     * Sets the media locator (does nothing).
     *
     * @param  source  the source media locator
     */
    @Override public void setLocator(final MediaLocator source) {

        /* Empty */
    }

    /**
     * Gets the media locator (returns <code>null</code>).
     *
     * @return  the media locator
     */
    @Override public MediaLocator getLocator() {

        return null;
    }

    /**
     * Content type is of RAW since we are sending buffers of video frames without a container
     * format.
     *
     * @return  the content type (RAW)
     */
    @Override public String getContentType() {

        return ContentDescriptor.RAW;
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void connect() {

        /* Empty */
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void disconnect() {

        /* Empty */
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void start() {

        /* Empty */
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void stop() {

        /* Empty */
    }
```

```
    /**
     * Returns the <code>PullBufferStream</code>.
     *
     * @return  the streams
     */
    @Override public PullBufferStream[] getStreams() {

        return this.streams.clone();
    }

    /**
     * Returns the duration of the movie. We could have derived the duration from the number of
     * frames and frame rate. But for the purpose of this program, it's not necessary.
     *
     * @return  <code>DURATION_UNKNOWN</code>
     */
    @Override public Time getDuration() {

        return DURATION_UNKNOWN;
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @return  a zero-length object list
     */
    @Override public Object[] getControls() {

        return new Object[0];
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @param   type  the type of control to get
     * @return  <code>null</code>
     */
    @Override public Object getControl(final String type) {

        return null;
    }
}

package com.srbenoit.media.movie;

import java.awt.Dimension;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.List;
import javax.imageio.ImageIO;
import javax.media.Buffer;
import javax.media.Format;
import javax.media.format.VideoFormat;
import javax.media.protocol.ContentDescriptor;
import javax.media.protocol.PullBufferStream;

/**
 * The source stream to go along with <code>BufferedImageDataSource</code>.
 */
public class BufferedImageSourceStream implements PullBufferStream {

    /** the list of image filenames */
    private final transient List<BufferedImage> images;

    /** the target video format */
    private final transient VideoFormat format;

    /** the index of the next image to be read */
    private transient int nextImage = 0;

    /** true if we are finished */
    private transient boolean ended = false;

    /**
     * Constructs a new <code>BufferedImageSourceStream</code>.
     *
     * @param  width       the target frame width
     * @param  height      the target frame height
     * @param  frameRate   the target frame rate
     * @param  imageList   the list of image filenames
     */
    public BufferedImageSourceStream(final int width, final int height, final int frameRate,
        final List<BufferedImage> imageList) {

        this.images = imageList;
```

```
        this.format = new VideoFormat(VideoFormat.JPEG, new Dimension(width, height),
                Format.NOT_SPECIFIED, Format.byteArray, frameRate);
    }

    /**
     * We should never need to block assuming data are read from files.
     *
     * @return  <code>false</code>
     */
    public boolean willReadBlock() {

        return false;
    }

    /**
     * This is called from the Processor to read a frame worth of video data.
     *
     * @param   buf  the buffer in which to store the data
     * @throws  IOException  if there is an error reading the data
     */
    public void read(final Buffer buf) throws IOException {

        BufferedImage imageFile;
        ByteArrayOutputStream baos;
        byte[] frameData;

        // Check if we've finished all the frames.
        if (this.nextImage >= this.images.size()) {

            // We are done. Set EndOfMedia.
            buf.setEOM(true);
            buf.setOffset(0);
            buf.setLength(0);
            this.ended = true;
        } else {

            imageFile = this.images.get(this.nextImage);
            this.nextImage++;

            if (imageFile == null) {

                // We are done. Set EndOfMedia.
                buf.setEOM(true);
                buf.setOffset(0);
                buf.setLength(0);
                this.ended = true;

                return;
            }

            baos = new ByteArrayOutputStream();
            ImageIO.write(imageFile, "JPEG", baos);
            baos.close();

            frameData = baos.toByteArray();

            buf.setData(frameData);
            buf.setOffset(0);
            buf.setLength(frameData.length);
            buf.setFormat(this.format);
            buf.setFlags(buf.getFlags() | Buffer.FLAG_KEY_FRAME);
        }
    }

    /**
     * Returns the format of each video frame. That will be JPEG.
     *
     * @return  the format
     */
    public Format getFormat() {

        return this.format;
    }

    /**
     * Gets the content descriptor.
     *
     * @return  the content type (RAW)
     */
    public ContentDescriptor getContentDescriptor() {

        return new ContentDescriptor(ContentDescriptor.RAW);
    }

    /**
     * Gets the content length.
```

```
     *
     * @return  0
     */
    public long getContentLength() {

        return 0;
    }

    /**
     * Tests whether the stream has ended.
     *
     * @return  true if stream has ended; false otherwise
     */
    public boolean endOfStream() {

        return this.ended;
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @return  a zero-length object list
     */
    public Object[] getControls() {

        return new Object[0];
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @param   type  the type of control to get
     * @return  <code>null</code>
     */
    public Object getControl(final String type) {

        return null;
    }
}

package com.srbenoit.media.movie;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import javax.media.MediaLocator;
import com.srbenoit.log.LoggedObject;

/**
 * Class to convert a series of frame files into a movie.
 */
public final class BuildMovie extends LoggedObject {

    /**
     * Private constructor to prevent instantiation.
     */
    private BuildMovie() {

        super();
    }

    /**
     * Main method to run the conversion.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        String fname;
        File file;
        MakeMovie maker;
        List<BufferedImage> images;
        BufferedImage img;
        MediaLocator loc;

        images = new ArrayList<BufferedImage>(1000);

        for (int i = 1; i < 999999; i++) {

            if ((i % 100) == 1) {
                LOG.finest(".");
            }
```

```
            fname = "/imp/frames/frame_" + Integer.toString(i / 1000)
                + Integer.toString((i / 100) % 10) + Integer.toString((i / 10) % 10)
                + Integer.toString(i % 10) + ".png";
            file = new File(fname); // NOPMD SRB

            if (file.exists()) {

                try {
                    img = ImageIO.read(file);
                } catch (IOException ex) {
                    LOG.log(Level.WARNING, "Exception reading file " + file.getAbsolutePath(), ex);

                    break;
                }

                if (img == null) {
                    break;
                }

                images.add(img);
            } else {
                break;
            }
        }

        if (images.isEmpty()) {
            LOG.info("No files to process");
        } else {
            LOG.log(Level.INFO, "Processing {0}", images.size());
            maker = new MakeMovie();
            loc = MakeMovie.createMediaLocator("file:/imp/cell_with_fixed_attractant.mov");

            try {
                maker.doItBuffered(800, 600, 24, images, loc);
            } catch (MovieMakingException e) {
                LOG.log(Level.WARNING, "Exception building movie", e);
            }
        }
    }
}

package com.srbenoit.media.movie;

import java.util.List;
import javax.media.MediaLocator;
import javax.media.Time;
import javax.media.protocol.ContentDescriptor;
import javax.media.protocol.PullBufferDataSource;
import javax.media.protocol.PullBufferStream;

/**
 * A DataSource to read from a list of JPEG image files and turn that into a stream of JMF buffers.
 * The DataSource is not seekable or positionable.
 */
public class ImageDataSource extends PullBufferDataSource {

    /** the list of input streams */
    private final transient ImageSourceStream[] streams;

    /**
     * Constructs a new <code>ImageDataSource</code>.
     *
     * @param  width       the target frame width
     * @param  height      the target frame height
     * @param  frameRate   the target frame rate
     * @param  images      the list of image names
     */
    ImageDataSource(final int width, final int height, final int frameRate,
        final List<String> images) {

        super();

        this.streams = new ImageSourceStream[1];
        this.streams[0] = new ImageSourceStream(width, height, frameRate, images);
    }

    /**
     * Sets the media locator (does nothing).
     *
     * @param  source  the source media locator
     */
    @Override public void setLocator(final MediaLocator source) {

        /* Empty */
    }

    /**
```

```
     * Gets the media locator (does nothing).
     *
     * @return  the media locator
     */
    @Override public MediaLocator getLocator() {

        return null;
    }

    /**
     * Content type is of RAW since we are sending buffers of video frames without a container
     * format.
     *
     * @return  the content type (RAW)
     */
    @Override public String getContentType() {

        return ContentDescriptor.RAW;
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void connect() {

        /* Empty */
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void disconnect() {

        /* Empty */
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void start() {

        /* Empty */
    }

    /**
     * Dummy implementation of the superclass method.
     */
    @Override public void stop() {

        /* Empty */
    }

    /**
     * Returns the <code>PullBufferStream</code>.
     *
     * @return  the streams
     */
    @Override public PullBufferStream[] getStreams() {

        return this.streams.clone();
    }

    /**
     * Returns the duration of the movie. We could have derived the duration from the number of
     * frames and frame rate. But for the purpose of this program, it's not necessary.
     *
     * @return  <code>DURATION_UNKNOWN</code>
     */
    @Override public Time getDuration() {

        return DURATION_UNKNOWN;
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @return  a zero-length object list
     */
    @Override public Object[] getControls() {

        return new Object[0];
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @param   type  the type of control to get
```

```
     * @return  <code>null</code>
     */
    @Override public Object getControl(final String type) {

        return null;
    }
}

package com.srbenoit.media.movie;

import java.awt.Dimension;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.List;
import javax.media.Buffer;
import javax.media.Format;
import javax.media.format.VideoFormat;
import javax.media.protocol.ContentDescriptor;
import javax.media.protocol.PullBufferStream;

/**
 * The source stream to go along with <code>ImageDataSource</code>.
 */
public class ImageSourceStream implements PullBufferStream {

    /** the list of image filenames */
    private final transient List<String> images;

    /** the target video format */
    private final transient VideoFormat format;

    /** the index of the next image to be read */
    private transient int nextImage = 0;

    /** true if we are finished */
    private transient boolean ended = false;

    /**
     * Constructs a new <code>ImageSourceStream</code>.
     *
     * @param  width       the target frame width
     * @param  height      the target frame height
     * @param  frameRate   the target frame rate
     * @param  imageList   the list of image filenames
     */
    public ImageSourceStream(final int width, final int height, final int frameRate,
        final List<String> imageList) {

        this.images = imageList;

        this.format = new VideoFormat(VideoFormat.JPEG, new Dimension(width, height),
                Format.NOT_SPECIFIED, Format.byteArray, frameRate);
    }

    /**
     * We should never need to block assuming data are read from files.
     *
     * @return  <code>false</code>
     */
    public boolean willReadBlock() {

        return false;
    }

    /**
     * Called from the Processor to read a frame worth of video data.
     *
     * @param   buf  the buffer in which to store the data
     * @throws  IOException  if there is an error reading the data
     */
    public void read(final Buffer buf) throws IOException {

        String imageFile;
        RandomAccessFile raFile;
        byte[] data;

        // Check if we've finished all the frames.
        if (this.nextImage >= this.images.size()) {

            // We are done. Set EndOfMedia.
            buf.setEOM(true);
            buf.setOffset(0);
            buf.setLength(0);
            this.ended = true;

            return;
        }
```

```
        imageFile = this.images.get(this.nextImage);
        this.nextImage++;

        // Open a random access file for the next image.
        raFile = new RandomAccessFile(imageFile, "r");

        // Check the input buffer type & size.
        if (buf.getData() instanceof byte[]) {
            data = (byte[]) buf.getData();

            // Check to see the given buffer is big enough for the frame.
            if ((data == null) || (data.length < raFile.length())) {
                data = new byte[(int) raFile.length()];
                buf.setData(data);
            }

            // Read the entire JPEG image from the file.
            raFile.readFully(data, 0, (int) raFile.length());

            buf.setOffset(0);
            buf.setLength((int) raFile.length());
            buf.setFormat(this.format);
            buf.setFlags(buf.getFlags() | Buffer.FLAG_KEY_FRAME);
        }

        // Close the random access file.
        raFile.close();
    }

    /**
     * Returns the format of each video frame. That will be JPEG.
     *
     * @return  the format
     */
    public Format getFormat() {

        return this.format;
    }

    /**
     * Gets the content descriptor.
     *
     * @return  the content type (RAW)
     */
    public ContentDescriptor getContentDescriptor() {

        return new ContentDescriptor(ContentDescriptor.RAW);
    }

    /**
     * Gets the content length.
     *
     * @return  0
     */
    public long getContentLength() {

        return 0;
    }

    /**
     * Tests whether the stream has ended.
     *
     * @return  <code>true</code> if stream has ended; <code>false</code> otherwise.
     */
    public boolean endOfStream() {

        return this.ended;
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @return  a zero-length object list
     */
    public Object[] getControls() {

        return new Object[0];
    }

    /**
     * Dummy implementation of the superclass method.
     *
     * @param   type  the type of control to get
     * @return  <code>null</code>
     */
    public Object getControl(final String type) {
```

```
        return null;
    }
}

package com.srbenoit.media.movie;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.logging.Level;
import javax.media.ConfigureCompleteEvent;
import javax.media.Controller;
import javax.media.ControllerEvent;
import javax.media.ControllerListener;
import javax.media.DataSink;
import javax.media.EndOfMediaEvent;
import javax.media.Format;
import javax.media.Manager;
import javax.media.MediaLocator;
import javax.media.PrefetchCompleteEvent;
import javax.media.Processor;
import javax.media.RealizeCompleteEvent;
import javax.media.ResourceUnavailableEvent;
import javax.media.control.TrackControl;
import javax.media.datasink.DataSinkErrorEvent;
import javax.media.datasink.DataSinkEvent;
import javax.media.datasink.DataSinkListener;
import javax.media.datasink.EndOfStreamEvent;
import javax.media.protocol.ContentDescriptor;
import javax.media.protocol.DataSource;
import javax.media.protocol.FileTypeDescriptor;
import com.srbenoit.log.LoggedObject;

/**
 * Generates a QuickTime movie from a series of JPEG frames.
 */
public class MakeMovie extends LoggedObject implements ControllerListener, DataSinkListener {

    /** object on which to synchronize waiting for the end of a file */
    private final transient Object waitFileSync = new Object();

    /** object on which to synchronize waiting for a state change */
    private final transient Object waitSync = new Object();

    /** flag indicating file is done */
    private transient boolean fileDone = false;

    /** flag indicating file read succeeded */
    private transient boolean fileSuccess = true;

    /** flag indicating state transition succeeded */
    private transient boolean stateTransOK = true;

    /**
     * Builds the movie.
     *
     * @param    width       the movie width
     * @param    height      the movie height
     * @param    frameRate   the movie frame rate
     * @param    inImages    the list of input frame images
     * @param    outML       the media locator for the produced movie
     * @return   <code>true</code> if successful; <code>false</code> otherwise
     * @throws   MovieMakingException  if there was an error
     */
    public boolean doItBuffered(final int width, final int height, final int frameRate,
        final BufferedImage[] inImages, final MediaLocator outML) throws MovieMakingException {

        List<BufferedImage> list;

        list = new ArrayList<BufferedImage>(inImages.length);
        list.addAll(Arrays.asList(inImages));

        return doItBuffered(width, height, frameRate, list, outML);
    }

    /**
     * Builds the movie.
     *
     * @param    width       the movie width
     * @param    height      the movie height
     * @param    frameRate   the movie frame rate
     * @param    inImages    the list of input frame images
     * @param    outML       the media locator for the produced movie
```

```
     * @return  <code>true</code> if successful; <code>false</code> otherwise
     * @throws  MovieMakingException  if there was an error
     */
    public boolean doItBuffered(final int width, final int height, final int frameRate,
        final List<BufferedImage> inImages, final MediaLocator outML) throws MovieMakingException {

        BufferedImageDataSource ids;
        Processor proc;
        TrackControl[] tcs;
        Format[] fmt;
        DataSink dsink;

        ids = new BufferedImageDataSource(width, height, frameRate, inImages);

        try {
            proc = Manager.createProcessor(ids);
        } catch (Exception e) {
            throw new MovieMakingException("Cannot create a processor from the data source.", e);
        }

        proc.addControllerListener(this);

        // Put the Processor into configured state so we can set
        // some processing options on the processor.
        proc.configure();

        if (!waitForState(proc, Processor.Configured)) {
            throw new MovieMakingException("Failed to configure the processor.");
        }

        // Set the output content descriptor to QuickTime.
        proc.setContentDescriptor(new ContentDescriptor(FileTypeDescriptor.QUICKTIME));

        // Query for the processor for supported formats.
        // Then set it on the processor.
        tcs = proc.getTrackControls();
        fmt = tcs[0].getSupportedFormats();

        if ((fmt == null) || (fmt.length <= 0)) {
            throw new MovieMakingException("The mux does not support the input format: "
                + tcs[0].getFormat());
        }

        tcs[0].setFormat(fmt[0]);

        // We are done with programming the processor. Let's just
        // realize it.
        proc.realize();

        if (!waitForState(proc, Controller.Realized)) {
            throw new MovieMakingException("Failed to realize the processor.");
        }

        // Now, we'll need to create a DataSink.
        dsink = createDataSink(proc, outML);

        if (dsink == null) {
            throw new MovieMakingException(
                "Failed to create a DataSink for the given output MediaLocator: " + outML);
        }

        dsink.addDataSinkListener(this);
        this.fileDone = false;

        // OK, we can now start the actual transcoding.
        try {
            proc.start();
            dsink.start();
        } catch (IOException e) {
            throw new MovieMakingException("IO error during processing", e);
        }

        // Wait for EndOfStream event.
        waitForFileDone();

        // Cleanup.
        try {
            dsink.close();
        } catch (Exception e) {
            LOG.log(Level.WARNING, "Exception closing data sink: {0}", e.getMessage());
        }

        proc.removeControllerListener(this);

        return true;
    }
```

```
/**
 * Builds the movie.
 *
 * @param    width       the movie width
 * @param    height      the movie height
 * @param    frameRate   the movie frame rate
 * @param    inFiles     the list of input files
 * @param    outML       the media locator for the produced movie
 * @return  <code>true</code> if successful; <code>false</code> otherwise
 * @throws  MovieMakingException  if there was an error
 */
public boolean doItFiles(final int width, final int height, final int frameRate,
    final List<String> inFiles, final MediaLocator outML) throws MovieMakingException {

    ImageDataSource ids;
    Processor proc;
    TrackControl[] tcs;
    Format[] fmt;
    DataSink dsink;

    ids = new ImageDataSource(width, height, frameRate, inFiles);

    try {
        proc = Manager.createProcessor(ids);
    } catch (Exception e) {
        throw new MovieMakingException("Cannot create a processor from the data source.", e);
    }

    proc.addControllerListener(this);

    // Put the Processor into configured state so we can set
    // some processing options on the processor.
    proc.configure();

    if (!waitForState(proc, Processor.Configured)) {
        throw new MovieMakingException("Failed to configure the processor.");
    }

    // Set the output content descriptor to QuickTime.
    proc.setContentDescriptor(new ContentDescriptor(FileTypeDescriptor.QUICKTIME));

    // Query for the processor for supported formats.
    // Then set it on the processor.
    tcs = proc.getTrackControls();
    fmt = tcs[0].getSupportedFormats();

    if ((fmt == null) || (fmt.length <= 0)) {
        throw new MovieMakingException("The mux does not support the input format: "
            + tcs[0].getFormat());
    }

    tcs[0].setFormat(fmt[0]);

    // We are done with programming the processor. Let's just
    // realize it.
    proc.realize();

    if (!waitForState(proc, Controller.Realized)) {
        throw new MovieMakingException("Failed to realize the processor.");
    }

    // Now, we'll need to create a DataSink.
    dsink = createDataSink(proc, outML);

    if (dsink == null) {
        throw new MovieMakingException(
            "Failed to create a DataSink for the given output MediaLocator: " + outML);
    }

    dsink.addDataSinkListener(this);
    this.fileDone = false;

    // OK, we can now start the actual transcoding.
    try {
        proc.start();
        dsink.start();
    } catch (IOException e) {
        throw new MovieMakingException("IO error during processing", e);
    }

    // Wait for EndOfStream event.
    waitForFileDone();

    // Cleanup.
    try {
        dsink.close();
    } catch (Exception e) {
```

```
            LOG.log(Level.WARNING, "Exception closing data sink: {0}", e.getMessage());
        }

        proc.removeControllerListener(this);

        return true;
    }

    /**
     * Creates the DataSink.
     *
     * @param   proc   the processor
     * @param   outML  the locator for the output data
     * @return  the data sink
     */
    public DataSink createDataSink(final Processor proc, final MediaLocator outML) {

        DataSource src;
        DataSink temp;
        DataSink sink;

        src = proc.getDataOutput();

        if (src == null) {
            LOG.warning("The processor does not have an output DataSource");
            sink = null;
        } else {

            try {
                temp = Manager.createDataSink(src, outML);
                temp.open();
                sink = temp;
            } catch (Exception e) {
                LOG.log(Level.WARNING, "Cannot create the DataSink", e);
                sink = null;
            }
        }

        return sink;
    }

    /**
     * Blocks until the processor has transitioned to the given state. Returns false if the
     * transition failed.
     *
     * @param   proc   the processor
     * @param   state  the state
     * @return  <code>true</code> if successful; <code>false</code> otherwise
     */
    public boolean waitForState(final Processor proc, final int state) {

        synchronized (this.waitSync) {

            try {

                while ((proc.getState() < state) && this.stateTransOK) {
                    this.waitSync.wait();
                }
            } catch (Exception e) {
                LOG.log(Level.WARNING, "Exception while waiting for state: {0}", e.getMessage());
            }
        }

        return this.stateTransOK;
    }

    /**
     * Controller listener.
     *
     * @param  evt  the controller event
     */
    public void controllerUpdate(final ControllerEvent evt) {

        if ((evt instanceof ConfigureCompleteEvent) || (evt instanceof RealizeCompleteEvent)
                || (evt instanceof PrefetchCompleteEvent)) {

            synchronized (this.waitSync) {
                this.stateTransOK = true;
                this.waitSync.notifyAll();
            }
        } else if (evt instanceof ResourceUnavailableEvent) {

            synchronized (this.waitSync) {
                this.stateTransOK = false;
                this.waitSync.notifyAll();
            }
        } else if (evt instanceof EndOfMediaEvent) {
```

```
            evt.getSourceController().stop();
            evt.getSourceController().close();
        }
    }

    /**
     * Blocks until file writing is done.
     *
     * @return  <code>true</code> if success; <code>false</code> otherwise
     */
    public boolean waitForFileDone() {

        synchronized (this.waitFileSync) {

            try {

                while (!this.fileDone) {
                    this.waitFileSync.wait();
                }
            } catch (Exception e) {
                LOG.log(Level.WARNING, "Exception while waiting for done: {0}", e.getMessage());
            }
        }

        return this.fileSuccess;
    }

    /**
     * Event handler for the file writer.
     *
     * @param  evt  the event
     */
    public void dataSinkUpdate(final DataSinkEvent evt) {

        if (evt instanceof EndOfStreamEvent) {

            synchronized (this.waitFileSync) {
                this.fileDone = true;
                this.waitFileSync.notifyAll();
            }
        } else if (evt instanceof DataSinkErrorEvent) {

            synchronized (this.waitFileSync) {
                this.fileDone = true;
                this.fileSuccess = false;
                this.waitFileSync.notifyAll();
            }
        }
    }

    /**
     * Creates a media locator from the given URL.
     *
     * @param   url  the URL (if the URL contains ':', it is assumed to be a complete URL; if it
     *               begins with a file path separator, it is assumed to be an absolute path;
     *               otherwise, it is a relative path beneath the user's home directory)
     * @return  the media locator
     */
    public static MediaLocator createMediaLocator(final String url) {

        String file;
        MediaLocator loc;

        if ((url.startsWith("C:\\")) || (url.startsWith("C:/")) || (url.startsWith("D:\\"))
                || (url.startsWith("D:/"))) {
            loc = new MediaLocator("file://" + url);
        } else if (url.indexOf(':') > 0) {
            loc = new MediaLocator(url);
        } else if (url.startsWith(File.separator)) {
            loc = new MediaLocator("file:" + url);
        } else {
            file = "file:" + System.getProperty("user.dir") + File.separator + url;
            loc = new MediaLocator(file);
        }

        return loc;
    }
}

package com.srbenoit.media.movie;

/**
 * An exception during creation of a movie.
 */
public class MovieMakingException extends Exception {

    /** version number for serialization */
```

```
    private static final long serialVersionUID = 6884259660418708801L;

    /**
     * Construct a new <code>MovieMakingException</code>.
     *
     * @param  message  the message
     */
    public MovieMakingException(final String message) {

        super(message);
    }

    /**
     * Construct a new <code>MovieMakingException</code>.
     *
     * @param  message  the message
     * @param  cause    the exception that caused the failure
     */
    public MovieMakingException(final String message, final Throwable cause) {

        super(message, cause);
    }
}
```

E.3 3-D Visualization

E.3.1 Rendering Pipeline (com.srbenoit.render)

This package provides tools to visualize and render scenes in three dimensions.

```
package com.srbenoit.render;

import com.srbenoit.geom.Point3;
import com.srbenoit.geom.Point3Int;
import com.srbenoit.geom.Transform3;
import com.srbenoit.geom.Vector3Int;

/**
 * A representation of a camera, which manages transformations from world space into view space,
 * clips lines to a view frustum (defined by a view angle, aspect ratio, and the near and far
 * clipping planes), projects into normalized device coordinates, then finally maps into screen
 * space. This class does not do any rendering.
 *
 * <p>The view coordinates are in a frame in which the camera is at the origin looking down the -Z
 * axis (so the origin in world space is mapped to the point (0, 0, -dist). This maps +X
 * coordinates to the right-pointing direction as seen from the camera, and +Y coordinates in the
 * upward-pointing direction (at least in the camera's default position, which is looking at the
 * origin from the +X axis). The key point is the the clip planes have negative Z coordinates, but
 * the clip distances are stored as positive values.
 */
public class Camera {

    /** a view angle (angle between left and right edges of screen) */
    public static final double VIEW_ANGLE = 50 * Math.PI / 180;

    /** precomputed value used in perspective projection */
    public static final double DIST = 1 / Math.tan(VIEW_ANGLE / 2);

    /** commonly used value */
    private static final double TWO_PI = 2 * Math.PI;

    /** commonly used value */
    private static final double HALF_PI = 0.5f * Math.PI;

    /** an object used to synchronize access to offscreen image */
    protected final transient Object synch;

    /** look-at point */
    private final transient Point3 lookAt;

    /** the transformation matrix */
    protected final transient Transform3 xform;

    /** the aspect ratio of the window */
    private transient double aspect = 1;

    /** near clipping plane distance */
```

```
private transient double nearClip;

/** far clipping plane distance */
private transient double farClip;

/** polar angle */
private transient double polar;

/** azimuthal angle */
private transient double azimuthal;

/** distance from camera to the look-at point */
private transient double distance;

/**
 * Constructs a new <code>Camera</code>. The camera will have its look-at point set to the
 * origin, and have its polar and azimuthal angles set to PI/2 and 0, respectively (looking at
 * the origin from the +x axis).
 *
 * <p>The near clipping and far clipping planes will be set to one tenth and ten times the
 * initial viewing distance.
 *
 * @param  initDistance  the initial distance from the look-at point of the camera
 */
public Camera(final double initDistance) {

    this.synch = new Object();

    this.lookAt = new Point3(); // Default is origin
    this.polar = HALF_PI;
    this.azimuthal = 0;
    this.distance = initDistance;
    this.nearClip = initDistance / 10;
    this.farClip = initDistance * 10;
    this.xform = new Transform3();
}

/**
 * Sets the point that the camera looks at and recomputes the transformation matrix.
 *
 * @param  lookAtX  the X coordinate of the look-at point
 * @param  lookAtY  the Y coordinate of the look-at point
 * @param  lookAtZ  the Z coordinate of the look-at point
 */
public void setLookAt(final double lookAtX, final double lookAtY, final double lookAtZ) {

    synchronized (this.synch) {
        this.lookAt.setPos(lookAtX, lookAtY, lookAtZ);
        computeMatrix();
    }
}

/**
 * Gets the point that the camera looks at.
 *
 * @return  the look-at point
 */
public Point3 getLookAt() {

    synchronized (this.synch) {
        return new Point3(this.lookAt);
    }
}

/**
 * Sets the aspect ratio of the view port and recomputes the transformation matrix.
 *
 * @param  aspectRatio  the aspect ratio (width / height)
 */
public void setAspect(final double aspectRatio) {

    synchronized (this.synch) {
        this.aspect = aspectRatio;
        computeMatrix();
    }
}

/**
 * Gets the aspect ratio of the view port.
 *
 * @return  the aspect ratio (width / height)
 */
public double getAspect() {

    synchronized (this.synch) {
        return this.aspect;
    }
```

```
}

/**
 * Sets the distance from the camera to the near clip plane. This value must be greater than
 * zero to avoid singularities in the projection to screen space.
 *
 * @param  nearClipDist  the new near clip distance
 */
public void setNearClip(final double nearClipDist) {

    synchronized (this.synch) {
        this.nearClip = nearClipDist;
    }
}

/**
 * Gets the distance from the camera to the near clip plane.
 *
 * @return  the near clip distance
 */
public double getNearClip() {

    synchronized (this.synch) {
        return this.nearClip;
    }
}

/**
 * Sets the distance from the camera to the far clip plane. This value must be greater than
 * zero to avoid singularities in the projection to screen space.
 *
 * @param  farClipDist  the new far clip distance
 */
public void setFarClip(final double farClipDist) {

    synchronized (this.synch) {
        this.farClip = farClipDist;
    }
}

/**
 * Gets the distance from the camera to the near clip plane.
 *
 * @return  the far clip distance
 */
public double getFarClip() {

    synchronized (this.synch) {
        return this.farClip;
    }
}

/**
 * Sets the polar angle and recomputes the transformation matrix.
 *
 * @param  angle          the new polar angle
 * @param  updateMatrix   <code>true</code> to recompute the matrix, <code>false</code> to leave
 *                        the matrix in place (used where there are multiple adjustments -
 *                        update the matrix only after the last adjustment)
 */
public void setPolarAngle(final double angle, final boolean updateMatrix) {

    synchronized (this.synch) {
        this.polar = angle;

        if (updateMatrix) {
            computeMatrix();
        }
    }
}

/**
 * Adjusts the polar angle by some amount. The polar angle is range limited - attempts to
 * adjust it above Pi or below negative Pi will result in the value stopping at Pi or negative
 * Pi, respectively (it does not "wrap around").
 *
 * @param  delta          the amount by which to adjust the angle (may be positive or negative)
 * @param  updateMatrix   <code>true</code> to recompute the matrix, <code>false</code> to leave
 *                        the matrix in place (used where there are multiple adjustments -
 *                        update the matrix only after the last adjustment)
 */
public void adjustPolarAngle(final double delta, final boolean updateMatrix) {

    synchronized (this.synch) {
        this.polar += delta;

        if (this.polar < 0) {
```

```
            this.polar = 0;
        }

        if (this.polar > Math.PI) {
            this.polar = Math.PI;
        }

        if (updateMatrix) {
            computeMatrix();
        }
    }
}

/**
 * Sets the azimuthal angle and recomputes the transformation matrix.
 *
 * @param  angle          the new azimuthal angle
 * @param  updateMatrix   <code>true</code> to recompute the matrix, <code>false</code> to leave
 *                        the matrix in place (used where there are multiple adjustments -
 *                        update the matrix only after the last adjustment)
 */
public void setAzimuthalAngle(final double angle, final boolean updateMatrix) {

    synchronized (this.synch) {
        this.azimuthal = angle;

        if (updateMatrix) {
            computeMatrix();
        }
    }
}

/**
 * Adjusts the azimuthal angle by some amount. The azimuthal angle "wraps around" if you
 * increase or decrease it beyond its limits.
 *
 * @param  delta          the amount by which to adjust the angle (may be positive or negative)
 * @param  updateMatrix   <code>true</code> to recompute the matrix, <code>false</code> to leave
 *                        the matrix in place (used where there are multiple adjustments -
 *                        update the matrix only after the last adjustment)
 */
public void adjustAzimuthalAngle(final double delta, final boolean updateMatrix) {

    synchronized (this.synch) {
        this.azimuthal += delta;

        while (this.azimuthal < 0) {
            this.azimuthal += TWO_PI;
        }

        while (this.azimuthal >= TWO_PI) {
            this.azimuthal -= TWO_PI;
        }

        if (updateMatrix) {
            computeMatrix();
        }
    }
}

/**
 * Gets the camera distance.
 *
 * @return  the distance
 */
public double getDistance() {

    synchronized (this.synch) {
        return this.distance;
    }
}

/**
 * Sets the camera distance and recomputes the transformation matrix.
 *
 * @param  dist           the new distance
 * @param  updateMatrix   <code>true</code> to recompute the matrix, <code>false</code> to leave
 *                        the matrix in place (used where there are multiple adjustments -
 *                        update the matrix only after the last adjustment)
 */
public void setDistance(final double dist, final boolean updateMatrix) {

    synchronized (this.synch) {

        if (dist < this.nearClip) {
            this.distance = this.nearClip;
        } else {
```

```
            this.distance = dist;
        }

        if (updateMatrix) {

            // Updating the matrix is easy so just do it.
            this.xform.set(2, 3, -this.distance);
        }
    }
}

/**
 * Adjusts the distance by some amount. The distance can grow without limit but must always be
 * at least the near clip distance. Attempts to adjust distance downward beyond the near clip
 * distance will result in distance stopping at that distance.
 *
 * @param  delta          the amount by which to adjust the distance (may be positive or
 *                        negative)
 * @param  updateMatrix   <code>true</code> to recompute the matrix, <code>false</code> to leave
 *                        the matrix in place (used where there are multiple adjustments -
 *                        update the matrix only after the last adjustment)
 */
public void adjustDistance(final double delta, final boolean updateMatrix) {

    synchronized (this.synch) {
        this.distance += delta;

        if (this.distance < this.nearClip) {
            this.distance = this.nearClip;
        }

        if (updateMatrix) {

            // Updating the matrix is easy so just do it.
            this.xform.set(2, 3, -this.distance);
        }
    }
}

/**
 * Applies the transformation to a tuple to convert world coordinates into view coordinates.
 *
 * @param  source  the tuple to transform
 * @param  dest    the tuple into which to place transformed coordinates
 */
public void transformPoint(final Point3Int source, final Point3Int dest) {

    synchronized (this.synch) {
        this.xform.transformPoint(source, dest);
    }
}

/**
 * Applies the transformation to a tuple to convert world coordinates into view coordinates.
 *
 * @param  source  the tuple to transform
 * @param  dest    the tuple into which to place transformed coordinates
 */
public void transformVec(final Vector3Int source, final Vector3Int dest) {

    synchronized (this.synch) {
        this.xform.transformVec(source, dest);
    }
}

/**
 * Perform the perspective transformation to take points from the view frame into normalized
 * device coordinates.
 *
 * <p>The view frame has its origin at the eye point, and the camera is looking down the Z axis
 * in the negative direction. X increases to the right, Y increases upward.
 *
 * <p>In normalized device coordinates, points within the viewing angle have coordinates with
 * -1 < x < 1 and -1 < y < 1. The Z coordinate stores a scaling factor that can be applied to
 * the size of objects (the thickness of a line, for instance) due to its distance from the
 * viewer.
 *
 * @param  point  the point to map
 */
public void toNormalizedDeviceCoordinates(final Point3Int point) {

    synchronized (this.synch) {
        point.setPos(-DIST * point.getPosX() / (this.aspect * point.getPosZ()),
            -DIST * point.getPosY() / point.getPosZ(), -DIST / point.getPosZ());
    }
}
```

```
/**
 * Scales the normalized device coordinates to the screen and centers them in the window.
 *
 * @param  width    the screen width
 * @param  height   the screen height
 * @param  vec      the vector to map
 */
public void vecToScreen(final int width, final int height, final Vector3Int vec) {

    synchronized (this.synch) {
        vec.setVecX((width + (vec.getVecX() * width)) * 0.5);
        vec.setVecY((height - (vec.getVecY() * height)) * 0.5);
    }
}

/**
 * Scales the normalized device coordinates to the screen and centers them in the window.
 *
 * @param  width    the screen width
 * @param  height   the screen height
 * @param  point    the point to map
 */
public void pointToScreen(final int width, final int height, final Point3Int point) {

    synchronized (this.synch) {
        point.setPosX((width + (point.getPosX() * width)) * 0.5);
        point.setPosY((height - (point.getPosY() * height)) * 0.5);
    }
}

/**
 * Performs the complete transformation from world to view coordinates, then to normalized
 * device coordinates, then to integer screen coordinates.
 *
 * @param  source   the tuple to transform
 * @param  width    the screen width
 * @param  height   the screen height
 * @param  point    the tuple into which to place transformed coordinates
 */
public void transformToScreen(final Point3Int source, final int width, final int height,
    final Point3Int point) {

    synchronized (this.synch) {
        this.xform.transformPoint(source, point);
        point.setPos(-DIST * point.getPosX() / (this.aspect * point.getPosZ()),
            -DIST * point.getPosY() / point.getPosZ(), -DIST / point.getPosZ());
        point.setPosX((int) (width + (point.getPosX() * width)) >> 1);
        point.setPosY((int) (height - (point.getPosY() * height)) >> 1);
    }
}

/**
 * Clips a line to the view frustum, testing whether the line falls completely outside the
 * frustum and can be culled. Clipping is done in the view frame (before converting to
 * normalized device coordinates), since it is possible to divide by zero when converting an
 * unclipped line to normalized device coordinates.
 *
 * @param   end1  on entry, the first end of the unclipped line; on exit, the first end of the
 *                clipped line
 * @param   end2  on entry, the second end of the unclipped line; on exit, the second end of
 *                the clipped line
 * @return  <code>true</code> if any portion of the line was within the view frustum; <code>
 *          false</code> if the line can be culled
 */
public boolean clipLine(final Point3 end1, final Point3 end2) {

    double end1Z;
    double end2Z;
    double lambda;
    boolean isVisible;

    end1Z = end1.getPosZ();
    end2Z = end2.getPosZ();

    if ((end1Z <= this.nearClip) || (end2Z <= this.nearClip)) {

        if (end1Z > this.nearClip) {

            // New start is intersection of line with near clip plane
            lambda = (end1Z - this.nearClip) / (end1Z - end2Z);
            end1.setPos(end1.getPosX() - (lambda * (end1.getPosX() - end2.getPosX())),
                end1.getPosY() - (lambda * (end1.getPosY() - end2.getPosY())), this.nearClip);
        } else if (end2Z > this.nearClip) {

            // New end is intersection of line with near clip plane
            lambda = (end2Z - this.nearClip) / (end2Z - end1Z);
            end2.setPos(end2.getPosX() - (lambda * (end2.getPosX() - end1.getPosX())),
```

```
                    end2.getPosY() - (lambda * (end2.getPosY() - end1.getPosY())), this.nearClip);
            }

            isVisible = true;
        } else {
            isVisible = false;
        }

        return isVisible;
    }

    /**
     * Computes the camera world-to-view transformation.
     */
    private void computeMatrix() {

        double sinPolar;
        double cosPolar;
        double sinAzimuth;
        double cosAzimuth;

        synchronized (this.synch) {
            sinPolar = Math.sin(this.polar);
            cosPolar = Math.cos(this.polar);
            sinAzimuth = Math.sin(this.azimuthal);
            cosAzimuth = Math.cos(this.azimuthal);

            this.xform.set(0, 0, -sinAzimuth);
            this.xform.set(1, 0, cosAzimuth);
            this.xform.set(2, 0, 0);

            this.xform.set(0, 1, -cosPolar * cosAzimuth);
            this.xform.set(1, 1, -cosPolar * sinAzimuth);
            this.xform.set(2, 1, sinPolar);

            this.xform.set(0, 2, sinPolar * cosAzimuth);
            this.xform.set(1, 2, sinPolar * sinAzimuth);
            this.xform.set(2, 2, cosPolar);

            this.xform.set(0, 3, (this.distance * sinPolar * cosAzimuth) + this.lookAt.getPosX());
            this.xform.set(1, 3, (this.distance * sinPolar * sinAzimuth) + this.lookAt.getPosY());
            this.xform.set(2, 3, (this.distance * cosPolar) + this.lookAt.getPosZ());

            this.xform.invert();
        }
    }
}

package com.srbenoit.render;

import java.awt.Color;
import com.srbenoit.geom.Point3;

/**
 * A light, which is a point source, non-directional light. Light level is based on the angle
 * between the normal vector and the vector to the light source. No specular at this point.
 */
public class Light extends Point3 {

    /** the light color */
    private final Color color;

    /**
     * Constructs a new <code>Light</code>.
     *
     * @param  xCoord       the X coordinate of the light
     * @param  yCoord       the Y coordinate of the light
     * @param  zCoord       the Z coordinate of the light
     * @param  lightColor   the color of the light
     */
    public Light(final double xCoord, final double yCoord, final double zCoord,
        final Color lightColor) {

        super(xCoord, yCoord, zCoord);

        this.color = lightColor;
    }

    /**
     * Gets the light color.
     *
     * @return  the color
     */
    public Color getColor() {

        return this.color;
    }
```

```
}

package com.srbenoit.render;

import com.srbenoit.log.LoggedObject;

/**
 * A two-dimensional line segment.
 */
public class LineSegment2D extends LoggedObject {

    /** out-code indicating a point has Y coordinate above the maximum */
    private static final int MAXY = 0x01;

    /** out-code indicating a point has Y coordinate below the minimum */
    private static final int MINY = 0x02;

    /** out-code indicating a point has X coordinate above the maximum */
    private static final int MAXX = 0x04;

    /** out-code indicating a point has X coordinate below the minimum */
    private static final int MINX = 0x08;

    /** the X coordinate of the first end */
    private double xCoord1;

    /** the Y coordinate of the first end */
    private double yCoord1;

    /** the X coordinate of the second end */
    private double xCoord2;

    /** the Y coordinate of the second end */
    private double yCoord2;

    /**
     * Constructs a new <code>LineSegment2D</code>.
     *
     * @param  end1x  the X coordinate of the first end
     * @param  end1y  the Y coordinate of the first end
     * @param  end2x  the X coordinate of the second end
     * @param  end2y  the Y coordinate of the second end
     */
    public LineSegment2D(final double end1x, final double end1y, final double end2x,
        final double end2y) {

        this.xCoord1 = end1x;
        this.yCoord1 = end1y;
        this.xCoord2 = end2x;
        this.yCoord2 = end2y;
    }

    /**
     * Gets the X coordinate of the first end of the line.
     *
     * @return  the X coordinate
     */
    public double getX1() {

        return this.xCoord1;
    }

    /**
     * Gets the Y coordinate of the first end of the line.
     *
     * @return  the Y coordinate
     */
    public double getY1() {

        return this.yCoord1;
    }

    /**
     * Gets the X coordinate of the second end of the line.
     *
     * @return  the X coordinate
     */
    public double getX2() {

        return this.xCoord2;
    }

    /**
     * Gets the Y coordinate of the second end of the line.
     *
     * @return  the Y coordinate
     */
```

```
public double getY2() {

    return this.yCoord2;
}

/**
 * An implementation of the Cohen-Sutherland algorithm for line clipping.
 *
 * @param   minX  the lower limit on X
 * @param   maxX  the upper limit on X
 * @param   minY  the lower limit on Y
 * @param   maxY  the upper limit on Y
 * @return  <code>true</code> if part of the line fell within the clip window and the line was
 *          clipped; <code>false</code> if the line falls completely outside the window and can
 *          be culled
 */
public boolean clip(final double minX, final double maxX, final double minY,
    final double maxY) {

    int out1;
    int out2;
    double recSlope;
    double slope;
    boolean visible;

    out1 = outcode1(minX, maxX, minY, maxY);
    out2 = outcode2(minX, maxX, minY, maxY);

    // Clip in a loop until both outcodes are zero
    for (;;) {

        if ((out1 & out2) == 0) {
            visible = true;

            // Not a trivial acceptance
            if ((out1 & MAXY) == MAXY) {
                recSlope = (this.xCoord2 - this.xCoord1) / (this.yCoord2 - this.yCoord1);
                this.xCoord1 += recSlope * (maxY - this.yCoord1);
                this.yCoord1 = maxY;
                out1 = outcode1(minX, maxX, minY, maxY);
            } else if ((out1 & MINY) == MINY) {
                recSlope = (this.xCoord2 - this.xCoord1) / (this.yCoord2 - this.yCoord1);
                this.xCoord1 += recSlope * (minY - this.yCoord1);
                this.yCoord1 = minY;
                out1 = outcode1(minX, maxX, minY, maxY);
            } else if ((out1 & MAXX) == MAXX) {
                slope = (this.yCoord2 - this.yCoord1) / (this.xCoord2 - this.xCoord1);
                this.yCoord1 += slope * (maxX - this.xCoord1);
                this.xCoord1 = maxX;
                out1 = outcode1(minX, maxX, minY, maxY);
            } else if ((out1 & MINX) == MINX) {
                slope = (this.yCoord2 - this.yCoord1) / (this.xCoord2 - this.xCoord1);
                this.yCoord1 += slope * (minX - this.xCoord1);
                this.xCoord1 = minX;
                out1 = outcode1(minX, maxX, minY, maxY);
            } else if ((out2 & MAXY) == MAXY) {
                recSlope = (this.xCoord1 - this.xCoord2) / (this.yCoord1 - this.yCoord2);
                this.xCoord2 += recSlope * (maxY - this.yCoord2);
                this.yCoord2 = maxY;
                out2 = outcode2(minX, maxX, minY, maxY);
            } else if ((out2 & MINY) == MINY) {
                recSlope = (this.xCoord1 - this.xCoord2) / (this.yCoord1 - this.yCoord2);
                this.xCoord2 += recSlope * (minY - this.yCoord2);
                this.yCoord2 = minY;
                out2 = outcode2(minX, maxX, minY, maxY);
            } else if ((out2 & MAXX) == MAXX) {
                slope = (this.yCoord1 - this.yCoord2) / (this.xCoord1 - this.xCoord2);
                this.yCoord2 += slope * (maxX - this.xCoord2);
                this.xCoord2 = maxX;
                out2 = outcode2(minX, maxX, minY, maxY);
            } else if ((out2 & MINX) == MINX) {
                slope = (this.yCoord1 - this.yCoord2) / (this.xCoord1 - this.xCoord2);
                this.yCoord2 += slope * (minX - this.xCoord2);
                this.xCoord2 = minX;
                out2 = outcode2(minX, maxX, minY, maxY);
            } else {
                break;
            }
        } else {
            visible = false;

            break;
        }
    }

    return visible;
}
```

```
/**
 * Compute the out-code for the Cohen-Sutherland line clipping algorithm.
 *
 * @param    minX  the lower limit on X
 * @param    maxX  the upper limit on X
 * @param    minY  the lower limit on Y
 * @param    maxY  the upper limit on Y
 * @return   the out-code
 */
private int outcode1(final double minX, final double maxX, final double minY,
    final double maxY) {

    int code;

    code = 0;

    if (this.xCoord1 > maxY) {
        code |= MAXY;
    }

    if (this.yCoord1 < minY) {
        code |= MINY;
    }

    if (this.xCoord1 > maxX) {
        code |= MAXX;
    }

    if (this.xCoord1 < minX) {
        code |= MINX;
    }

    return code;
}

/**
 * Compute the out-code for the Cohen-Sutherland line clipping algorithm.
 *
 * @param    minX  the lower limit on X
 * @param    maxX  the upper limit on X
 * @param    minY  the lower limit on Y
 * @param    maxY  the upper limit on Y
 * @return   the out-code
 */
private int outcode2(final double minX, final double maxX, final double minY,
    final double maxY) {

    int code;

    code = 0;

    if (this.yCoord2 > maxY) {
        code |= MAXY;
    }

    if (this.yCoord2 < minY) {
        code |= MINY;
    }

    if (this.xCoord2 > maxX) {
        code |= MAXX;
    }

    if (this.xCoord2 < minX) {
        code |= MINX;
    }

    return code;
}

/**
 * Generates the string representation of the line.
 *
 * @return  the string representation
 */
@Override public String toString() {
    StringBuilder str;

    str = new StringBuilder(50);

    str.append("Line Segment: [(");
    str.append(Float.toString((float) this.xCoord1));
    str.append(',');
    str.append(Float.toString((float) this.yCoord1));
    str.append(")-(");
    str.append(Float.toString((float) this.xCoord2));
```

```
        str.append(',');
        str.append(Float.toString((float) this.yCoord2));
        str.append(")]");

        return str.toString();
    }
}

package com.srbenoit.render;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.ComponentEvent;
import java.awt.event.ComponentListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseWheelEvent;
import java.awt.event.MouseWheelListener;
import java.awt.image.BufferedImage;
import java.util.logging.Level;
import com.srbenoit.log.LoggedPanel;

/**
 * A panel that implements a double-buffered offscreen image. Two offscreen images are maintained
 * that are the same size as the panel (this can change if the window is resized). One of these
 * images is passed to a <code>RenderPipeline</code> for rendering, while the other is used to
 * satisfy paint requests. When the rendering is complete, the back buffers are flipped, and a
 * repaint is requested, sending the most recent image to the display.
 *
 * <p>A 3-D program will construct a scene and camera, construct one of these panels, then begin a
 * main loop where geometry is updated and rendering is requested. Rendering requests go through
 * this panel, which calls a render pipeline (maintained internal to this object), passing the
 * passive image. Meanwhile, any requests to draw the window use the active (most recently
 * completed) rendered image.
 */
public class RenderPanel extends LoggedPanel implements ComponentListener, MouseListener,
    MouseMotionListener, MouseWheelListener {

    /** version number for serialization */
    private static final long serialVersionUID = -8751092645837569281L;

    /** object on which to synchronize access to member variables */
    private final Object synch;

    /** the current width of the offscreen images */
    private int width;

    /** the current height of the offscreen images */
    private int height;

    /** distance by which to change camera per wheel move */
    private final double perWheel;

    /** the most recently drawn image */
    private BufferedImage active;

    /** an image to which the next render should be directed */
    private BufferedImage passive;

    /** temporary storage for an incorrectly sized image rendered while size was changing */
    private BufferedImage temp;

    /** the camera that will be used in rendering the scene */
    private Camera camera;

    /** the render pipeline that will to the actual rendering */
    private RenderPipeline pipeline;

    /** the point where a drag began */
    private Point dragStart;

    /**
     * Constructs a new <code>DoubleBufferPanel</code>. This should be called from the AWT event
     * dispatcher thread.
     *
     * @param  initWidth   the initial preferred width
     * @param  initHeight  the initial preferred height
     * @param  theCamera   the camera that will be used in rendering the scene
     */
    public RenderPanel(final int initWidth, final int initHeight, final Camera theCamera) {

        super();

        this.synch = new Object();
        this.perWheel = theCamera.getDistance() / 100;
```

```
        setPreferredSize(new Dimension(initWidth, initHeight));
        setSize(initWidth, initHeight);

        this.active = null;
        this.passive = null;
        this.temp = null;
        this.width = 0;
        this.height = 0;
        updateOffscreenImages();

        this.camera = theCamera;
        this.pipeline = new RenderPipeline();
        this.dragStart = null;

        addComponentListener(this);
        addMouseListener(this);
        addMouseMotionListener(this);
        addMouseWheelListener(this);
    }

    /**
     * Renders a scene using a specified camera onto the passive back buffer image, stores the
     * result where it will be used on the next repaint, then requests a repaint. This must NOT be
     * called from the AWT event thread, so that repaints of the window can occur independent of
     * the rendering process.
     *
     * @param  scene  the scene to render
     */
    public void render(final Scene scene) {

        BufferedImage target;

        synchronized (this.synch) {
            target = this.passive;
        }

        if (target != null) {
            this.pipeline.render(scene, this.camera, target);

            synchronized (this.synch) {

                if (this.passive == target) {

                    // Images have not been reallocated, so just flip buffers
                    this.passive = this.active;
                    this.active = target;
                    this.temp = null;
                } else {

                    // A resize has resulted in new images, so place our results in "temp"
                    this.temp = target;
                }
            }
        }

        repaint();
    }

    /**
     * Tests the current size of the panel against the offscreen image sizes, and if needed,
     * creates new offscreen images. This is done in a synchronized block so a completing render
     * cannot stomp an image we create, and this method will not stomp the output of a render.
     */
    private void updateOffscreenImages() {

        int actWidth;
        int actHeight;

        actWidth = getWidth();
        actHeight = getHeight();

        synchronized (this.synch) {

            LOG.log(Level.FINE, "Building offscreen images {0}x{1}",
                new Object[] { actWidth, actHeight });

            if ((actWidth != this.width) || (actHeight != this.height)) {
                this.active = new BufferedImage(actWidth, actHeight, BufferedImage.TYPE_INT_RGB);
                this.passive = new BufferedImage(actWidth, actHeight, BufferedImage.TYPE_INT_RGB);

                this.width = actWidth;
                this.height = actHeight;
            }
        }
    }
```

```
/**
 * Draws the panel contents using the most up-to-date rendered image available.
 *
 * @param  grx  the <code>graphics</code> to which to draw.
 */
@Override public void paintComponent(final Graphics grx) {

    int actWidth;
    int actHeight;
    double xScale;
    double yScale;
    double scale;
    int scaledWidth;
    int scaledHeight;

    super.paintComponent(grx);

    synchronized (this.synch) {

        if (this.temp == null) {

            if (this.active != null) {
                grx.drawImage(this.active, 0, 0, null);
            }

        } else {
            actWidth = getWidth();
            actHeight = getHeight();
            xScale = actWidth / this.temp.getWidth();
            yScale = actHeight / this.temp.getHeight();
            scale = (xScale > yScale) ? xScale : yScale;
            scaledWidth = (int) ((this.temp.getWidth() * scale) + 0.5); // at least actWidth
            scaledHeight = (int) ((this.temp.getHeight() * scale) + 0.5); // at least actHeight
            grx.drawImage(this.temp, (actWidth - scaledWidth) / 2,
                (actHeight - scaledHeight) / 2, scaledWidth, scaledHeight, null);
        }
    }
}

/**
 * Handles panel resize events, which require allocation of new offscreen images.
 *
 * @param  evt  the component event
 */
public void componentResized(final ComponentEvent evt) {

    updateOffscreenImages();
}

/**
 * Handles panel move events, which we ignore.
 *
 * @param  evt  the component event
 */
public void componentMoved(final ComponentEvent evt) {
    // No action
}

/**
 * Handles panel shown events, which causes a test for actual size against current size of the
 * offscreen images, and may result in allocation of new offscreen images.
 *
 * <p>NOTE: Rendering is done outside the AWT event loop, but writes to the offscreen images.
 * At the end of a render process, if the rendered image is not the size specified in <code>
 * width</code> and <code>height</code>, we store the image in the <code>temp</code> member
 * variable. Otherwise, the end of the render cycle stores the result as the active image, and
 * clears the <code>temp</code> member. When a repaint is requested, if the <code>temp</code>
 * object has an image, that image is drawn, but scaled to fit the window.
 *
 * @param  evt  the component event
 */
public void componentShown(final ComponentEvent evt) {

    updateOffscreenImages();
}

/**
 * Handles panel hidden events, which we ignore.
 *
 * @param  evt  the component event
 */
public void componentHidden(final ComponentEvent evt) {
    // No action
}

/**
 * Handles mouse click events.
```

```
     *
     * @param  evt  the mouse event
     */
    @Override public void mouseClicked(final MouseEvent evt) {
        // No action
    }

    /**
     * Handles mouse press events.
     *
     * @param  evt  the mouse event
     */
    @Override public void mousePressed(final MouseEvent evt) {

        this.dragStart = evt.getPoint();
    }

    /**
     * Handles mouse release events.
     *
     * @param  evt  the mouse event
     */
    @Override public void mouseReleased(final MouseEvent evt) {

        this.dragStart = null;
    }

    /**
     * Handles mouse entered events.
     *
     * @param  evt  the mouse event
     */
    @Override public void mouseEntered(final MouseEvent evt) {
        // No action
    }

    /**
     * Handles mouse exited events.
     *
     * @param  evt  the mouse event
     */
    @Override public void mouseExited(final MouseEvent evt) {
        // No action
    }

    /**
     * Handles mouse drag events.
     *
     * @param  evt  the mouse event
     */
    @Override public void mouseDragged(final MouseEvent evt) {

        Point where;
        int moveX;
        int moveY;

        if (this.dragStart != null) {
            where = evt.getPoint();

            moveX = this.dragStart.x - where.x;
            moveY = this.dragStart.y - where.y;

            if (moveX != 0) {
                this.camera.adjustAzimuthalAngle(moveX / 30.0, true);
            }

            if (moveY != 0) {
                this.camera.adjustPolarAngle(moveY / 30.0, true);
            }

            this.dragStart = where;
        }
    }

    /**
     * Handles mouse move events.
     *
     * @param  evt  the mouse event
     */
    @Override public void mouseMoved(final MouseEvent evt) {
        // No action
    }

    /**
     * Handles mouse wheel events.
     *
     * @param  evt  the mouse wheel event
```

```
     */
    public void mouseWheelMoved(final MouseWheelEvent evt) {

        int units;
        double delta;

        units = evt.getUnitsToScroll();
        delta = this.perWheel * units;
        this.camera.adjustDistance(delta, true);
    }
}

package com.srbenoit.render;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.Stroke;
import java.awt.geom.Line2D;
import java.awt.geom.Path2D;
import java.awt.image.BufferedImage;
import com.srbenoit.geom.BasedVector3;
import com.srbenoit.geom.Vector3;
import com.srbenoit.log.LoggedObject;

/**
 * A render pipeline that can take a <code>Scene</code>, a <code>Camera</code>, and a <code>
 * BufferedImage</code> and generate a rendered view of the scene in the image.
 */
public class RenderPipeline extends LoggedObject {

    /**
     * Constructs a new <code>RenderPipeline</code>.
     */
    public RenderPipeline() {

        super();
    }

    /**
     * Renders the scene.
     *
     * @param  scene    the scene to render
     * @param  camera   the camera to use to render the scene
     * @param  image    the <code>BufferedImage</code> to which to render
     */
    public void render(final Scene scene, final Camera camera, final BufferedImage image) {

        double aspect;

        // Transform world objects into view space (atomic operation, locks the scene) and
        // build a self-consistent scene in view space so world objects can be modified without
        // impacting the render process.  This computes the normal vectors at each view vertex.
        scene.worldToView(camera);

        // Do back-face culling and lighting
        cullBackfaces(scene);
        lightFaces(scene);

        // We now transform into normalized device coordinates, based on the aspect ratio of the
        // target image.  Normal vectors are not transformed here.
        aspect = (double) image.getWidth() / (double) image.getHeight();
        camera.setAspect(aspect);
        scene.viewToNormalized(camera);

        // TODO: clip faces to the view frustum and near/far clip planes

        scene.normalizedToScreen(camera, image.getWidth(), image.getHeight());

        rasterize(scene, image);
    }

    /**
     * Test faces to see whether they are back-facing, and cull those that are.
     *
     * @param  scene  the scene
     */
    private void cullBackfaces(final Scene scene) {

        ViewFaceIterator iter;
        ViewFace face;
        ViewVertex vert;
        double dot;
        Vector3 vec;

        vec = new Vector3();
```

```
        iter = new ViewFaceIterator(scene);

        while (iter.hasNext()) {
            face = iter.next();
            vert = face.getVertex0();
            vec.setVec(vert.getPosX(), vert.getPosY(), vert.getPosZ());
            dot = vec.dot(face);
            face.setCulled(dot > 0);
        }
    }

    /**
     * Computes the light values at each face based on the face normal and the vectors to all light
     * sources.
     *
     * @param  scene  the scene
     */
    private void lightFaces(final Scene scene) {

        ViewFaceIterator iter;
        ViewFace face;
        ViewVertex vert;
        int numLights;
        Light light;
        Color col;
        double red;
        double grn;
        double blu;
        Vector3 vecToLight;
        double dot;

        iter = new ViewFaceIterator(scene);
        numLights = scene.numLights();
        vecToLight = new Vector3();

        if (numLights > 0) {

            while (iter.hasNext()) {
                face = iter.next();

                if (face.isCulled()) {
                    continue;
                }

                vert = face.getVertex0();

                red = 0.3f;
                grn = 0.3f;
                blu = 0.3f;

                for (int i = 0; i < numLights; i++) {
                    light = scene.getViewLight(i);
                    col = light.getColor();
                    vecToLight.vectorBetween(vert, light);
                    vecToLight.normalize();
                    dot = vecToLight.dot(face);
                    red += dot * col.getRed() / 255.0;
                    grn += dot * col.getGreen() / 255.0;
                    blu += dot * col.getBlue() / 255.0;
                }

                if (red < 0.1) {
                    red = 0.1;
                }

                if (grn < 0.1) {
                    grn = 0.1;
                }

                if (blu < 0.1) {
                    blu = 0.1;
                }

                face.setColor(red, grn, blu);
            }
        }
    }

    /**
     * Rasterizes the scene onto the image.
     *
     * @param  scene  the scene to render
     * @param  image  the image onto which to draw the scene
     */
    private void rasterize(final Scene scene, final BufferedImage image) {
```

```
            Graphics2D grx;
            int count;
            BasedVector3 vec;
            ViewFaceIterator iter;
            ViewVertex vert0;
            ViewVertex vert1;
            ViewVertex vert2;
            ViewFace face;
            Path2D path;
            Line2D line;
            Stroke orig;

            // Now we rasterize (for now, just a wireframe)
            grx = (Graphics2D) image.getGraphics();
            grx.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
            grx.setColor(Color.BLACK);
            grx.fillRect(0, 0, image.getWidth(), image.getHeight());

            // Draw all the based vectors
            grx.setColor(Color.RED);
            count = scene.numBasedVectors();

            orig = grx.getStroke();
            grx.setStroke(new BasicStroke(3));

            for (int i = 0; i < count; i++) {
                vec = scene.getBasedVector(i);

                line = new Line2D.Double(vec.getPosX(), vec.getPosY(), vec.getVecX(), vec.getVecY());
                grx.draw(line);
            }

            grx.setStroke(orig);

            iter = new ViewFaceIterator(scene);

            while (iter.hasNext()) {
                face = iter.next();

                if (face.isCulled()) {
                    continue;
                }

                vert0 = face.getVertex0();
                vert1 = face.getVertex1();
                vert2 = face.getVertex2();

                grx.setColor(face.getColor());

                path = new Path2D.Double();
                line = new Line2D.Double(vert0.getPosX(), vert0.getPosY(), vert1.getPosX(),
                        vert1.getPosY());
                path.append(line, true);
                line = new Line2D.Double(vert1.getPosX(), vert1.getPosY(), vert2.getPosX(),
                        vert2.getPosY());
                path.append(line, true);
                line = new Line2D.Double(vert2.getPosX(), vert2.getPosY(), vert0.getPosX(),
                        vert0.getPosY());
                path.append(line, true);
                grx.fill(path);

                grx.setColor(new Color(0, 0, 0, 128));
                grx.draw(path);
            }
        }
    }
```

```
package com.srbenoit.render;

import java.util.BitSet;
import com.srbenoit.geom.BasedVector3;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.sparsearray.SparseArrayException;

/**
 * A scene, which behaves much like a <code>SparseArray</code>, but for which four lists are
 * maintained in parallel. A list of "world coordinate" vertices and "world coordinate" faces
 * represents the live scene content, and parallel lists of vertices and faces are maintained.
 *
 * <p>The rendering pipeline, then, consists of locking the scene and transforming all
 * world-coordinate vertices and faces into view coordinates in the parallel arrays, and unlocking
 * the scene so the live content can continue to evolve. Then the view-coordinate objects, in their
 * self-consistent state, are further processed through culling, lighting, projecting into
 * canonical screen space, clipping, and rasterization.
 */
public class Scene extends LoggedObject {
```

```
/** an object on which to synchronize access to scene structures */
private final Object synch;

/** filled status of each object in the <code>WorldVertex</code> array */
private transient BitSet worldVertexStatus;

/** filled status of each object in the <code>ViewVertex</code> array */
private transient BitSet viewVertexStatus;

/** total number of objects in the <code>WorldVertex</code> array */
private transient int numWorldVertices;

/** index of first available position in <code>WorldVertex</code> array */
private transient int firstOpenWorldVertex;

/** array of <code>WorldVertex</code> objects */
private transient WorldVertex[] worldVertices;

/** array of <code>ViewVertex</code> objects */
private transient ViewVertex[] viewVertices;

/** filled status of each object in the <code>WorldFace</code> array */
private transient BitSet worldFaceStatus;

/** filled status of each object in the <code>ViewFace</code> array */
private transient BitSet viewFaceStatus;

/** total number of objects in the <code>WorldFace</code> array */
private transient int numWorldFaces;

/** index of first available position in <code>WorldFace</code> array */
private transient int firstOpenWorldFace;

/** array of <code>WorldFace</code> objects */
private transient WorldFace[] worldFaces;

/** array of <code>ViewFace</code> objects */
private transient ViewFace[] viewFaces;

/** the lights in world-space */
private transient Light[] worldLights;

/** the lights in view-space */
private transient Light[] viewLights;

/** the based vectors in world-space */
private transient BasedVector3[] worldBasedVectors;

/** the based vectors in view-space */
private transient BasedVector3[] viewBasedVectors;

/**
 * Constructs a new <code>Scene</code>.
 *
 * @param  expectedFaceCount  the estimated number of faces in the scene
 */
public Scene(final int expectedFaceCount) {

    int numFaces;
    int numVert;

    this.synch = new Object();

    // For (Nf) faces, assuming triangular faces only, there are three edges per face, each
    // edge shared by 2 faces, so number of edges (Ne) is (3/2)Nf.  Assuming Euler
    // characteristic 2, so number of vertices (Nv) is 2 + Ne - Nf = 2 + (Nf/2).

    // Round face counts to the next multiple of 128, vertex count will be multiple of 64
    numFaces = ((expectedFaceCount + 127) / 128) * 128;
    numVert = numFaces / 2;

    this.worldVertexStatus = new BitSet(numVert);
    this.viewVertexStatus = new BitSet(numVert);
    this.numWorldVertices = 0;
    this.firstOpenWorldVertex = 0;
    this.worldVertices = new WorldVertex[numVert];
    this.viewVertices = new ViewVertex[numVert];

    this.worldFaceStatus = new BitSet(numFaces);
    this.viewFaceStatus = new BitSet(numFaces);
    this.numWorldFaces = 0;
    this.firstOpenWorldFace = 0;
    this.worldFaces = new WorldFace[numFaces];
    this.viewFaces = new ViewFace[numFaces];

    this.worldLights = new Light[0];
    this.worldBasedVectors = new BasedVector3[0];
```

```
}

/**
 * Adds a vertex to the world-coordinate vertex array, increasing allocated storage if needed.
 *
 * @param   vertex  the vertex to add
 * @return  the index of the newly added vertex in the vertex array
 */
public int addVertex(final WorldVertex vertex) {

    int len;
    int index;

    // If we need to allocate more space, do it
    synchronized (this.synch) {
        len = this.worldVertices.length;

        if (this.numWorldVertices == len) {
            setVertexCapacity(len + 128);
            index = len;
            this.firstOpenWorldVertex = len + 1;
        } else {
            index = this.firstOpenWorldVertex;

            for (int i = index + 1; i < len; i++) {

                if (!this.worldVertexStatus.get(i)) {
                    this.firstOpenWorldVertex = i;

                    break;
                }
            }
        }

        this.worldVertices[index] = vertex;
        this.viewVertices[index] = new ViewVertex();
        this.worldVertexStatus.set(index);
        this.numWorldVertices++;
    }

    vertex.setIndexInScene(index);

    return index;
}

/**
 * Attempts to add a world-coordinate vertex at a particular index.
 *
 * @param   vertex  the vertex to add
 * @param   index   the index at which to add the vertex
 * @throws  SparseArrayException  if the supplied index is not valid (either the array does not
 *                                contain the index, or there is already a vertex at the index)
 */
public void addVertex(final WorldVertex vertex, final int index) throws SparseArrayException {

    synchronized (this.synch) {

        if (this.worldVertices.length > index) {

            if (this.worldVertexStatus.get(index)) {
                throw new SparseArrayException("Index " + index + " already occupied");
            }

            this.worldVertices[index] = vertex;
            this.worldVertexStatus.set(index);
            this.numWorldVertices++;

            vertex.setIndexInScene(index);
        } else {
            throw new SparseArrayException("Index " + index + " out of bounds");
        }
    }
}

/**
 * Adds a vertex to the surface. The newly added vertex is not referenced by any faces.
 *
 * @param   xCoord  the X coordinate
 * @param   yCoord  the Y coordinate
 * @param   zCoord  the Z coordinate
 * @return  the immutable index of the vertex
 */
public WorldVertex addVertex(final double xCoord, final double yCoord, final double zCoord) {

    int index;
    WorldVertex vertex;
```

```
        vertex = new WorldVertex(xCoord, yCoord, zCoord);
        index = addVertex(vertex);
        vertex.setIndexInScene(index);

        return vertex;
    }

    /**
     * Removes a world-coordinate vertex from the array.
     *
     * @param   index  the index of the vertex to remove
     * @return  <code>true</code> if a vertex was present at the requested index and was removed;
     *          <code>false</code> otherwise
     */
    public WorldVertex removeVertex(final int index) {

        WorldVertex obj;

        synchronized (this.synch) {

            if (this.worldVertexStatus.get(index)) {
                obj = this.worldVertices[index];
                this.worldVertexStatus.clear(index);
                this.numWorldVertices--;

                if (index < this.firstOpenWorldVertex) {
                    this.firstOpenWorldVertex = index;
                }

                obj.setIndexInScene(-1);
            } else {
                obj = null;
            }
        }

        return obj;
    }

    /**
     * Gets the index of the next filled index in the world vertex array after (or including) a
     * given starting index.
     *
     * @param   index  the starting index
     * @return  the next filled index, or -1 if no indices are filled from the start index on
     */
    public int nextWorldVertexFilled(final int index) {

        return this.worldVertexStatus.nextSetBit(index);
    }

    /**
     * Gets the index of the next filled index in the view vertex array after (or including) a
     * given starting index.
     *
     * @param   index  the starting index
     * @return  the next filled index, or -1 if no indices are filled from the start index on
     */
    public int nextViewVertexFilled(final int index) {

        return this.viewVertexStatus.nextSetBit(index);
    }

    /**
     * Gets a world-coordinate vertex from the array.
     *
     * @param   index  the index of the vertex to get
     * @return  the vertex
     */
    public WorldVertex getWorldVertex(final int index) {

        synchronized (this.synch) {
            return this.worldVertices[index];
        }
    }

    /**
     * Gets a view-coordinate vertex from the array.
     *
     * @param   index  the index of the vertex to get
     * @return  the vertex
     */
    public ViewVertex getViewVertex(final int index) {

        synchronized (this.synch) {
            return this.viewVertices[index];
        }
    }
```

```
/**
 * Gets the length of the allocated array of world-coordinate vertices.
 *
 * @return  the length of the array (includes empty positions)
 */
public int vertexCapacity() {

    synchronized (this.synch) {
        return this.worldVertices.length;
    }
}

/**
 * Sets the capacity of the world-coordinate vertex array, allocating new objects if needed. If
 * the current array is already large enough to accommodate the request, no changes are made
 * (that is, the capacity after this request may be larger than the requested capacity).
 *
 * @param  newCap  the new capacity
 */
public void setVertexCapacity(final int newCap) {

    int len;
    WorldVertex[] newWorld;
    ViewVertex[] newView;

    synchronized (this.synch) {
        len = this.worldVertices.length;

        if (newCap > len) {
            newWorld = new WorldVertex[newCap];
            newView = new ViewVertex[newCap];
            System.arraycopy(this.worldVertices, 0, newWorld, 0, len);
            System.arraycopy(this.viewVertices, 0, newView, 0, len);
            this.worldVertices = newWorld;
            this.viewVertices = newView;
        }
    }
}

/**
 * Adds a face to the world-coordinate face array, increasing allocated storage if needed.
 *
 * @param   face  the face to add
 * @return  the index of the newly added object in the face array
 */
public int addFace(final WorldFace face) {

    int len;
    int index;

    // If we need to allocate more space, do it
    synchronized (this.synch) {
        len = this.worldFaces.length;

        if (this.numWorldFaces == len) {
            setFaceCapacity(len + 128);
            index = len;
            this.firstOpenWorldFace = len + 1;
        } else {
            index = this.firstOpenWorldFace;

            for (int i = index + 1; i < len; i++) {

                if (!this.worldFaceStatus.get(i)) {
                    this.firstOpenWorldFace = i;

                    break;
                }
            }
        }

        this.worldFaces[index] = face;
        this.worldFaceStatus.set(index);
        this.viewFaces[index] = new ViewFace();
        this.numWorldFaces++;
    }

    face.setIndexInScene(index);

    return index;
}

/**
 * Attempts to add a world-coordinate face at a particular index.
 *
 * @param   face   the face to add
```

```
     * @param   index  the index at which to add the face
     * @throws  SparseArrayException  if the supplied index is not valid (either the array does not
     *                                contain the index, or there is already a face at the index)
     */
    public void addFace(final WorldFace face, final int index) throws SparseArrayException {

        synchronized (this.synch) {

            if (this.worldFaces.length > index) {

                if (this.worldFaceStatus.get(index)) {
                    throw new SparseArrayException("Index " + index + " already occupied");
                }

                this.worldFaces[index] = face;
                this.worldFaceStatus.set(index);
                this.numWorldFaces++;
            } else {
                throw new SparseArrayException("Index " + index + " out of bounds");
            }
        }
    }

    /**
     * Adds a vertex to the surface. The newly added vertex is not referenced by any faces.
     *
     * @param   vert0  the X coordinate
     * @param   vert1  the Y coordinate
     * @param   vert2  the Z coordinate
     * @return  the immutable index of the vertex
     */
    public WorldFace addFace(final WorldVertex vert0, final WorldVertex vert1,
        final WorldVertex vert2) {

        int index;
        WorldFace face;

        face = new WorldFace(vert0, vert1, vert2);
        index = addFace(face);
        face.setIndexInScene(index);

        return face;
    }

    /**
     * Removes a world-coordinate face from the array.
     *
     * @param   index  the index of the face to remove
     * @return  <code>true</code> if a face was present at the requested index and was removed;
     *          <code>false</code> otherwise
     */
    public WorldFace removeFace(final int index) {

        WorldFace obj;

        synchronized (this.synch) {

            if (this.worldFaceStatus.get(index)) {
                obj = this.worldFaces[index];
                this.worldFaceStatus.clear(index);
                this.numWorldFaces--;

                if (index < this.firstOpenWorldFace) {
                    this.firstOpenWorldFace = index;
                }
            } else {
                obj = null;
            }
        }

        return obj;
    }

    /**
     * Gets the index of the next filled index in the world face array after (or including) a given
     * starting index.
     *
     * @param   index  the starting index
     * @return  the next filled index, or -1 if no indices are filled from the start index on
     */
    public int nextWorldFaceFilled(final int index) {

        return this.worldFaceStatus.nextSetBit(index);
    }

    /**
     * Gets the index of the next filled index in the view face array after (or including) a given
```

```
     * starting index.
     *
     * @param   index  the starting index
     * @return  the next filled index, or -1 if no indices are filled from the start index on
     */
    public int nextViewFaceFilled(final int index) {

        return this.viewFaceStatus.nextSetBit(index);
    }

    /**
     * Gets a world-coordinate face from the array.
     *
     * @param   index  the index of the face to get
     * @return  the face
     */
    public WorldFace getWorldFace(final int index) {

        synchronized (this.synch) {
            return this.worldFaces[index];
        }
    }

    /**
     * Gets a view-coordinate face from the array.
     *
     * @param   index  the index of the face to get
     * @return  the face
     */
    public ViewFace getViewFace(final int index) {

        synchronized (this.synch) {
            return this.viewFaces[index];
        }
    }

    /**
     * Gets the length of the allocated array of world-coordinate faces.
     *
     * @return  the length of the array (includes empty positions)
     */
    public int faceCapacity() {

        synchronized (this.synch) {
            return this.worldFaces.length;
        }
    }

    /**
     * Sets the capacity of the world-coordinate face array, allocating new objects if needed. If
     * the current array is already large enough to accommodate the request, no changes are made
     * (that is, the capacity after this request may be larger than the requested capacity).
     *
     * @param  newCap  the new capacity
     */
    public void setFaceCapacity(final int newCap) {

        int len;
        WorldFace[] newWorld;
        ViewFace[] newView;

        synchronized (this.synch) {
            len = this.worldVertices.length;

            if (newCap > len) {
                newWorld = new WorldFace[newCap];
                newView = new ViewFace[newCap];
                System.arraycopy(this.worldFaces, 0, newWorld, 0, len);
                System.arraycopy(this.viewFaces, 0, newView, 0, len);
                this.worldFaces = newWorld;
                this.viewFaces = newView;
            }
        }
    }

    /**
     * Adds a light to the scene.
     *
     * @param  light  the light to add
     */
    public void addLight(final Light light) {

        int len;
        Light[] newArray;

        synchronized (this.synch) {
            len = this.worldLights.length;
```

```
            newArray = new Light[len + 1];

            if (len > 0) {
                System.arraycopy(this.worldLights, 0, newArray, 0, len);
            }

            newArray[len] = light;
            this.worldLights = newArray;

            newArray = new Light[len + 1];

            if (len > 0) {
                System.arraycopy(this.viewLights, 0, newArray, 0, len);
            }

            newArray[len] = new Light(0, 0, 0, light.getColor());
            this.viewLights = newArray;
        }
    }

    /**
     * Gets the number of lights. NOTE: Lights cannot be added once rendering starts, or the light
     * count will be off and we may try to use a view-space light that has not been transformed. In
     * the future, we should treat lights like vertices and faces.
     *
     * @return  the number of lights
     */
    public int numLights() {

        return this.viewLights.length;
    }

    /**
     * Gets a particular light in view coordinates.
     *
     * @param   index  the index of the light to get
     * @return  the light
     */
    public Light getViewLight(final int index) {

        return this.viewLights[index];
    }

    /**
     * Adds a based vector to the scene.
     *
     * @param  vec  the based vector to add
     */
    public void addBasedVector(final BasedVector3 vec) {

        int len;
        BasedVector3[] newArray;

        synchronized (this.synch) {
            len = this.worldBasedVectors.length;
            newArray = new BasedVector3[len + 1];

            if (len > 0) {
                System.arraycopy(this.worldBasedVectors, 0, newArray, 0, len);
            }

            newArray[len] = vec;
            this.worldBasedVectors = newArray;

            newArray = new BasedVector3[len + 1];

            if (len > 0) {
                System.arraycopy(this.viewBasedVectors, 0, newArray, 0, len);
            }

            newArray[len] = new BasedVector3();
            this.viewBasedVectors = newArray;
        }
    }

    /**
     * Gets the number of based vectors. NOTE: Based vectors cannot be added once rendering starts,
     * or the vector count will be off and we may try to use a view-space vector that has not been
     * transformed. In the future, we should treat based vectors like vertices and faces.
     *
     * @return  the number of based vectors
     */
    public int numBasedVectors() {

        return this.worldBasedVectors.length;
    }
```

```
/**
 * Gets a particular based vector in view coordinates.
 *
 * @param   index  the index of the based vector to get
 * @return  the based vector
 */
public BasedVector3 getBasedVector(final int index) {

    return this.viewBasedVectors[index];
}

/**
 * Transforms all vertices and faces from world to view coordinates using a specified camera.
 * This is an atomic operation that locks the vertex and face arrays for the duration so we get
 * a consistent state in the transformed object set.
 *
 * @param  camera  the camera that holds the transformation we will use
 */
public void worldToView(final Camera camera) {

    int len;

    synchronized (this.synch) {

        // Transform all vertices
        this.viewVertexStatus.clear();
        this.viewVertexStatus.or(this.worldVertexStatus);

        len = this.worldVertices.length;

        for (int i = 0; i < len; i++) {

            if (this.worldVertexStatus.get(i)) {
                this.viewVertices[i].transformFrom(camera, this.worldVertices[i]);
            }
        }

        // Transform all faces (normal vectors)
        this.viewFaceStatus.clear();
        this.viewFaceStatus.or(this.worldFaceStatus);

        len = this.worldFaces.length;

        for (int i = 0; i < len; i++) {

            if (this.worldFaceStatus.get(i)) {
                this.viewFaces[i].transformFrom(camera, this.viewVertices, this.worldFaces[i]);
            }
        }

        // Transform lights
        len = this.worldLights.length;

        for (int i = 0; i < len; i++) {
            camera.transformPoint(this.worldLights[i], this.viewLights[i]);
        }

        // Transform based vectors
        len = this.worldBasedVectors.length;

        for (int i = 0; i < len; i++) {
            camera.transformPoint(this.worldBasedVectors[i], this.viewBasedVectors[i]);
            camera.transformVec(this.worldBasedVectors[i], this.viewBasedVectors[i]);
        }
    }
}

/**
 * Applies the perspective transformation to vertices to map the view frustum to the X and Y
 * range -1 \le X \le 1 and -1 \le Y \le 1, and where Z values are now positive (a left-handed
 * frame).
 *
 * @param  camera  the camera that holds the transformation we will use
 */
public void viewToNormalized(final Camera camera) {

    int len;

    // Transform vertices
    len = this.viewVertices.length;

    for (int i = 0; i < len; i++) {

        if (this.viewVertexStatus.get(i)) {
            camera.toNormalizedDeviceCoordinates(this.viewVertices[i]);
        }
    }
```

```
            // Transform based vectors
            len = this.worldBasedVectors.length;

            for (int i = 0; i < len; i++) {
                camera.toNormalizedDeviceCoordinates(this.viewBasedVectors[i]);
            }
        }

        /**
         * Applies the perspective transformation to vertices to map the view frustum to the X and Y
         * range -1 \le X \le 1 and -1 \le Y \le 1, and where Z values are now positive (a left-handed
         * frame).
         *
         * @param  camera  the camera that holds the transformation we will use
         * @param  width   the width of the image being rendered
         * @param  height  the height of the image being rendered
         */
        public void normalizedToScreen(final Camera camera, final int width, final int height) {

            int len;

            // Transform vertices
            len = this.viewVertices.length;

            for (int i = 0; i < len; i++) {

                if (this.viewVertexStatus.get(i)) {
                    camera.pointToScreen(width, height, this.viewVertices[i]);
                }
            }

            // Transform based vectors
            len = this.worldBasedVectors.length;

            for (int i = 0; i < len; i++) {
                camera.vecToScreen(width, height, this.viewBasedVectors[i]);
                camera.pointToScreen(width, height, this.viewBasedVectors[i]);
            }
        }
    }

    package com.srbenoit.render;

    import java.awt.Color;
    import com.srbenoit.geom.Vector3;

    /**
     * A triangular face in view coordinates. This class extends <code>Tuple3</code>, where the
     * superclass fields represent the normal vector to the face.
     */
    public class ViewFace extends Vector3 {

        /** the index of this face in the scene where it is contained */
        private int indexInScene;

        /** the first vertex in the face */
        private ViewVertex vertex0;

        /** the second vertex in the face */
        private ViewVertex vertex1;

        /** the third vertex in the face */
        private ViewVertex vertex2;

        /** the light color of the face */
        private Color color;

        /** flag indicating face has been culled */
        private boolean culled;

        /**
         * Constructs a new <code>ViewFace</code>.
         */
        public ViewFace() {

            super();
        }

        /**
         * Gets the index of this face in the scene.
         *
         * @return  the index
         */
        public int getIndexInScene() {

            return this.indexInScene;
```

```
}

/**
 * Gets the first vertex in the face.
 *
 * @return  the vertex
 */
public ViewVertex getVertex0() {

    return this.vertex0;
}

/**
 * Gets the second vertex in the face.
 *
 * @return  the vertex
 */
public ViewVertex getVertex1() {

    return this.vertex1;
}

/**
 * Gets the third vertex in the face.
 *
 * @return  the vertex
 */
public ViewVertex getVertex2() {

    return this.vertex2;
}

/**
 * Copies a world-coordinate face object into a view-coordinate face object. This sets the
 * vertex references in the view-space object to the view-space vertex objects that correspond
 * to the world-space vertices that the world-space face references.
 *
 * @param  camera     the camera to use to transform points and vectors
 * @param  vertices   the view-space vertices
 * @param  world      the world-space face from which to copy
 */
public void transformFrom(final Camera camera, final ViewVertex[] vertices,
    final WorldFace world) {

    // Get the view vertices that correspond to the correct world vertices
    this.vertex0 = vertices[world.getVertex0().getIndexInScene()];
    this.vertex1 = vertices[world.getVertex1().getIndexInScene()];
    this.vertex2 = vertices[world.getVertex2().getIndexInScene()];

    this.indexInScene = world.getIndexInScene();

    // Transform the face normal vector
    camera.transformVec(world, this);
}

/**
 * Sets the face color based on lighting (not yet on material).
 *
 * @param  red  the red component (0-1, unclamped)
 * @param  grn  the green component (0-1, unclamped)
 * @param  blu  the blue component (0-1, unclamped)
 */
public void setColor(final double red, final double grn, final double blu) {

    double colR;
    double colG;
    double colB;

    colR = (red > 1.0) ? 1.0 : ((red < 0.0) ? 0.0 : red);
    colG = (grn > 1.0) ? 1.0 : ((grn < 0.0) ? 0.0 : grn);
    colB = (blu > 1.0) ? 1.0 : ((blu < 0.0) ? 0.0 : blu);

    this.color = new Color((int) (colR * 255), (int) (colG * 255), (int) (colB * 255), 128);
}

/**
 * Gets the face color.
 *
 * @return  the color
 */
public Color getColor() {

    return this.color;
}

/**
 * Sets the flag indicating whether this face has been culled.
```

```
     *
     * @param  isCulled  <code>true</code> if the face is to be culled, <code>false if not
     */
    public void setCulled(final boolean isCulled) {

        this.culled = isCulled;
    }

    /**
     * Tests whether this face has been culled.
     *
     * @return  <code>true</code> if the face is to be culled, <code>false if not
     */
    public boolean isCulled() {

        return this.culled;
    }
}

package com.srbenoit.render;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * An iterator over the view faces in a scene.
 */
public class ViewFaceIterator implements Iterator<ViewFace> {

    /** the scene being iterated */
    private final transient Scene scene;

    /** the index of the current element, -1 if finished or not yet started */
    private transient int currentElement;

    /** the index of the next element to be returned, -1 if finished */
    private transient int nextElement;

    /**
     * Constructs a new <code>ViewFaceIterator</code>.
     *
     * @param  sourceScene  the scene being iterated
     */
    public ViewFaceIterator(final Scene sourceScene) {

        this.scene = sourceScene;
        this.nextElement = scene.nextViewFaceFilled(0);
        this.currentElement = -1;
    }

    /**
     * Returns <code>true</code> if the iteration has more elements. In other words, returns <code>
     * true</code> if <code>next</code> would return an element rather than throwing an exception.
     *
     * @return  <code>true</code> if the iteration has more elements; <code>false</code> otherwise
     */
    public boolean hasNext() {

        return this.nextElement != -1;
    }

    /**
     * Returns the next element in the iteration.
     *
     * @return  the next element in the iteration
     * @throws  NoSuchElementException  if the iteration has no more elements
     */
    public ViewFace next() throws NoSuchElementException {

        ViewFace result;

        if (this.nextElement == -1) {
            throw new NoSuchElementException();
        }

        result = this.scene.getViewFace(this.nextElement);
        this.currentElement = this.nextElement;
        this.nextElement = this.scene.nextViewFaceFilled(this.currentElement + 1);

        return result;
    }

    /**
     * Removes from the underlying collection the last element returned by the iterator (optional
     * operation). This method can be called only once per call to <code>next</code>.
     *
     * <p>The behavior of an iterator is unspecified if the underlying collection is modified while
```

```
     * the iteration is in progress in any way other than by calling this method.
     *
     * @throws  IllegalStateException  if the <code>next</code> method has not yet been called, or
     *                                 the <code>remove</code> method has already been called after
     *                                 the last call to the <code>next</code> method
     */
    public void remove() {

        throw new UnsupportedOperationException("Cannot remove from a view face array");
    }
}

package com.srbenoit.render;

import com.srbenoit.geom.Point3;
import com.srbenoit.geom.Vector3;

/**
 * A vertex in view coordinates.
 */
public class ViewVertex extends Point3 {

    /** the index of this vertex in the scene where it is contained */
    private int indexInScene;

    /** the vertex normal */
    private final Vector3 normal;

    /**
     * Constructs a new <code>ViewVertex</code>.
     */
    public ViewVertex() {

        super();

        this.normal = new Vector3();
    }

    /**
     * Gets the index of this face in the scene.
     *
     * @return  the index
     */
    public int getIndexInScene() {

        return this.indexInScene;
    }

    /**
     * Gets the vertex normal.
     *
     * @return  the normal vector
     */
    public Vector3 getNormal() {

        return this.normal;
    }

    /**
     * Copies a world-coordinate vertex object into a view-coordinate vertex object.
     *
     * @param  camera  the camera to use to transform points and vectors
     * @param  world   the world-space vertex from which to copy
     */
    public void transformFrom(final Camera camera, final WorldVertex world) {

        this.indexInScene = world.getIndexInScene();

        // Transform the vertex position
        camera.transformPoint(world, this);

        // Build and transform the vertex normal
        world.buildNormal(this.normal);
        camera.transformVec(this.normal, this.normal);
    }
}

package com.srbenoit.render;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * An iterator over the view vertices in a scene.
 */
public class ViewVertexIterator implements Iterator<ViewVertex> {
```

```
    /** the scene being iterated */
    private final transient Scene scene;

    /** the index of the current element, -1 if finished or not yet started */
    private transient int currentElement;

    /** the index of the next element to be returned, -1 if finished */
    private transient int nextElement;

    /**
     * Constructs a new <code>ViewVertexIterator</code>.
     *
     * @param  sourceScene  the scene being iterated
     */
    public ViewVertexIterator(final Scene sourceScene) {

        this.scene = sourceScene;
        this.nextElement = scene.nextViewVertexFilled(0);
        this.currentElement = -1;
    }

    /**
     * Returns <code>true</code> if the iteration has more elements. In other words, returns <code>
     * true</code> if <code>next</code> would return an element rather than throwing an exception.
     *
     * @return  <code>true</code> if the iteration has more elements; <code>false</code> otherwise
     */
    public boolean hasNext() {

        return this.nextElement != -1;
    }

    /**
     * Returns the next element in the iteration.
     *
     * @return  the next element in the iteration
     * @throws  NoSuchElementException  if the iteration has no more elements
     */
    public ViewVertex next() throws NoSuchElementException {

        ViewVertex result;

        if (this.nextElement == -1) {
            throw new NoSuchElementException();
        }

        result = this.scene.getViewVertex(this.nextElement);
        this.currentElement = this.nextElement;
        this.nextElement = this.scene.nextViewVertexFilled(this.currentElement + 1);

        return result;
    }

    /**
     * Removes from the underlying collection the last element returned by the iterator (optional
     * operation). This method can be called only once per call to <code>next</code>.
     *
     * <p>The behavior of an iterator is unspecified if the underlying collection is modified while
     * the iteration is in progress in any way other than by calling this method.
     *
     * @throws  IllegalStateException  if the <code>next</code> method has not yet been called, or
     *                                 the <code>remove</code> method has already been called after
     *                                 the last call to the <code>next</code> method
     */
    public void remove() {

        throw new UnsupportedOperationException("Cannot remove from a view vertex array");
    }
}

package com.srbenoit.render;

import com.srbenoit.geom.Vector3;

/**
 * A triangular face that makes up a scene. This class extends <code>Tuple3</code>, where the
 * superclass fields represent the normal vector to the face.
 */
public class WorldFace extends Vector3 {

    /** the index of this face in the scene where it is contained */
    private int indexInScene;

    /** the first vertex in the face */
    private WorldVertex vertex0;

    /** the second vertex in the face */
```

```
    private WorldVertex vertex1;

    /** the third vertex in the face */
    private WorldVertex vertex2;

    /**
     * Constructs a new <code>WorldFace</code>.
     *
     * @param  vert0  the first vertex in the face
     * @param  vert1  the second vertex in the face
     * @param  vert2  the third vertex in the face
     */
    public WorldFace(final WorldVertex vert0, final WorldVertex vert1, final WorldVertex vert2) {

        super();

        this.vertex0 = vert0;
        this.vertex1 = vert1;
        this.vertex2 = vert2;

        computeNormal();

        this.indexInScene = -1;
    }

    /**
     * Sets the index of this face in the scene.
     *
     * @param  index  the index
     */
    public void setIndexInScene(final int index) {

        this.indexInScene = index;
    }

    /**
     * Gets the index of this face in the scene.
     *
     * @return  the index
     */
    public int getIndexInScene() {

        return this.indexInScene;
    }

    /**
     * Gets the first vertex in the face.
     *
     * @return  the vertex
     */
    public WorldVertex getVertex0() {

        return this.vertex0;
    }

    /**
     * Gets the second vertex in the face.
     *
     * @return  the vertex
     */
    public WorldVertex getVertex1() {

        return this.vertex1;
    }

    /**
     * Gets the third vertex in the face.
     *
     * @return  the vertex
     */
    public WorldVertex getVertex2() {

        return this.vertex2;
    }

    /**
     * Gets a specified vertex in the face.
     *
     * @param   index  the index of the vertex may be outside the legal range - this value will be
     *                 taken modulo the number of vertices before use)
     * @return  the vertex
     */
    public WorldVertex getVertex(final int index) {

        int actual;
        WorldVertex vert;
```

```
        actual = index % 3;

        if (actual < 0) {
            actual += 3;
        }

        switch (actual) {

        case 0:
            vert = this.vertex0;
            break;

        case 1:
            vert = this.vertex1;
            break;

        default:
            vert = this.vertex2;
            break;
        }

        return vert;
    }

    /**
     * Computes the normal vector.
     */
    public final void computeNormal() {

        double dx1;
        double dy1;
        double dz1;
        double dx2;
        double dy2;
        double dz2;
        double crossX;
        double crossY;
        double crossZ;

        dx1 = this.vertex1.getPosX() - this.vertex0.getPosX();
        dy1 = this.vertex1.getPosY() - this.vertex0.getPosY();
        dz1 = this.vertex1.getPosZ() - this.vertex0.getPosZ();

        dx2 = this.vertex2.getPosX() - this.vertex0.getPosX();
        dy2 = this.vertex2.getPosY() - this.vertex0.getPosY();
        dz2 = this.vertex2.getPosZ() - this.vertex0.getPosZ();

        crossX = (dy1 * dz2) - (dz1 * dy2);
        crossY = (dz1 * dx2) - (dx1 * dz2);
        crossZ = (dx1 * dy2) - (dy1 * dx2);

        setVec(crossX, crossY, crossZ);
        normalize();
    }
}

package com.srbenoit.render;

import com.srbenoit.geom.Point3;
import com.srbenoit.geom.Vector3;

/**
 * A vertex in world coordinates that a scene is composed of.
 */
public class WorldVertex extends Point3 {

    /** the index of this vertex in the scene where it is contained */
    private int indexInScene;

    /** the number of faces this vertex is part of */
    private int numFaces;

    /** the list of faces that contain the vertex */
    private WorldFace[] faces;

    /** the index of this vertex in each face in the faces array */
    private int[] indexInFaces;

    /**
     * Constructs a new <code>WorldVertex</code>.
     */
    public WorldVertex() {

        super();

        this.numFaces = 0;
        this.faces = new WorldFace[5];
```

```
        this.indexInFaces = new int[5];
        this.indexInScene = -1;
    }

    /**
     * Constructs a new <code>WorldVertex</code>.
     *
     * @param  xCoord  the X coordinate
     * @param  yCoord  the Y coordinate
     * @param  zCoord  the Z coordinate
     */
    public WorldVertex(final double xCoord, final double yCoord, final double zCoord) {

        super(xCoord, yCoord, zCoord);

        this.numFaces = 0;
        this.faces = new WorldFace[5];
        this.indexInFaces = new int[5];
        this.indexInScene = -1;
    }

    /**
     * Sets the index of this vertex in the scene.
     *
     * @param  index  the index
     */
    public void setIndexInScene(final int index) {

        this.indexInScene = index;
    }

    /**
     * Gets the index of this vertex in the scene.
     *
     * @return  the index
     */
    public int getIndexInScene() {

        return this.indexInScene;
    }

    /**
     * Adds a face to the list of faces this vertex is part of.
     *
     * @param  face   the face
     * @param  index  the index of this vertex in the face
     */
    public void addFace(final WorldFace face, final int index) {

        WorldFace[] newArray;
        int[] newIndices;

        if (this.numFaces == this.faces.length) {
            newArray = new WorldFace[this.numFaces + 5];
            System.arraycopy(this.faces, 0, newArray, 0, this.numFaces);
            this.faces = newArray;

            newIndices = new int[this.numFaces + 5];
            System.arraycopy(this.indexInFaces, 0, newIndices, 0, this.numFaces);
            this.indexInFaces = newIndices;
        }

        this.faces[this.numFaces] = face;
        this.indexInFaces[this.numFaces] = index;
        this.numFaces++;
    }

    /**
     * Gets the number of faces this vertex takes part in.
     *
     * @return  the number of faces
     */
    public int getNumFaces() {

        return this.numFaces;
    }

    /**
     * Gets a face this vertex takes part in.
     *
     * @param   index  the index of the face
     * @return  the face
     */
    public WorldFace getFace(final int index) {

        return this.faces[index];
    }
```

```
    /**
     * Gets the index of this vertex in one of the faces that it is part of.
     *
     * @param   index  the index of the face
     * @return  the index of this vertex in the specified face
     */
    public int getIndexInFace(final int index) {

        return this.indexInFaces[index];
    }

    /**
     * Computes the average of the normal vectors of all attached faces and normalizes the result
     * to obtain the normal vector at the vertex.
     *
     * @param  normal  the vector in which to place the computed normal
     */
    public void buildNormal(final Vector3 normal) {

        double x;
        double y;
        double z;

        if (this.numFaces == 0) {
            normal.setVec(0, 0, 1);
        } else {
            x = 0;
            y = 0;
            z = 0;

            for (int i = 0; i < this.numFaces; i++) {
                x += this.faces[i].getVecX();
                y += this.faces[i].getVecY();
                z += this.faces[i].getVecZ();
            }

            normal.setVec(x, y, z);
            normal.normalize(); // This method already deals with the 0-length case
        }
    }
}
```

E.4 Mathematics

E.4.1 Math Utilities (com.srbenoit.math)

This package contains some basic math utilities.

```
package com.srbenoit.math;

import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * A library with fast approximations to transcendental functions.
 */
public final class FastMath extends LoggedObject {

    /** a lookup tables used to speed up square computations */
    private final static int[] TABLE = {
            0, 16, 22, 27, 32, 35, 39, 42, 45, 48, 50, 53, 55, 57, 59, 61, 64, 65, 67, 69, 71, 73,
            75, 76, 78, 80, 81, 83, 84, 86, 87, 89, 90, 91, 93, 94, 96, 97, 98, 99, 101, 102, 103,
            104, 106, 107, 108, 109, 110, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122,
            123, 124, 125, 126, 128, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139,
            140, 141, 142, 143, 144, 144, 145, 146, 147, 148, 149, 150, 150, 151, 152, 153, 154,
            155, 155, 156, 157, 158, 159, 160, 160, 161, 162, 163, 163, 164, 165, 166, 167, 167,
            168, 169, 170, 170, 171, 172, 173, 173, 174, 175, 176, 176, 177, 178, 178, 179, 180,
            181, 181, 182, 183, 183, 184, 185, 185, 186, 187, 187, 188, 189, 189, 190, 191, 192,
            192, 193, 193, 194, 195, 195, 196, 197, 197, 198, 199, 199, 200, 201, 201, 202, 203,
            203, 204, 204, 205, 206, 206, 207, 208, 208, 209, 209, 210, 211, 211, 212, 212, 213,
            214, 214, 215, 215, 216, 217, 217, 218, 218, 219, 219, 220, 221, 221, 222, 222, 223,
            224, 224, 225, 225, 226, 226, 227, 227, 228, 229, 229, 230, 230, 231, 231, 232, 232,
            233, 234, 234, 235, 235, 236, 236, 237, 237, 238, 238, 239, 240, 240, 241, 241, 242,
            242, 243, 243, 244, 244, 245, 245, 246, 246, 247, 247, 248, 248, 249, 249, 250, 250,
            251, 251, 252, 252, 253, 253, 254, 254, 255
        };
```

```
/**
 * Private constructor to prevent instantiation.
 */
private FastMath() {

    super();
}

/**
 * A faster replacement for (int)(java.lang.Math.sqrt(x)). Completely accurate for x < 289...
 *
 * @param   val  the value whose square root is to be taken
 * @return  the approximate square root
 */
public static int fastSqrt(final long val) {

    int root;
    int intVal;

    if (val > Integer.MAX_VALUE) {
        root = (int) (Math.sqrt(val));
    } else {
        intVal = (int) val;

        if (intVal >= 0x10000) {

            if (intVal >= 0x1000000) {

                if (intVal >= 0x10000000) {

                    if (intVal >= 0x40000000) {
                        root = (TABLE[intVal >> 24] << 8);
                    } else {
                        root = (TABLE[intVal >> 22] << 7);
                    }
                } else if (intVal >= 0x4000000) {
                    root = (TABLE[intVal >> 20] << 6);
                } else {
                    root = (TABLE[intVal >> 18] << 5);
                }
            } else if (intVal >= 0x100000) {

                if (intVal >= 0x400000) {
                    root = (TABLE[intVal >> 16] << 4);
                } else {
                    root = (TABLE[intVal >> 14] << 3);
                }
            } else if (intVal >= 0x40000) {
                root = (TABLE[intVal >> 12] << 2);
            } else {
                root = (TABLE[intVal >> 10] << 1);
            }
        } else if (intVal >= 0x100) {

            if (intVal >= 0x1000) {

                if (intVal >= 0x4000) {
                    root = TABLE[intVal >> 8];
                } else {
                    root = (TABLE[intVal >> 6] >> 1);
                }
            } else if (intVal >= 0x400) {
                root = (TABLE[intVal >> 4] >> 2);
            } else {
                root = (TABLE[intVal >> 2] >> 3);
            }
        } else if (intVal >= 0) {
            root = TABLE[intVal] >> 4;
        } else {
            throw new IllegalArgumentException("Attemt to take the square root of " + intVal);
        }
    }

    return root;
}

/**
 * Main method to exercise methods in this class.
 *
 * @param  args  command-line arguments
 */
public static void main(final String... args) {

    long start;
    long end;
    int total1;
```

```
        int total2;
        int root;
        int est;
        float err;
        float maxerr = 0;

        // Compare values
        for (int i = 0; i < 2000000000; i += 20) {
            root = (int) Math.sqrt(i);
            est = FastMath.fastSqrt(i);
            err = (float) Math.abs((root - est) / (double) root);

            if (err > maxerr) {
                maxerr = err;
            }
        }

        LOG.log(Level.INFO, "Maximum error over 0-2000000000 is {0}%", (maxerr * 100));

        // Compare speeds
        total1 = 0;
        start = System.currentTimeMillis();

        for (int i = 0; i < 2000000000; i += 20) {
            total1 += (int) Math.sqrt(i);
        }

        end = System.currentTimeMillis();
        LOG.log(Level.INFO, "Math.sqrt: {0}", (end - start));

        total2 = 0;
        start = System.currentTimeMillis();

        for (int i = 0; i < 2000000000; i += 20) {
            total2 += FastMath.fastSqrt(i);
        }

        end = System.currentTimeMillis();
        LOG.log(Level.INFO, "FastMath.fastSqrt: {0}", (end - start));

        LOG.log(Level.INFO, "Outcomes: {0}, {1}", new Object[] { total1, total2 });
    }
}

package com.srbenoit.math;

/**
 * A data class to store a list of counters (planes) over a series of times.
 */
public class Histogram {

    /** the histogram data (first index is plane, second is time) */
    private final transient int[][] data;

    /**
     * Constructs a new <code>Histogram</code>.
     *
     * @param  numPlanes  the number of planes of data to include in the histogram
     * @param  numTimes   the number of time indexes of data array to build
     */
    public Histogram(final int numPlanes, final int numTimes) {

        if ((numPlanes <= 0) || (numTimes <= 0)) {
            throw new IllegalArgumentException("Histogram must have length");
        }

        this.data = new int[numPlanes][numTimes];
    }

    /**
     * Gets the number of planes in the histogram.
     *
     * @return  the number of planes
     */
    public int getNumPlanes() {

        return this.data.length;
    }

    /**
     * Gets the number of time points in the histogram.
     *
     * @return  the number of time points
     */
    public int getNumTimes() {

        return this.data[0].length;
```

```
}

/**
 * Increments the current (time index 0) value in the histogram.
 *
 * @param  plane  the plane of the data value to increment
 */
public void incrementValue(final int plane) {

    synchronized (this.data) {
        this.data[plane][0]++;
    }
}

/**
 * Gets a particular value in the histogram.
 *
 * @param   plane  the plane of the data value to get
 * @param   time   the time index of the data value to get
 * @return  the data value
 */
public int getValue(final int plane, final int time) {

    synchronized (this.data) {
        return this.data[plane][time];
    }
}

/**
 * Gets the list of all values in the histogram.
 *
 * @return  the list of values
 */
public int[][] getValues() {

    int[][] values;

    synchronized (this.data) {
        values = new int[this.data.length][];

        for (int i = 0; i < this.data.length; i++) {
            values[i] = this.data[i].clone();
        }
    }

    return values;
}

/**
 * Sets the current (index 0) value of a particular plane.
 *
 * @param  plane     the plane for which to set the value
 * @param  newValue  the new value
 */
public void setValue(final int plane, final int newValue) {

    synchronized (this.data) {
        this.data[plane][0] = newValue;
    }
}

/**
 * Sets the list of all values in the histogram.
 *
 * @param  newValues  the list of new values
 */
public void setValues(final int[][] newValues) {

    synchronized (this.data) {

        if (newValues.length != this.data.length) {
            throw new IllegalArgumentException("Histogram length mismatch");
        }

        for (int plane = 0; plane < this.data.length; plane++) {

            if (newValues[plane].length != this.data[plane].length) {
                throw new IllegalArgumentException("Histogram length mismatch");
            }

            System.arraycopy(newValues[plane], 0, this.data[plane], 0, this.data[plane].length);
        }
    }
}

/**
 * Shifts each counter in the histogram to the next larger slot, discarding the data in the
```

```
     * highest slot, and initializes the lowest slot to a particular value.
     *
     * <p>This is useful when the slots represent windows of time, and we move into a new window -
     * the oldest count is discarded, all counts are shifted to the next older window, and the new
     * window is started.
     */
    public void shiftHigher() {

        synchronized (this.data) {

            for (int plane = 0; plane < this.data.length; plane++) {

                // shift all histogram data up one slot
                for (int i = this.data.length - 1; i > 0; i--) {
                    this.data[plane][i] = this.data[plane][i - 1];
                }

                // initialize the [0] entry in each plane to zero
                this.data[plane][0] = 0;
            }
        }
    }
}
```

E.4.2 Graphing (com.srbenoit.math.grapher)

This package collects many one-off general utilities to simplify common functions or provide functionality missing in the JDK.

```
package com.srbenoit.math.grapher;

/**
 * An interface for functions that can be graphed.
 */
public interface Graphable {

    /**
     * Gets the number of dimensions of the graph (2 for a function of a single value).
     *
     * @return  the number of dimensions of the graph
     */
    int graphDimensions();

    /**
     * Gets a default domain over which the graph will show the main features of the function. This
     * domain should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the left and right endpoints of the default domain
     */
    double[] defaultDomain();

    /**
     * Gets a default range over which the graph will show the main features of the function. This
     * range should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the lower and upper limits of the default range
     */
    double[] defaultRange();

    /**
     * Computes the graph values at a coordinate or coordinates. The number of coordinates needed is
     * one less than the dimension.
     *
     * @param   coordinates  the list of coordinates
     * @return  the graph values at that location
     */
    double valueAt(double... coordinates);
}
```

```
package com.srbenoit.math.grapher;

import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
```

```
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.ui.UIUtilities;

/**
 * A graphing tool that renders the graph of a function.
 */
public class Grapher extends LoggedObject {

    /** PNG file format */
    private static final String PNG = "PNG";

    /** JPEG file format */
    private static final String JPEG = "JPEG";

    /** GIF file format */
    private static final String GIF = "GIF";

    /** BMP file format */
    private static final String BMP = "BMP";

    /** WBMP file format */
    private static final String WBMP = "WBMP";

    /** the image that displays the graph */
    private final transient GraphImage image;

    /**
     * Create a <code>Grapher</code> to render a function's graph.
     *
     * @param  xSize  the width of graphs to generate
     * @param  ySize  the height of graphs to generate
     */
    public Grapher(final int xSize, final int ySize) {

        super();

        this.image = new GraphImage(xSize, ySize);
    }

    /**
     * Gets the <code>GraphImage</code> that this grapher draws to.
     *
     * @return  the <code>GraphImage</code>
     */
    public GraphImage getGraphImage() {

        return this.image;
    }

    /**
     * Renders the graph.
     *
     * @param  functions  the functions to be graphed
     */
    public void graph(final Graphable... functions) {

        this.image.graph(functions);
    }

    /**
     * Exports the image of the graph to a file.
     *
     * @param   file  the file to which to export
     * @throws  IOException  if there is an error writing the file
     */
    public void export(final File file) throws IOException {

        String fname;
        int pos;
        String ext;
        String format;
        File target;

        fname = file.getName();
        pos = fname.lastIndexOf('.');

        if (pos == -1) {
            format = JPEG;
            target = new File(file.getParent(), file.getName() + ".jpg");
        } else {
            ext = fname.substring(pos + 1).toUpperCase();

            if (("JPG".equals(ext)) || (JPEG.equals(ext))) {
                format = JPEG;
                target = file;
```

```
            } else if (PNG.equals(ext)) {
                format = PNG;
                target = file;
            } else if (GIF.equals(ext)) {
                format = GIF;
                target = file;
            } else if (BMP.equals(ext)) {
                format = BMP;
                target = file;
            } else if (WBMP.equals(ext)) {
                format = WBMP;
                target = file;
            } else {
                format = JPEG;
                target = new File(file.getParent(), file.getName() + ".jpg");
            }
        }

        ImageIO.write(this.image.getImage(), format, target);
    }

    /**
     * Builds a <code>JFrame</code> and displays the grapher in that frame.
     *
     * @param  title  the frame title
     */
    public void showInFrame(final String title) {

        FrameBuilder builder;

        builder = new FrameBuilder(this, title);
        SwingUtilities.invokeLater(builder);
    }

    /**
     * Class to construct a frame and display the grapher as its content pane from within the AWT
     * event thread.
     */
    private static class FrameBuilder implements Runnable {

        /** the <code>Grapher</code> whose panel is to be shown in the frame */
        private final Grapher grapher;

        /** the frame title */
        private final String frameTitle;

        /**
         * Constructs a new <code>FrameBuilder</code>.
         *
         * @param  grapherToShow  the <code>Grapher</code> whose panel is to be shown in the frame
         * @param  title          the frame title
         */
        protected FrameBuilder(final Grapher grapherToShow, final String title) {

            this.grapher = grapherToShow;
            this.frameTitle = title;
        }

        /**
         * Builds the frame and installs the <code>Grapher</code>'s panel is its content pane.
         */
        public void run() {

            JFrame frame;
            JLabel label;

            frame = new JFrame(this.frameTitle);
            frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
            label = new JLabel(new ImageIcon(this.grapher.getGraphImage().getImage()));
            frame.setContentPane(label);
            frame.pack();

            UIUtilities.positionFrame(frame, 0.5, 0.5);

            frame.setVisible(true);
        }
    }
}

package com.srbenoit.math.grapher;

import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
```

```
/**
 * A container for a generated image of a graph.
 */
public class GraphImage {

    /** the width of the graph image */
    private final int width;

    /** the height of the graph image */
    private final int height;

    /** the offscreen image to which to render */
    private final transient BufferedImage image;

    /** the functions to be graphed */
    private transient Graphable[] toGraph;

    /**
     * Create a <code>GraphImage</code> to render a function's graph.
     *
     * @param  theWidth   the width of the image
     * @param  theHeight  the height of the image
     */
    public GraphImage(final int theWidth, final int theHeight) {

        super();

        this.width = theWidth;
        this.height = theHeight;
        this.image = new BufferedImage(theWidth, theHeight, BufferedImage.TYPE_INT_RGB);
    }

    /**
     * Gets the image in which the graph will be rendered.
     *
     * @return  the image
     */
    public BufferedImage getImage() {

        return this.image;
    }

    /**
     * Draws the graph, including axes, grid lines, etc.
     *
     * @param  graphables  the <code>Graphable</code> objects to draw
     */
    public void graph(final Graphable... graphables) {

        double[] domain;
        double[] range;
        Graphable fxn;
        Graphics2D g2d;

        this.toGraph = graphables.clone();

        g2d = (Graphics2D) this.image.getGraphics();
        g2d.setColor(Color.WHITE);
        g2d.fillRect(0, 0, this.width, this.height);

        // Compute the overall domain and range
        domain = new double[] { Double.MAX_VALUE, -Double.MAX_VALUE };
        range = new double[] { Double.MAX_VALUE, -Double.MAX_VALUE };

        for (int inx = 0; inx < this.toGraph.length; inx++) {
            fxn = this.toGraph[inx];

            if (fxn.defaultDomain()[0] < domain[0]) {
                domain[0] = fxn.defaultDomain()[0];
            }

            if (fxn.defaultDomain()[1] > domain[1]) {
                domain[1] = fxn.defaultDomain()[1];
            }

            if (fxn.defaultRange()[0] < range[0]) {
                range[0] = fxn.defaultRange()[0];
            }

            if (fxn.defaultRange()[1] > range[1]) {
                range[1] = fxn.defaultRange()[1];
            }
        }

        drawAxesAndGrid(g2d, domain, range);
        drawFunctions(g2d, domain, range);
    }
```

```
/**
 * Draws a set of vertical and horizontal axes and grid lines.
 *
 * @param  grx      the <code>Graphics</code> to which to draw
 * @param  domain   the lower and upper limit for the domain (x axis)
 * @param  range    the lower and upper limit for the range (y axis)
 */
private void drawAxesAndGrid(final Graphics2D grx, final double[] domain,
    final double[] range) {

    double[] xGrid;
    double[] yGrid;
    double current;
    double delta;
    double pct;
    int pixel;
    String lbl;
    Font origFont;
    Font font;
    FontMetrics metrics;
    int strWidth;
    int xLabelY;
    int yLabelX;
    boolean yLabelRight = false;
    boolean xLabelUp = false;
    Color gridColor;

    gridColor = new Color(220, 220, 255);
    xGrid = new double[2];
    yGrid = new double[2];

    // Build the font for axis labels and get its metrics
    font = new Font("Dialog", Font.PLAIN, 11);
    origFont = grx.getFont();
    grx.setFont(font);
    metrics = grx.getFontMetrics();
    strWidth = metrics.stringWidth("0.000001");

    // Compute grid sizes in the x and y directions
    computeGrid(domain, xGrid);
    computeGrid(range, yGrid);

    // Draw the vertical grid lines and the Y axis, and as we go, figure
    // out the X position for our labels of the Y axis
    yLabelX = -1;
    delta = domain[1] - domain[0];
    current = xGrid[0];

    while (current < domain[1]) {
        pct = (current - domain[0]) / delta;
        pixel = (int) ((this.width * pct) + 0.5);

        if (Math.abs(current / xGrid[1]) < 0.00001) {

            if ((pixel + 2) < (this.width - strWidth)) {
                yLabelX = pixel;
            } else {
                yLabelX = pixel;
                yLabelRight = true;
            }

            grx.setColor(Color.BLACK);
        } else {
            grx.setColor(gridColor);
        }

        grx.drawLine(pixel, 0, pixel, this.height);
        current += xGrid[1];
    }

    if (yLabelX == -1) {

        if (domain[1] > 0) {
            yLabelX = 2;
        } else {
            yLabelX = this.width - 2;
            yLabelRight = false;
        }
    }

    // Draw the horizontal grid lines and the X axis, and as we go, figure
    // out the Y position for our labels of the X axis
    xLabelY = -1;
    delta = range[1] - range[0];
    current = yGrid[0];
```

```
while (current < range[1]) {
    pct = (current - range[0]) / delta;
    pixel = this.height - (int) ((this.height * pct) + 0.5);

    if (Math.abs(current / yGrid[1]) < ((yGrid[1] - yGrid[0]) * 0.00001)) {

        if ((pixel + 2) < (this.width - strWidth)) {
            xLabelY = pixel;
        } else {
            xLabelY = pixel;
            xLabelUp = true;
        }

        grx.setColor(Color.BLACK);
    } else {
        grx.setColor(gridColor);
    }

    grx.drawLine(0, pixel, this.width, pixel);
    current += yGrid[1];
}

if (xLabelY == -1) {

    if (domain[1] > 0) {
        xLabelY = 2;
    } else {
        xLabelY = this.height - 2;
        xLabelUp = true;
    }
}

grx.setColor(Color.BLACK);

// Draw the labels for the X axis
delta = domain[1] - domain[0];
current = xGrid[0];

if (Math.abs(current) < (xGrid[1] / 1000)) {
    current = 0;
}

while (current < domain[1]) {
    pct = (current - domain[0]) / delta;
    pixel = (int) ((this.width * pct) + 0.5);

    if (pixel != yLabelX) {
        lbl = Float.toString((float) current);

        strWidth = metrics.stringWidth(lbl);

        if (xLabelUp) {
            grx.drawString(lbl, pixel - (strWidth / 2), xLabelY - 1);
        } else {
            grx.drawString(lbl, pixel - (strWidth / 2), xLabelY + metrics.getAscent() + 1);
        }
    }

    current += xGrid[1];
}

// Draw the labels for the Y axis
delta = range[1] - range[0];
current = yGrid[0];
current = xGrid[0];

if (Math.abs(current) < (yGrid[1] / 1000)) {
    current = 0;
}

while (current < range[1]) {
    pct = (current - range[0]) / delta;
    pixel = this.height - (int) ((this.height * pct) + 0.5);

    lbl = Float.toString((float) current);

    strWidth = metrics.stringWidth(lbl);

    if (yLabelRight) {
        grx.drawString(lbl, yLabelX - strWidth - 4, pixel + metrics.getAscent() + 1);
    } else {
        grx.drawString(lbl, yLabelX + 4, pixel + metrics.getAscent() + 1);
    }

    current += yGrid[1];
}
```

```
        grx.setFont(origFont);
    }

    /**
     * Computes the settings for a grid.
     *
     * @param  limits  the lower and upper limits of the graph along the axis of interest (may be
     *                 adjusted by this method)
     * @param  grid    an array of 2 doubles which will contain the coordinate of the first grid
     *                 line (in [0]) and the step size (in [1]) of the grid
     */
    private void computeGrid(final double[] limits, final double[] grid) {

        double extents;
        double exp;
        double scale;
        double scaledExtents;
        double scaledStep;
        double scaledLower;
        int sign;

        // See what extent of the real line the limits encompass
        extents = limits[1] - limits[0];

        // Get the order of magnitude of the extent, then compute 10 raised to
        // that power. For example, if the extent is 3.5, the scale will be
        // 1. Dividing the extents by the scale will produce a scaled extents
        // value between 1 and 10.

        exp = Math.log10(extents);

        if (exp >= 0) {
            scale = Math.pow(10, (int) exp);
        } else {
            scale = Math.pow(10, (int) (exp - 1));
        }

        scaledExtents = extents / scale;
        scaledLower = limits[0] / scale;

        // System.out.println("Extents = " + (float)extents + ", scale = "
        // + scale + ", Scaled extents = " + (float)scaledExtents
        // + ", scaled lower = " + scaledLower);

        // TODO: Refine the choices below based on number of pixels

        sign = (scaledLower >= 0) ? 1 : 0;

        if (scaledExtents >= 6) {
            scaledStep = 1.0;
            grid[0] = (int) (scaledLower + (sign * 0.99));
        } else if (scaledExtents >= 3) {
            scaledStep = 0.5;
            grid[0] = (int) ((scaledLower + (sign * 0.499)) * 2);
        } else if (scaledExtents >= 2) {
            scaledStep = 0.25;
            grid[0] = (int) ((scaledLower + (sign * 0.249)) * 4);
        } else if (scaledExtents >= 1.4) {
            scaledStep = 0.2;
            grid[0] = (int) ((scaledLower + (sign * 0.199)) * 5);
        } else {
            scaledStep = 0.1;
            grid[0] = (int) ((scaledLower + (sign * 0.099)) * 10);
        }

        grid[1] = scaledStep * scale;

        if (Math.abs(grid[0] - limits[0]) < (grid[1] / 10)) {
            limits[0] -= grid[1] / 10;
        }
    }

    /**
     * Draws a set of vertical and horizontal axes and grid lines.
     *
     * @param  grx     the <code>Graphics</code> to which to draw
     * @param  domain  the lower and upper limit for the domain (x axis)
     * @param  range   the lower and upper limit for the range (y axis)
     */
    private void drawFunctions(final Graphics2D grx, final double[] domain, final double[] range) {

        double[] pos;
        Graphable fxn;
        double value;
        double prior;
        double frac;
        int yPrior;
```

```
            int yCurrent;
            int col;

            pos = new double[] { domain[0] };

            for (int inx = 0; inx < this.toGraph.length; inx++) {
                fxn = this.toGraph[inx];
                col = 255 * inx / this.toGraph.length;

                prior = fxn.valueAt(pos);
                yPrior = (int) ((prior - range[0]) / (range[1] - range[0]) * this.height);

                grx.setColor(new Color(col, 0, 255 - col)); // NOPMD SRB

                for (int x = 1; x < this.width; x++) {
                    frac = (double) x / this.width;
                    pos[0] = domain[0] + (frac * (domain[1] - domain[0]));
                    value = fxn.valueAt(pos);

                    if (!Double.isNaN(value)) {
                        yCurrent = (int) ((value - range[0]) / (range[1] - range[0]) * this.height);

                        if (!Double.isNaN(prior)) {
                            grx.drawLine(x - 1, this.height - yPrior, x, this.height - yCurrent);
                        }

                        yPrior = yCurrent;
                        prior = value;
                    }
                }
            }
        }
    }
}

package com.srbenoit.math.grapher;

/**
 * A generic graphable that accepts a list of points then produces a piecewise linear graph that
 * interpolates the points linearly.
 */
public class PointListGraph implements Graphable {

    /** the list of X coordinates */
    private final transient double[] xCoords;

    /** the Y coordinates corresponding to the X coordinates */
    private final transient double[] yCoords;

    /** the lowest X value */
    private final transient double[] xRange;

    /** the highest X value */
    private final transient double[] yRange;

    /**
     * Constructs a new <code>PointListGraph</code>.
     *
     * @param  xValues  the list of X coordinates
     * @param  yValues  the list of Y coordinates
     */
    public PointListGraph(final double[] xValues, final double[] yValues) {

        boolean sorted;
        double tempX;
        double tempY;

        if ((xValues == null) || (yValues == null)) {
            throw new IllegalArgumentException("null array");
        }

        if (xValues.length == 0) {
            throw new IllegalArgumentException("zero length array");
        }

        if (xValues.length != yValues.length) {
            throw new IllegalArgumentException("Value length mismatch");
        }

        this.xCoords = xValues.clone();
        this.yCoords = yValues.clone();

        // Put the lists in order by X coordinate (silly bubble sort)
        sorted = false;

        while (!sorted) {
            sorted = true;
```

```
            for (int i = 1; i < this.xCoords.length; i++) {

                if (this.xCoords[i - 1] > this.xCoords[i]) {
                    tempX = this.xCoords[i];
                    tempY = this.yCoords[i];
                    this.xCoords[i] = this.xCoords[i - 1];
                    this.yCoords[i] = this.yCoords[i - 1];
                    this.xCoords[i - 1] = tempX;
                    this.yCoords[i - 1] = tempY;
                    sorted = false;
                }
            }
        }

        this.xRange = new double[2];
        this.yRange = new double[2];

        // Compute domain and range limits
        this.xRange[0] = this.xCoords[0];
        this.xRange[1] = this.xCoords[0];
        this.yRange[0] = this.yCoords[0];
        this.yRange[1] = this.yCoords[0];

        for (int i = 1; i < this.xCoords.length; i++) {

            if (!Double.isInfinite(this.xCoords[i])) {

                if (this.xCoords[i] < this.xRange[0]) {
                    this.xRange[0] = this.xCoords[i];
                }

                if (this.xCoords[i] > this.xRange[1]) {
                    this.xRange[1] = this.xCoords[i];
                }
            }

            if (!Double.isInfinite(this.yCoords[i])) {

                if (this.yCoords[i] < this.yRange[0]) {
                    this.yRange[0] = this.yCoords[i];
                }

                if (this.yCoords[i] > this.yRange[1]) {
                    this.yRange[1] = this.yCoords[i];
                }
            }
        }

        // Pad the domain and range to get some margin on a plot
        tempX = (this.xRange[1] - this.xRange[0]) * 0.1;
        tempY = (this.yRange[1] - this.yRange[0]) * 0.1;
        this.xRange[0] -= tempX;
        this.xRange[1] += tempX;
        this.yRange[0] -= tempY;
        this.yRange[1] += tempY;
    }

    /**
     * Gets the number of dimensions of the graph (2 for a function of a single value).
     *
     * @return  the number of dimensions of the graph
     */
    public int graphDimensions() {

        return 2;
    }

    /**
     * Gets a default domain over which the graph will show the main features of the function. This
     * domain should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the left and right endpoints of the default domain
     */
    public double[] defaultDomain() {

        return this.xRange.clone();
    }

    /**
     * Gets a default range over which the graph will show the main features of the function. This
     * range should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the lower and upper limits of the default range
     */
    public double[] defaultRange() {

        return this.yRange.clone();
```

```
    }

    /**
     * Computes the graph value at a coordinate or coordinates. The number of coordinates needed is
     * one less than the dimension.
     *
     * @param   coordinates  the list of coordinates
     * @return  the graph value at that location
     */
    public double valueAt(final double... coordinates) {

        double value;
        double frac;

        if (coordinates.length != 1) {
            throw new IllegalArgumentException("Invalid number of coordinates");
        }

        value = Double.NaN;

        for (int i = 1; i < this.xCoords.length; i++) {

            if (this.xCoords[i] >= coordinates[0]) {

                if (this.xCoords[i - 1] <= coordinates[0]) {
                    frac = (coordinates[0] - this.xCoords[i - 1])
                        / (this.xCoords[i] - this.xCoords[i - 1]);
                    value = this.yCoords[i - 1] + (frac * (this.yCoords[i] - this.yCoords[i - 1]));
                }

                break;
            }
        }

        return value;
    }
}
```

E.4.3 Linear Algebra (com.srbenoit.math.linear)

This package contains classes to perform linear algebra in arbitrary dimension, and to support distributed processing of large matrices by many threads or processes.

```
package com.srbenoit.math.linear;

/**
 * An exception thrown by point and vector arrays when invalid operations are attempted.
 */
public final class ArrayException extends RuntimeException {

    /** version number for serialization */
    private static final long serialVersionUID = -8557346213421131643L;

    /**
     * Constructs a new <code>ArrayException</code> with <code>null</code> as its detail message.
     */
    public ArrayException() {

        super();
    }

    /**
     * Constructs a new <code>ArrayException</code> with the specified detail message.
     *
     * @param  message  the detail message
     */
    public ArrayException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>ArrayException</code> with the specified detail message and cause.
     *
     * <p>Note that the detail message associated with <code>cause</code> is <i>not</i>
     * automatically incorporated in this runtime exception's detail message.
     *
```

```
     * @param  message  the detail message
     * @param  cause    the cause
     */
    public ArrayException(final String message, final Throwable cause) {

        super(message, cause);
    }

    /**
     * Constructs a new <code>ArrayException</code> with the specified cause and a detail message of
     * <code>(cause==null ? null : cause.toString())</code> (which typically contains the class and
     * detail message of <code>cause</code> ).
     *
     * @param  cause  the cause
     */
    public ArrayException(final Throwable cause) {

        super(cause);
    }
}

package com.srbenoit.math.linear;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import java.text.DecimalFormat;
import java.util.Arrays;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.util.EvaluationException;

/**
 * Manages a matrix of doubles, and includes ways for multiple processes to distribute the work of
 * populating the matrix. A process can request the next uncomputed array element, which marks the
 * element as in-progress. Includes methods to compute the L-U decomposition of the matrix, and
 * perform back-substitution using the result to solve systems of the form A x = b. This also
 * includes methods to save and reload the matrix state, so a lengthy process can be stopped and
 * restarted.
 */
public class DMatrix extends LoggedObject {

    /** characters used to read and write in hex */
    private static final char[] HEX = "0123456789ABCDEF".toCharArray();

    /** the matrix data */
    private final transient double[][] data;

    /** a set of flags for each data element */
    private final transient byte[][] flags;

    /** the row of the first uncompleted element */
    private transient int firstUncRow = 0;

    /** the column of the first uncompleted element */
    private transient int firstUncCol = 0;

    /** the L-U decomposition of the matrix */
    private transient double[][] lud;

    /** the permuted row indexes for the decomposed matrix */
    private transient int[] ludIndexes;

    /** <code>true</code> if an even number of row interchanges; <code>false</code> if odd */
    private transient boolean ludExchangesEven = true;

    /**
     * Constructs a new <code>DMatrix</code> whose values are initialized to zero.
     *
     * @param  numRows     the number of rows
     * @param  numColumns  the number of columns
     */
    public DMatrix(final int numRows, final int numColumns) {

        if (numRows < 1) {
            throw new IllegalArgumentException("Invalid number of rows: " + numRows);
        }

        if (numColumns < 1) {
            throw new IllegalArgumentException("Invalid number of columns: " + numColumns);
        }

        this.data = new double[numRows][numColumns];
        this.flags = new byte[numRows][numColumns];
```

```
}

/**
 * Constructs a new <code>DMatrix</code> whose values are initialized to provided values
 *
 * @param  numRows      the number of rows
 * @param  numColumns   the number of columns
 * @param  values       the initial values for the matrix, in row-major order
 */
public DMatrix(final int numRows, final int numColumns, final double... values) {

    this(numRows, numColumns);

    int index;
    byte mask;

    mask = (byte) (0x01 << MatrixElementFlag.COMPLETED.flagBit());

    if (values.length != (numRows * numColumns)) {
        throw new IllegalArgumentException(
            "Number of supplied values does not match matrix size");
    }

    index = 0;

    for (int i = 0; i < numRows; i++) {

        for (int j = 0; j < numColumns; j++) {
            this.data[i][j] = values[index];
            this.flags[i][j] |= mask;
            index++;
        }
    }

    this.firstUncRow = this.data.length;
    this.firstUncCol = 0;
}

/**
 * Constructs a new <code>DMatrix</code> whose values are a copy of the values in another
 * <code>DMatrix</code>.
 *
 * @param  matrix  the matrix to copy
 */
public DMatrix(final DMatrix matrix) {

    // Safe outside synch because matrix size is immutable
    this(matrix.data.length, matrix.data[0].length);

    synchronized (matrix) {

        for (int i = 0; i < this.data.length; i++) {
            System.arraycopy(this.data[i], 0, matrix.data[i], 0, this.data[i].length);
            System.arraycopy(this.flags[i], 0, matrix.flags[i], 0, this.flags[i].length);
        }

        this.firstUncRow = matrix.firstUncRow;
        this.firstUncCol = matrix.firstUncCol;
    }
}

/**
 * Gets the number of rows in this matrix.
 *
 * @return  the number of rows
 */
public int numRows() {

    // Not synchronized because matrix size is immutable
    return this.data.length;
}

/**
 * Gets the number of columns in this matrix.
 *
 * @return  the number of columns
 */
public int numColumns() {

    // Not synchronized because matrix size is immutable
    return this.data[0].length;
}

/**
 * Gets the row and column of the next uncompleted (and not currently in progress) element, and
 * mark it as in-progress.
 *
```

```
     * @return  a two-integer array containing the row and column of the element (<code>null</code>
     *          if there are no uncompleted elements in the matrix)
     */
    public int[] getUncompletedElement() {

        int[] rowCol = null;

        synchronized (this) {

            if (this.firstUncRow < this.data.length) {
                setFlag(this.firstUncRow, this.firstUncCol, MatrixElementFlag.IN_PROGRESS, true);
                rowCol = new int[] { this.firstUncRow, this.firstUncCol };
                advanceFirstUncompleted();
            }
        }

        return rowCol;
    }

    /**
     * Moves the index of the first uncompleted element forward to the next uncompleted (and not
     * in-progress) element.
     */
    private void advanceFirstUncompleted() {

        // NOTE: called only from within synchronized block.

        while (this.firstUncRow < this.data.length) {

            if (isCompleted(this.firstUncRow, this.firstUncCol)
                    || isInProgress(this.firstUncRow, this.firstUncCol)) {

                this.firstUncCol++;

                if (this.firstUncCol >= this.data[0].length) {
                    this.firstUncRow++;
                    this.firstUncCol = 0;
                }
            } else {
                break;
            }
        }
    }

    /**
     * Clears the value in a matrix cell and mark it as not completed, not in progress, and not out
     * of range.
     *
     * @param  row  the index of the data value's row
     * @param  col  the index of the data value's column
     */
    public void clearElementValue(final int row, final int col) {

        synchronized (this) {

            if (isCompleted(row, col)) {
                this.data[row][col] = 0;
                setFlag(row, col, MatrixElementFlag.COMPLETED, false);
                setFlag(row, col, MatrixElementFlag.IN_PROGRESS, false);
                setFlag(row, col, MatrixElementFlag.OUT_OF_RANGE, false);
                clearLud();

                // If this row/column is before the first uncompleted
                // row/column, move the first uncompleted one back to here.
                if (this.firstUncRow > row) {
                    this.firstUncRow = row;
                    this.firstUncCol = col;
                } else if ((this.firstUncRow == row) && (this.firstUncCol > col)) {
                    this.firstUncCol = col;
                }
            }
        }
    }

    /**
     * Copies values from another matrix.
     *
     * @param  src  a matrix from which to copy values
     */
    public void set(final DMatrix src) {

        synchronized (this) {

            if ((src.numRows() == numRows()) && (src.numColumns() == numColumns())) {

                for (int i = 0; i < this.data.length; i++) {
                    System.arraycopy(src.data[i], 0, this.data[i], 0, this.data[i].length);
```

```
                System.arraycopy(src.flags[i], 0, this.flags[i], 0, this.flags[i].length);
            }

            this.firstUncRow = src.firstUncRow;
            this.firstUncCol = src.firstUncCol;
        } else {
            throw new IllegalArgumentException("Matrix size mismatch");
        }
    }
}

/**
 * Stores a value in a matrix cell and marks it as completed and no longer in progress.
 *
 * @param  row    the index of the data value's row
 * @param  col    the index of the data value's column
 * @param  value  the data value
 */
public void set(final int row, final int col, final double value) {

    synchronized (this) {

        if (this.data[row][col] != value) {
            this.data[row][col] = value;
            clearLud();
        }

        setFlag(row, col, MatrixElementFlag.COMPLETED, true);
        setFlag(row, col, MatrixElementFlag.IN_PROGRESS, false);

        if ((row == this.firstUncRow) && (col == this.firstUncCol)) {
            advanceFirstUncompleted();
        }
    }
}

/**
 * Gets the data value of a matrix element.
 *
 * @param   row  the row (indexed from 0)
 * @param   col  the column (indexed from 0)
 * @return  the data value stored at that row/column
 */
public double get(final int row, final int col) {

    synchronized (this) {
        return this.data[row][col];
    }
}

/**
 * Gets a particular flag value.
 *
 * @param   row         the row (indexed from 0)
 * @param   col         the column (indexed from 0)
 * @param   whichFlag   the flag to test
 * @return  the flag value
 */
private boolean getFlag(final int row, final int col, final MatrixElementFlag whichFlag) {

    byte mask;

    mask = (byte) (0x01 << whichFlag.flagBit());

    synchronized (this) {
        return (this.flags[row][col] & mask) == mask;
    }
}

/**
 * Tests whether a matrix element is marked as completed.
 *
 * @param   row  the row (indexed from 0)
 * @param   col  the column (indexed from 0)
 * @return  <code>true</code> if completed; <code>false</code> if not
 */
public boolean isCompleted(final int row, final int col) {

    return getFlag(row, col, MatrixElementFlag.COMPLETED);
}

/**
 * Tests whether a matrix element is marked as in progress.
 *
 * @param   row  the row (indexed from 0)
 * @param   col  the column (indexed from 0)
 * @return  <code>true</code> if in progress; <code>false</code> if not
```

```
     */
    public boolean isInProgress(final int row, final int col) {

        return getFlag(row, col, MatrixElementFlag.IN_PROGRESS);
    }

    /**
     * Tests whether a matrix element is marked as out of range.
     *
     * @param   row  the row (indexed from 0)
     * @param   col  the column (indexed from 0)
     * @return  <code>true</code> if out of range; <code>false</code> if not
     */
    public boolean isOutOfRange(final int row, final int col) {

        return getFlag(row, col, MatrixElementFlag.OUT_OF_RANGE);
    }

    /**
     * Tests whether a matrix element is marked as tagged.
     *
     * @param   row  the row (indexed from 0)
     * @param   col  the column (indexed from 0)
     * @return  <code>true</code> if tagged; <code>false</code> if not
     */
    public boolean isTagged(final int row, final int col) {

        return getFlag(row, col, MatrixElementFlag.TAGGED);
    }

    /**
     * Sets a particular flag value.
     *
     * @param  row         the row (indexed from 0)
     * @param  col         the column (indexed from 0)
     * @param  whichFlag   the flag to set
     * @param  flagValue   <code>true</code> to flag the value; <code>false</code> to clear the flag
     */
    public void setFlag(final int row, final int col, final MatrixElementFlag whichFlag,
        final boolean flagValue) {

        byte mask;

        mask = (byte) (0x01 << whichFlag.flagBit());

        synchronized (this) {

            if (flagValue) {
                this.flags[row][col] |= mask;
            } else {
                this.flags[row][col] &= (~mask);
            }
        }
    }

    /**
     * Tests whether the entire matrix has been computed.
     *
     * @return  <code>true</code> if matrix is complete; <code>false</code> otherwise
     */
    private boolean isMatrixComplete() {

        byte mask;
        boolean complete = true;

        mask = (byte) (0x01 << MatrixElementFlag.COMPLETED.flagBit());

        synchronized (this) {

            for (int row = 0; row < this.flags.length; row++) {

                for (int col = 0; col < this.flags[0].length; col++) {

                    if ((this.flags[row][col] & mask) != mask) {
                        complete = false;

                        break;
                    }
                }
            }
        }

        return complete;
    }

    /**
     * Generates the <code>String</code> representation of the matrix.
```

```
     *
     * @return  the string representation
     */
    @Override public String toString() {

        DecimalFormat format;
        StringBuilder str;
        int numRows;
        int numCols;
        String[][] values;
        int[] widths;
        int delta;
        String crlf;

        format = new DecimalFormat("#######");
        crlf = System.getProperty("line.separator");

        numRows = this.data.length;
        numCols = this.data[0].length;
        widths = new int[numCols];

        synchronized (this) {

            // Generate String representations of each value and get the
            // maximum width of each
            values = new String[numRows][numCols];

            for (int r = 0; r < numRows; r++) {

                for (int c = 0; c < numCols; c++) {
                    values[r][c] = format.format(this.data[r][c]);

                    if (values[r][c].length() > widths[c]) {
                        widths[c] = values[r][c].length();
                    }
                }
            }

            str = new StringBuilder(50);

            for (int r = 0; r < numRows; r++) {
                str.append('[');
                str.append('␣');

                for (int c = 0; c < numCols; c++) {
                    str.append(values[r][c]);
                    delta = widths[c] - values[r][c].length();

                    while (delta > 0) {
                        str.append('␣');
                        delta--;
                    }

                    str.append('␣');
                }

                str.append(']');
                str.append(crlf);
            }
        }

        return str.toString();
    }

    /**
     * Computes the result of multiplying this matrix on the right by a given matrix.
     *
     * @param   mat  the matrix by which to multiply
     * @return  the product [this] x [mat]
     * @throws  EvaluationException  if the matrix dimensions are not compatible
     */
    public DMatrix mul(final DMatrix mat) throws EvaluationException {

        int thisNumRows;
        int thisNumCols;
        int matNumRows;
        int matNumCols;
        DMatrix product;
        double sum;
        String err;

        // Check the sizes this.numCols == mat.numRows
        thisNumRows = numRows();
        thisNumCols = numColumns();
        matNumRows = mat.numRows();
        matNumCols = mat.numColumns();
```

```
        if (thisNumCols != matNumRows) {
            err = "Matrix dimensions not compatible for multiplication ("
                + thisNumRows + "x" + thisNumCols + " matrix multiplied by "
                + matNumRows + "x" + matNumCols + " matrix)";
            LOG.warning(err);
            throw new EvaluationException(err);
        }

        product = new DMatrix(thisNumRows, matNumCols);

        synchronized (this) {

            synchronized (mat) {

                for (int r = 0; r < thisNumRows; r++) {

                    for (int c = 0; c < matNumCols; c++) {
                        sum = 0;

                        for (int k = 0; k < thisNumCols; k++) {
                            sum += get(r, k) * mat.get(k, c);
                        }

                        product.set(r, c, sum);
                    }
                }
            }
        }

        return product;
    }

    /**
     * Computes the result of multiplying this matrix on the right by a given vector.
     *
     * @param   vector  the vector by which to multiply
     * @return  the product [this] x [vector]
     * @throws  EvaluationException  if the vector and matrix sizes are incompatible
     */
    public TupleN transform(final TupleN vector) throws EvaluationException {

        int thisNumRows;
        int thisNumCols;
        TupleN product;
        double sum;
        String err;

        // Check the sizes this.numCols == vector.length
        thisNumCols = numColumns();

        if (thisNumCols != vector.getDimension()) {
            err = "Vector must have the same number of rows as the matrix has columns";
            LOG.warning(err);
            throw new EvaluationException(err);
        }

        thisNumRows = numRows();

        product = new TupleN(vector.getDimension(), false);

        synchronized (this) {

            synchronized (vector) {

                for (int r = 0; r < thisNumRows; r++) {
                    sum = 0;

                    for (int c = 0; c < thisNumCols; c++) {
                        sum += vector.get(c) * get(r, c);
                    }

                    product.set(r, sum);
                }
            }
        }

        return product;
    }

    /**
     * Computes the L-U decomposition of the matrix.
     *
     * @throws  EvaluationException          if the matrix is not square
     * @throws  SingularMatrixException      if the matrix is singular
     * @throws  MatrixNotCompleteException   if the matrix has not been completely computed
     */
    public void luDecompose() throws EvaluationException, SingularMatrixException,
```

```
    MatrixNotCompleteException {

    int size;
    double big;
    double temp;
    double[] scaling;
    String err;

    if (!isMatrixComplete()) {
        throw new MatrixNotCompleteException(
            "Cannot compute L-U decomposition of an incomplete matrix");
    }

    size = numRows();

    // L-U Decomposition works only for square matrices!
    if (size != numColumns()) {
        err = "Attempt to perform L-U decomposition on a non-square (" + size + "x"
            + this.data[0].length + ") matrix";
        LOG.warning(err);
        throw new EvaluationException(err);
    }

    synchronized (this) {

        // Prepare the data areas and copy the matrix
        this.lud = new double[size][];

        for (int row = 0; row < size; row++) {
            this.lud[row] = this.data[row].clone();
        }

        this.ludIndexes = new int[size];
        this.ludExchangesEven = true;

        // Loop over rows to get implicit scaling information
        scaling = new double[size];

        // Loop over rows to get implicit scaling information
        for (int row = 0; row < size; row++) {
            big = 0.0;

            for (int col = 0; col < size; col++) {
                temp = Math.abs(this.lud[row][col]);

                if (temp > big) {
                    big = temp;
                }
            }

            if (big == 0.0) {
                err = "Attempt to perform L-U decomposition on a singular matrix";
                LOG.warning(err);
                throw new SingularMatrixException(err);
            }

            scaling[row] = 1.0 / big;
        }

        // This is the loop over columns in Crout's method
        croutsMethod(size, scaling);
    }
}

/**
 * Clears the LUD data structures.
 */
private void clearLud() {

    // NOTE: called only from within synchronized block.

    this.lud = null;
    this.ludIndexes = null;
    this.ludExchangesEven = true;
}

/**
 * Implementation of Crout's Method of performing L-U decomposition.
 *
 * @param   size      the size of the (square) matrix
 * @param   scaling   the scaling factors for the rows
 * @throws  SingularMatrixException  if the matrix is singular
 */
private void croutsMethod(final int size, final double[] scaling)
    throws SingularMatrixException {

    // NOTE: called only from within synchronized block.
```

```
        int row;
        int col;
        int rmax;
        double big;
        double dum;
        double sum;
        String err;

        for (col = 0; col < size; col++) {

            for (row = 0; row < col; row++) {
                sum = this.lud[row][col];

                for (int k = 0; k < row; k++) {
                    sum -= this.lud[row][k] * this.lud[k][col];
                }

                this.lud[row][col] = sum;
            }

            // Search for the largest pivot element.
            big = 0.0;
            rmax = 0;

            for (row = col; row < size; row++) {
                sum = this.lud[row][col];

                for (int k = 0; k < col; k++) {
                    sum -= this.lud[row][k] * this.lud[k][col];
                }

                this.lud[row][col] = sum;
                dum = scaling[row] * Math.abs(sum);

                if (dum >= big) {
                    big = dum;
                    rmax = row;
                }
            }

            // If we need to interchange rows, do so.
            if (col != rmax) {
                interchange(size, rmax, col);
                scaling[rmax] = scaling[col];
            }

            this.ludIndexes[col] = rmax;

            // If pivot element is zero, matrix is singular.
            if (this.lud[col][col] == 0) {
                clearLud();

                err = "Attempt to perform L-U decomposition on a singular matrix";
                LOG.warning(err);
                throw new SingularMatrixException(err);
            }

            // Divide by the pivot element.
            if (col != (size - 1)) {
                dum = 1.0 / this.lud[col][col];

                for (row = col + 1; row < size; row++) {
                    this.lud[row][col] *= dum;
                }
            }
        }
    }

    /**
     * Performs a row interchange.
     *
     * @param  size    the size of the (square) matrix
     * @param  first   the index of the first row
     * @param  second  the index of the second row
     */
    private void interchange(final int size, final int first, final int second) {

        // NOTE: called only from within synchronized block.

        double dum;

        for (int k = 0; k < size; k++) {
            dum = this.lud[first][k];
            this.lud[first][k] = this.lud[second][k];
            this.lud[second][k] = dum;
        }
```

```
        this.ludExchangesEven ^= true;
    }

    /**
     * Solves Ax = b using the L-U decomposition of A computed previously by the <code>
     * luDecompose</code> method.
     *
     * @param   rhs  the right-hand side
     * @return  the x vector that solves the system
     * @throws  EvaluationException          if the matrix is not square
     * @throws  SingularMatrixException      if the matrix is singular
     * @throws  MatrixNotCompleteException   if the matrix has not been completely computed
     */
    public TupleN luSolve(final TupleN rhs) throws EvaluationException, SingularMatrixException,
        MatrixNotCompleteException {

        int size;
        TupleN lhs;
        double sum;
        int indexes;
        int pivot = -1;
        String err;

        synchronized (this) {

            if (this.lud == null) {
                luDecompose();
            }

            size = numColumns();

            if (rhs.getDimension() != size) {
                err = "Right-hand side vector must match number of columns in matrix";
                LOG.warning(err);
                throw new EvaluationException(err);
            }

            lhs = rhs.copy();

            for (int i = 0; i < size; i++) {
                indexes = this.ludIndexes[i];
                sum = lhs.get(indexes);
                lhs.set(indexes, lhs.get(i));

                if (pivot > -1) {

                    for (int j = pivot; j < i; j++) {
                        sum -= this.lud[i][j] * lhs.get(j);
                    }
                } else if (sum != 0) {
                    pivot = i;
                }

                lhs.set(i, sum);
            }

            for (int i = size - 1; i >= 0; i--) {
                sum = lhs.get(i);

                for (int j = i + 1; j < size; j++) {
                    sum -= this.lud[i][j] * lhs.get(j);
                }

                lhs.set(i, sum / this.lud[i][i]);
            }
        }

        return lhs;
    }

    /**
     * Computes the inverse of the matrix.
     *
     * @return  the inverse
     * @throws  EvaluationException          if the matrix is not square
     * @throws  SingularMatrixException      if the matrix is singular
     * @throws  MatrixNotCompleteException   if the matrix has not been completely computed
     */
    public DMatrix inverse() throws EvaluationException, SingularMatrixException,
        MatrixNotCompleteException {

        int size;
        DMatrix inverse;
        TupleN col;
        TupleN vec;
```

```
        if (!isMatrixComplete()) {
            throw new MatrixNotCompleteException("Cannot invert an incomplete matrix");
        }

        size = numRows();

        if (size != numColumns()) {
            LOG.log(Level.WARNING, "Attempt to invert a non-square ({0}x{1}) matrix",
                new Object[] { size, this.data[0].length });
            throw new EvaluationException("Attempt to invert a non-square (" + size + "x"
                + this.data[0].length + ") matrix");
        }

        synchronized (this) {

            // We use the L-U decomposition to take the inverse.
            if (this.lud == null) {
                luDecompose();
            }

            inverse = new DMatrix(size, size);
            col = new TupleN(size, false);

            for (int c = 0; c < size; c++) {
                col.set(c, 1);
                vec = luSolve(col);
                col.set(c, 0);

                for (int r = 0; r < size; r++) {
                    inverse.set(r, c, vec.get(r));
                }
            }
        }

        return inverse;
    }

    /**
     * Computes the transpose of the matrix.
     *
     * @return  the transpose
     * @throws  MatrixNotCompleteException  if the matrix has not been completely computed
     */
    public DMatrix transpose() throws MatrixNotCompleteException {

        DMatrix transpose;

        if (!isMatrixComplete()) {
            throw new MatrixNotCompleteException("Cannot transpose an incomplete matrix");
        }

        transpose = new DMatrix(this.data[0].length, this.data.length);

        synchronized (this) {

            for (int r = 0; r < this.data.length; r++) {

                for (int c = 0; c < this.data[0].length; c++) {

                    if (isCompleted(r, c)) {
                        transpose.set(c, r, this.data[r][c]);
                    }
                }
            }
        }

        return transpose;
    }

    /**
     * Tests whether all entries in the matrix are within \epsilon of the entries in a test matrix.
     * The test performed on each matrix element is "if (Math.abs(thisMatrix[r][c] -
     * refMatrix[r][c]) < epsilon)". If this statement is true for all entries, this method returns
     * true.
     *
     * @param   ref       the reference matrix to compare to
     * @param   epsilon   the tolerance
     * @return  <code>true</code> if all entries are within tolerance, <code>false</code> otherwise
     * @throws  MatrixNotCompleteException  if the matrix has not been completely computed
     */
    public boolean epsilonEquals(final DMatrix ref, final double epsilon)
        throws MatrixNotCompleteException {

        boolean equals;

        if (!isMatrixComplete() || !ref.isMatrixComplete()) {
            throw new MatrixNotCompleteException("Cannot test equality of an incomplete matrix");
```

```
        }

        synchronized (this) {

            synchronized (ref) {

                if ((this.data.length == ref.data.length)
                        && (this.data[0].length == ref.data[0].length)) {
                    equals = true;

outer:
                    for (int r = 0; r < this.data.length; r++) {

                        for (int c = 0; c < this.data[0].length; c++) {

                            if (Math.abs(this.data[r][c] - ref.data[r][c]) > epsilon) {
                                equals = false;

                                break outer;
                            }
                        }
                    }
                } else {
                    equals = false;
                }
            }
        }

        return equals;
    }

    /**
     * Tests whether all entries in the matrix are equal to the entries in a test matrix.
     *
     * @param   ref  the matrix to compare to
     * @return  <code>true</code> if matrix sizes and all entries are equal, <code>false</code>
     *          otherwise
     * @throws  MatrixNotCompleteException  if the matrix has not been completely computed
     */
    public boolean equalsMatrix(final DMatrix ref) throws MatrixNotCompleteException {

        boolean equals;

        if (!isMatrixComplete() || !ref.isMatrixComplete()) {
            throw new MatrixNotCompleteException("Cannot test equality of an incomplete matrix");
        }

        synchronized (this) {

            synchronized (ref) {

                if ((this.data.length == ref.data.length)
                        && (this.data[0].length == ref.data[0].length)) {
                    equals = true;

outer:
                    for (int r = 0; r < this.data.length; r++) {

                        for (int c = 0; c < this.data[0].length; c++) {

                            if (this.data[r][c] != ref.data[r][c]) {
                                equals = false;

                                break outer;
                            }
                        }
                    }
                } else {
                    equals = false;
                }
            }
        }

        return equals;
    }

    /**
     * Tests whether an object is a <code>DMatrix</code> and is equal to this matrix.
     *
     * @param   obj  the object to compare to
     * @return  <code>true</code> if the object is a <code>DMatrix</code> and matrix sizes and all
     *          entries are equal, <code>false</code> otherwise
     */
    @Override public boolean equals(final Object obj) {

        boolean equals;
```

```
    if (obj instanceof DMatrix) {

        try {
            equals = equalsMatrix((DMatrix) obj);
        } catch (MatrixNotCompleteException e) {
            equals = false;
        }
    } else {
        equals = false;
    }

    return equals;
}

/**
 * Computes the hash code of the object.
 *
 * @return  the hash code
 */
@Override public int hashCode() {

    synchronized (this) {
        return this.data.hashCode();
    }
}

/**
 * Saves the matrix, including the completion status of all values, to a file. The in-progress
 * status of values, and any LU decomposition data is not saved with the matrix. The matrix is
 * saved in a format that will allow GnuPlot to display it as a two-dimensional data set using
 * the 'SPLOT' command. Completion status of values is stored in the file as GnuPlot comments.
 *
 * @param   file  the file to which to write
 * @throws  IOException  if there is an error writing to the file
 */
public void save(final File file) throws IOException {

    File tmp;
    FileOutputStream fos;
    PrintStream out;
    int row;
    int col;

    // If there is an existing file, we write this out to a new temporary
    // file, then replace the existing file only on successful write.
    if (file.exists()) {
        tmp = File.createTempFile("DMatrix", "dat", file.getParentFile());
    } else {
        tmp = file;
    }

    fos = new FileOutputStream(tmp);
    out = new PrintStream(fos);

    out.println("# DMatrix v1.0 storage file - do not alter comments");

    out.print("# numRows=");
    out.println(this.data.length);

    out.print("# numCols=");
    out.println(this.data[0].length);

    // Output the completed status of the matrix
    synchronized (this) {

        for (row = 0; row < this.data.length; row++) {
            out.print("# ");
            out.print(row);
            out.print(":");

            for (col = 0; col < this.data[0].length; col++) {
                out.print(HEX[this.flags[row][col] & 0x0F]);
            }

            out.println("");
        }

        // Now output the rows
        for (row = 0; row < this.data.length; row++) {
            out.println("");

            for (col = 0; col < this.data[0].length; col++) {
                out.print(row);
                out.print(' ');
                out.print(col);
                out.print(' ');
                out.println(this.data[row][col]);
```

```
            }
        }
    }

    out.println("# END");

    out.close();
    fos.close();

    if (!tmp.equals(file)) {

        if (file.exists()) {
            LOG.log(Level.FINE, "Deleting {0}", file.getAbsolutePath());

            if (!file.delete()) {
                throw new IOException("Unable to overwrite '" + file.getAbsolutePath() + "'");
            }
        }

        LOG.log(Level.FINE, "Renaming to {0}", file.getAbsolutePath());

        if (!tmp.renameTo(file)) {
            throw new IOException("Unable to rename temp file to '" + file.getAbsolutePath()
                + "'");
        }
    }
}

/**
 * Loads the matrix, including the completion status of all values, from a file.
 *
 * @param   file  the file from which to load
 * @return  the loaded matrix
 * @throws  IOException  if there is an error reading from the file
 */
public static DMatrix load(final File file) throws IOException {

    FileReader reader;
    BufferedReader buf;
    String line;
    int numRows;
    int numCols;
    DMatrix matrix;
    int row;
    int col;
    String expect;
    char[] chars;
    String[] fields;

    reader = new FileReader(file);
    buf = new BufferedReader(reader);

    // Read the first-line comment
    line = buf.readLine();

    if (!line.startsWith("# DMatrix v1.0")) {
        throw new IOException(
            "Invalid matrix file - first line does not begin with '# DMatrix v1.0'.");
    }

    // Read the number of rows and columns
    line = buf.readLine();

    if (!line.startsWith("# numRows=")) {
        throw new IOException(
            "Invalid matrix file - second line does not contain '# numRows=...'");
    }

    numRows = Integer.parseInt(line.substring(10));

    line = buf.readLine();

    if (!line.startsWith("# numCols=")) {
        throw new IOException(
            "Invalid matrix file - third line does not contain '# numCols=...'");
    }

    numCols = Integer.parseInt(line.substring(10));

    matrix = new DMatrix(numRows, numCols);

    // Read in the completed status of the matrix
    for (row = 0; row < matrix.data.length; row++) {
        line = buf.readLine();
        expect = "# " + row + ":";

        if (!line.startsWith(expect)) {
```

```
                throw new IOException("Invlaid matrix file - line " + (3 + row)
                    + " does not contain '" + expect + "'");
            }

            chars = line.substring(expect.length()).toCharArray();

            if (chars.length != matrix.data[0].length) {
                throw new IOException("Invlaid matrix file - line " + (3 + row) + " does not have "
                    + matrix.data[0].length + " flag values");
            }

            for (col = 0; col < matrix.data[0].length; col++) {
                matrix.flags[row][col] = -1;

                for (int i = 0; i < HEX.length; i++) {

                    if (HEX[i] == chars[col]) {
                        matrix.flags[row][col] = (byte) i;

                        break;
                    }
                }

                if (matrix.flags[row][col] == -1) {
                    throw new IOException("Invlaid matrix file - line " + (3 + row)
                        + " has invalid flag value: '" + chars[col] + "'");
                }
            }
        }

        // Read in the matrix data values.
        for (row = 0; row < matrix.data.length; row++) {
            line = buf.readLine();

            if (line.length() > 0) {
                throw new IOException(
                    "Invlaid matrix file - expected blank line before matrix row " + row);
            }

            for (col = 0; col < matrix.data[0].length; col++) {
                line = buf.readLine();
                fields = line.split(" ");

                if (fields.length != 3) {
                    throw new IOException("Invlaid matrix file - matrix row " + row + " column "
                        + col + " does not have 3 fields");
                }

                if (Integer.parseInt(fields[0]) != row) {
                    throw new IOException("Invlaid matrix file - matrix row " + row + " column "
                        + col + " has bad row number");
                }

                if (Integer.parseInt(fields[1]) != col) {
                    throw new IOException("Invlaid matrix file - matrix row " + row + " column "
                        + col + " has bad column number");
                }

                matrix.data[row][col] = Double.parseDouble(fields[2]);
            }
        }

        // Check for proper termination.
        line = buf.readLine();

        if (!line.startsWith("# END")) {
            throw new IOException(
                "Invlaid matrix file - line after matrix does not begin with '# END'.");
        }

        buf.close();
        reader.close();

        // Finally, compute the first uncompleted cell
        matrix.firstUncRow = 0;
        matrix.firstUncCol = 0;
        matrix.advanceFirstUncompleted();

        return matrix;
    }

    /**
     * Generates a list of ranges to use to color map the matrix. Each range will have roughly the
     * same number of matrix values in it, and will be in ascending order by value. If the matrix
     * has many cells with the same data value, it may not be possible to distribute the color
     * ranges evenly. If the matrix is not complete, only the completed data is used. If there are
     * fewer values in the matrix than the number of ranges requested, all ranges after a certain
```

```
     * point will have the same value.
     *
     * @param   numRanges  the number of ranges to generate
     * @return  the color range table - an array of doubles whose values are the boundaries of each
     *          color range; the last entry in the array is at least as large as the largest data
     *          value in the matrix
     */
    public double[] colorRanges(final int numRanges) {

        double[] ranges;
        int total;
        int count;
        double[] values;
        double which;

        ranges = new double[numRanges];

        // We scan the matrix copying all distinct data values into a list.
        synchronized (this) {
            total = 0;

            for (int r = 0; r < this.data.length; r++) {

                for (int c = 0; c < this.data[0].length; c++) {

                    if (isCompleted(r, c)) {
                        total++;
                    }
                }
            }

            values = new double[total];
            count = 0;

            for (int r = 0; r < this.data.length; r++) {

                for (int c = 0; c < this.data[0].length; c++) {

                    if (isCompleted(r, c)) {
                        values[count] = this.data[r][c];
                        count++;
                    }
                }
            }

            // Now we sort the data values into ascending order
            Arrays.sort(values);

            // If there are more ranges (colors) than the length of the values
            // array, we still want to use all the colors, so we just divide
            // the range linearly. Otherwise, we distribute ranges evenly among
            // the values.
            if (numRanges > values.length) {

                for (int i = 0; i < values.length; i++) {
                    which = i * numRanges / values.length;
                    Arrays.fill(ranges, (int) which, numRanges - 1, values[i]);
                }
            } else {

                for (double i = 0; i < numRanges; i++) {
                    which = i * values.length / numRanges;

                    ranges[(int) i] = (which < (values.length - 2)) ? values[(int) which + 1]
                                                                    : values[values.length - 1];
                }
            }
        }

        return ranges;
    }

    /**
     * Main method for testing.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        DMatrix mat;
        DMatrix mat2;
        int[] rowCol;

        mat = new DMatrix(4, 4);
        mat.set(0, 0, 1);
        mat.set(1, 1, Math.PI);
        mat.set(2, 2, Math.E);
```

```
        mat.set(3, 3, Math.sqrt(2));

        try {
            mat.save(new File("/imp/testMatrix.dat"));

            LOG.fine(mat.toString());

            mat2 = load(new File("/imp/testMatrix.dat"));

            LOG.fine(mat2.toString());

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "A: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "B: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "C: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "D: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "E: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "F: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "G: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "H: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "I: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "J: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "K: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.log(Level.FINE, "L: {0}, {1}", new Object[] { rowCol[0], rowCol[1] });

            rowCol = mat2.getUncompletedElement();
            LOG.fine(Boolean.toString(rowCol == null));
        } catch (IOException e) {
            LOG.throwing("DMatrix", "main", e);
        }
    }
}
```

```
package com.srbenoit.math.linear;

/**
 * A labeled bit number used to store flags associated with matrix elements.
 */
public enum MatrixElementFlag {

    /** bit number for a 'completed' flag */
    COMPLETED(0),

    /** bit number for a 'in-progress' flag */
    IN_PROGRESS(1),

    /** bit number for a 'out-of-range' flag */
    OUT_OF_RANGE(2),

    /** bit number for a 'tagged' flag */
    TAGGED(3);

    /** the bit used to store this flag */
    private final int flagBit;

    /**
     * Constructs a new <code>MatrixElementFlag</code>.
     *
     * @param  whichBit  the bit used to store this flag (0-7)
     */
    private MatrixElementFlag(final int whichBit) {

        this.flagBit = whichBit;
    }
```

```
    /**
     * Gets the bit used to store this flag.
     *
     * @return  the bit used to store this flag (0-7)
     */
    public int flagBit() {

        return this.flagBit;
    }

    /**
     * Generates the mask byte (a byte with a single bit set) for a particular flag.
     *
     * @return  the mask byte
     */
    public byte toMask() {

        return (byte) (0x01 << this.flagBit);
    }
}

package com.srbenoit.math.linear;

/**
 * An exception when a matrix that has not been completely evaluated is used.
 */
public class MatrixNotCompleteException extends Exception {

    /** version number for serialization */
    private static final long serialVersionUID = 4180455776728790366L;

    /**
     * Constructs a new <code>MatrixNotCompleteException</code>.
     *
     * @param  message  the exception message.
     */
    public MatrixNotCompleteException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>MatrixNotCompleteException</code>.
     *
     * @param  message  the exception message.
     * @param  cause    the exception that led to this exception.
     */
    public MatrixNotCompleteException(final String message, final Throwable cause) {

        super(message, cause);
    }
}

package com.srbenoit.math.linear;

/**
 * An exception thrown when an attempt is made to invert a singular matrix.
 */
public final class SingularMatrixException extends RuntimeException {

    /** version number for serialization */
    private static final long serialVersionUID = -5321393728885816859L;

    /**
     * Constructs a new <code>SingularMatrixException</code> with <code>null</code> as its detail
     * message.
     */
    public SingularMatrixException() {

        super();
    }

    /**
     * Constructs a new <code>SingularMatrixException</code> with the specified detail message.
     *
     * @param  message  the detail message
     */
    public SingularMatrixException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>SingularMatrixException</code> with the specified detail message and
     * cause.
     *
     * <p>Note that the detail message associated with <code>cause</code> is <i>not</i>
```

```
     * automatically incorporated in this runtime exception's detail message.
     *
     * @param  message  the detail message
     * @param  cause    the cause
     */
    public SingularMatrixException(final String message, final Throwable cause) {

        super(message, cause);
    }

    /**
     * Constructs a new <code>SingularMatrixException</code> with the specified cause and a detail
     * message of <code>(cause==null ? null : cause.toString())</code> (which typically contains the
     * class and detail message of <code>cause</code> ).
     *
     * @param  cause  the cause
     */
    public SingularMatrixException(final Throwable cause) {

        super(cause);
    }
}

package com.srbenoit.math.linear;

/**
 * A tuple characterized by N coordinates.
 */
public class TupleN implements Cloneable {

    /** true if tuple transforms as a point; false to transform as a vector */
    private boolean asPoint;

    /** the coordinate data */
    private double[] data;

    /** the length, computed lazily */
    private transient double len;

    /**
     * Constructs a new <code>TupleN</code> with all three coordinates zero.
     *
     * @param  numCoords  the number of coordinates in the tuple
     * @param  isPoint    <code>true</code> if tuple transforms as a point; <code>false</code> to
     *                    transform as a vector
     */
    public TupleN(final int numCoords, final boolean isPoint) {

        if (numCoords < 1) {
            throw new IllegalArgumentException("Invalid number of coordinates for tuple: "
                + numCoords);
        }

        this.data = new double[numCoords];

        this.asPoint = isPoint;
        this.len = 0f;
    }

    /**
     * Constructs and initializes a <code>TupleN</code> from the specified coordinates.
     *
     * @param  numCoords  the number of coordinates in the tuple
     * @param  isPoint    <code>true</code> if tuple transforms as a point; <code>false</code> to
     *                    transform as a vector
     * @param  values     the initial values for the tuple; if there are fewer values than <code>
     *                    numCoords</code>, then the remaining coordinates after these values have
     *                    been installed are initialized to zero; if there are more values than
     *                    <code>numCoords</code>, the extra values are ignored
     */
    public TupleN(final int numCoords, final boolean isPoint, final double... values) {

        this(numCoords, isPoint);

        int count;

        count = (numCoords < values.length) ? numCoords : values.length;
        System.arraycopy(values, 0, this.data, 0, count);

        this.len = -1;
    }

    /**
     * Constructs and initializes a <code>TupleN</code> from the specified <code>TupleN</code>.
     *
     * @param  tuple  the <code>TupleN</code> containing the initialization data
     */
```

```
public TupleN(final TupleN tuple) {

    this(tuple.getDimension(), tuple.isAsPoint());

    this.data = tuple.get();
    this.len = tuple.getCurLength();
}

/**
 * Gets the number of coordinates in the tuple.
 *
 * @return  the number of coordinates
 */
public int getDimension() {

    return this.data.length;
}

/**
 * Checks that the dimension of this tuple matches that of another tuple, and returns that
 * dimension if so.
 *
 * @param   other  the <code>TupleN</code>to check
 * @return  the dimension of the tuples
 * @throws  IllegalArgumentException  if dimensions fail to match
 */
public int dimCheck(final TupleN other) {

    if (getDimension() != other.getDimension()) {
        throw new IllegalArgumentException("Dimension mismatch");
    }

    return getDimension();
}

/**
 * Test whether the tuple transforms as a point or as a vector.
 *
 * @return  <code>true</code> if tuple transforms as a point; <code>false</code> to transform
 *          as a vector
 */
public boolean isAsPoint() {

    return this.asPoint;
}

/**
 * Sets the flag that determines whether the tuple transforms as a point or as a vector.
 *
 * @param  isPoint  <code>true</code> if tuple transforms as a point; <code>false</code> to
 *                  transform as a vector
 */
public void setAsPoint(final boolean isPoint) {

    this.asPoint = isPoint;
}

/**
 * Gets a coordinate value.
 *
 * @param   index  the index of the coordinate value to retrieve
 * @return  the coordinate
 */
public double get(final int index) {

    return this.data[index];
}

/**
 * Gets all coordinate values.
 *
 * @return  the coordinates
 */
public double[] get() {

    return this.data.clone();
}

/**
 * Sets a coordinate value.
 *
 * @param  index  the index of the coordinate value to set
 * @param  coord  the new coordinate value
 */
public void set(final int index, final double coord) {

    this.data[index] = coord;
```

```
}

/**
 * Sets the coordinate values. This can change the dimension of the tuple.
 *
 * @param  coords  the new coordinate values
 */
public void set(final double[] coords) {

    this.data = coords.clone();
    this.len = -1;
}

/**
 * Returns a string that contains the values of this <code>TupleN</code>. The form is (p1, p2,
 * ..., pN) for points, [p1, p2, ..., pN] for vectors.
 *
 * @return  the <code>String</code> representation
 */
@Override public String toString() {

    StringBuilder str;

    str = new StringBuilder(20);
    str.append(this.asPoint ? '(' : '[');

    for (int i = 0; i < getDimension(); i++) {

        if (i > 0) {
            str.append(", ");
        }

        str.append(this.data[i]);
    }

    str.append(this.asPoint ? ')' : ']');

    return str.toString();
}

/**
 * Makes a copy of the tuple.
 *
 * @return  the copy
 */
public TupleN copy() {

    TupleN obj;

    try {
        obj = (TupleN) clone();
        obj.set(this.data);
    } catch (CloneNotSupportedException e) {
        obj = new TupleN(this);
    }

    return obj;
}

/**
 * Generates a hash code for the object.
 *
 * @return  the hash code
 */
@Override public int hashCode() {

    int hash = 0;

    for (int i = 0; i < getDimension(); i++) {
        hash += (int) this.data[i];
    }

    return hash;
}

/**
 * Tests whether this object is equal to another object. To be equal, the other object must
 * also be a <code>TupleN</code> and must have the same number of coordinates, the same
 * coordinate values, and the same <code>isPoint</code> value.
 *
 * <p>The lazily computed length value is not used in the comparison.
 *
 * @param   obj  the object to test for equality
 * @return  <code>true</code> of the objects are equal; <code>false</code> if not
 */
@Override public boolean equals(final Object obj) {
```

```
        TupleN other;
        boolean equal;

        if (obj instanceof TupleN) {
            other = (TupleN) obj;

            if ((other.isAsPoint() == isAsPoint()) && (other.getDimension() == getDimension())) {
                equal = true;

                for (int i = 0; i < getDimension(); i++) {

                    if (other.get(i) != get(i)) {
                        equal = false;

                        break;
                    }
                }
            } else {
                equal = false;
            }
        } else {
            equal = false;
        }

        return equal;
    }

    /**
     * Adds a scaled version of a tuple to this tuple (this = this + scale * tuple).
     *
     * @param  scale  the scalar value
     * @param  tuple  the tuple to be scaled then added
     */
    public void addScaled(final double scale, final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException(
                "addScaled: Tuples being added must be same dimension");
        }

        for (int i = 0; i < getDimension(); i++) {
            set(i, get(i) + (scale * tuple.get(i)));
        }

        this.len = -1.0;

        // Vector + vector = vector
        // Vector + point = point
        // Point + vector = point
        // Point + point = point
        this.asPoint |= tuple.isAsPoint();
    }

    /**
     * Adds a tuple offset to the tuple.
     *
     * @param  tuple  the tuple offset
     */
    public void add(final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException("add: Tuples being added must be same dimension");
        }

        for (int i = 0; i < getDimension(); i++) {
            set(i, get(i) + tuple.get(i));
        }

        this.len = -1.0;

        // Vector + vector = vector
        // Vector + point = point
        // Point + vector = point
        // Point + point = point
        this.asPoint |= tuple.isAsPoint();
    }

    /**
     * Sets the value of this <code>TupleN</code> to the sum of <code>tuple1</code> and <code>
     * tuple2</code> (this = tuple1 + tuple2).
     *
     * @param  tuple1  the first tuple
     * @param  tuple2  the second tuple
     */
    public void add(final TupleN tuple1, final TupleN tuple2) {

        if (tuple1.getDimension() != tuple2.getDimension()) {
```

```
            throw new IllegalArgumentException("add: Tuples being added must be same dimension");
        }

        if (this.getDimension() != tuple1.getDimension()) {
            throw new IllegalArgumentException(
                "add: Tuples being added must be same dimension as destination tuple");
        }

        for (int i = 0; i < tuple1.getDimension(); i++) {
            set(i, tuple1.get(i) + tuple2.get(i));
        }

        this.len = -1.0;

        // Vector + vector = vector
        // Vector + point = point
        // Point + vector = point
        // Point + point = point
        this.asPoint = tuple1.isAsPoint() | tuple2.isAsPoint();
    }

    /**
     * Subtracts a tuple offset from the tuple.
     *
     * @param  tuple  the tuple offset
     */
    public void sub(final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException(
                "sub: Tuples being subtracted must be same dimension");
        }

        for (int i = 0; i < getDimension(); i++) {
            set(i, get(i) - tuple.get(i));
        }

        this.len = -1.0;

        // Vector - vector = vector
        // Vector - point = point
        // Point - vector = point
        // Point - point = vector
        this.asPoint = isAsPoint() != tuple.isAsPoint();
    }

    /**
     * Sets the value of this <code>TupleN</code> to the difference of <code>tuple1</code> and
     * <code>tuple2</code> (this = tuple1 - tuple2).
     *
     * @param  tuple1  the first tuple
     * @param  tuple2  the second tuple
     */
    public void sub(final TupleN tuple1, final TupleN tuple2) {

        if (tuple1.getDimension() != tuple2.getDimension()) {
            throw new IllegalArgumentException(
                "sub: Tuples being subtracted must be same dimension");
        }

        if (getDimension() != tuple1.getDimension()) {
            throw new IllegalArgumentException(
                "sub: Tuples being subtracted must be same dimension as destination tuple");
        }

        for (int i = 0; i < tuple1.getDimension(); i++) {
            set(i, tuple1.get(i) - tuple2.get(i));
        }

        this.len = -1.0;

        // Vector - vector = vector
        // Vector - point = point
        // Point - vector = point
        // Point - point = vector
        this.asPoint = tuple1.isAsPoint() != tuple2.isAsPoint();
    }

    /**
     * Computes the square of the Euclidean distance between this tuple and <code>tuple</code>.
     *
     * @param   tuple  the other tuple
     * @return  the square of the distance
     */
    public double distSquared(final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
```

```
            throw new IllegalArgumentException(
                "distSquared: Tuples being compared must be same dimension");
        }

        double delta;
        double sum;

        sum = 0;

        for (int i = 0; i < getDimension(); i++) {
            delta = get(i) - tuple.get(i);
            sum += delta * delta;
        }

        return sum;
    }

    /**
     * Computes the Euclidean distance between this tuple and <code>tuple</code>.
     *
     * @param   tuple  the other tuple
     * @return  the distance
     */
    public double dist(final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException(
                "dist: Tuples being compared must be same dimension");
        }

        double delta;
        double sum;

        sum = 0;

        for (int i = 0; i < getDimension(); i++) {
            delta = get(i) - tuple.get(i);
            sum += delta * delta;
        }

        return Math.sqrt(sum);
    }

    /**
     * Negates this <code>TupleN</code> in place.
     */
    public void negate() {

        for (int i = 0; i < getDimension(); i++) {
            set(i, -get(i));
        }
    }

    /**
     * Sets this <code>TupleN</code> to the scalar multiplication of the scale factor with this.
     *
     * @param  scale  the scalar value
     */
    public void scale(final double scale) {

        for (int i = 0; i < getDimension(); i++) {
            this.data[i] *= scale;
        }

        if (this.len > 0) {

            if (scale > 0) {
                this.len *= scale;
            } else {
                this.len *= -scale;
            }
        }
    }

    /**
     * Sets this <code>TupleN</code> to the scalar multiplication of <code>tuple</code>.
     *
     * @param  scale  the scalar value
     * @param  tuple  the source tuple
     */
    public void scale(final double scale, final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException(
                "scale: Tuple being scaled must be same dimension as destination tuple");
        }
```

```
        for (int i = 0; i < getDimension(); i++) {
            set(i, scale * tuple.get(i));
        }

        if (tuple.getCurLength() > 0) {

            if (scale > 0) {
                this.len = tuple.getCurLength() * scale;
            } else {
                this.len = -tuple.getCurLength() * scale;
            }
        } else {
            this.len = -1.0;
        }

        this.asPoint = tuple.isAsPoint();
    }

    /**
     * Sets this <code>TupleN</code> to the scalar multiplication of itself and then adds <code>
     * tuple</code> (this = scale * this + tuple).
     *
     * @param  scale  the scalar value
     * @param  tuple  the tuple to be added
     */
    public void scaleAdd(final double scale, final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException(
                "scaleAdd: Tuples being added must be same dimension");
        }

        for (int i = 0; i < getDimension(); i++) {
            set(i, (scale * get(i)) + tuple.get(i));
        }

        this.len = -1.0;

        // Vector + vector = vector
        // Vector + point = point
        // Point + vector = point
        // Point + point = point
        this.asPoint |= tuple.isAsPoint();
    }

    /**
     * Sets this <code>TupleN</code> to the scalar multiplication of <code>tuple1</code> and then
     * adds <code>tuple2</code> (this = scale * tuple1 + tuple2).
     *
     * @param  scale   the scalar value
     * @param  tuple1  the tuple to be scaled and added
     * @param  tuple2  the tuple to be added without a scale
     */
    public void scaleAdd(final double scale, final TupleN tuple1, final TupleN tuple2) {

        if (tuple1.getDimension() != tuple2.getDimension()) {
            throw new IllegalArgumentException(
                "scaleAdd: Tuples being added must be same dimension");
        }

        if (this.getDimension() != tuple1.getDimension()) {
            throw new IllegalArgumentException(
                "scaleAdd: Tuples being added must be same dimension as destination tuple");
        }

        for (int i = 0; i < getDimension(); i++) {
            set(i, (scale * tuple1.get(i)) + tuple2.get(i));
        }

        this.len = -1.0;

        // Vector + vector = vector
        // Vector + point = point
        // Point + vector = point
        // Point + point = point
        this.asPoint = tuple1.isAsPoint() | tuple2.isAsPoint();
    }

    /**
     * Returns the squared length of the tuple.
     *
     * @return  the squared length of the tuple
     */
    public double lengthSquared() {

        double sum;
```

```
        if (this.len < 0) {
            sum = 0;

            for (int i = 0; i < getDimension(); i++) {
                sum += this.data[i] * this.data[i];
            }
        } else {
            sum = this.len * this.len;
        }

        return sum;
    }

    /**
     * Returns the length of the tuple.
     *
     * @return  the length of the tuple
     */
    public double length() {

        double sum;

        if (this.len < 0) {
            sum = 0;

            for (int i = 0; i < getDimension(); i++) {
                sum += this.data[i] * this.data[i];
            }

            this.len = Math.sqrt(sum);
        }

        return this.len;
    }

    /**
     * Returns the current length of the tuple.
     *
     * @return  the current length, or -1 if the length has not yet been computed
     */
    public double getCurLength() {

        return this.len;
    }

    /**
     * Computes the dot product of this <code>TupleN</code> and <code>tuple</code>.
     *
     * @param   tuple  the other tuple
     * @return  the dot product
     */
    public double dot(final TupleN tuple) {

        double prod;

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException(
                "dot: Tuples in dot product must be same dimension");
        }

        prod = 0;

        for (int i = 0; i < getDimension(); i++) {
            prod += get(i) * tuple.get(i);
        }

        return prod;
    }

    /**
     * Computes the angle between this tuple and another tuple (where both are considered as
     * vectors).
     *
     * @param   tuple  The other tuple.
     * @return  the angle between this tuple and <code>tuple</code>, in radians
     */
    public double angle(final TupleN tuple) {

        return Math.acos(this.dot(tuple) / (this.length() * tuple.length()));
    }

    /**
     * Normalizes this <code>TupleN</code> in place.
     */
    public void normalize() {

        double length;
```

```
        double recip;

        length = length();

        if (length == 0) {
            this.data[0] = 1;

            for (int i = 1; i < getDimension(); i++) {
                this.data[i] = 0;
            }
        } else if (length != 1.0) {
            recip = 1.0 / length;

            for (int i = 1; i < getDimension(); i++) {
                this.data[i] *= recip;
            }
        }

        this.len = 1.0;
    }

    /**
     * Sets this <code>TupleN</code> to the normalization of another tuple.
     *
     * @param  tuple  the tuple to normalize
     */
    public void normalize(final TupleN tuple) {

        if (getDimension() != tuple.getDimension()) {
            throw new IllegalArgumentException(
                "Tuple being normalized must be same dimension as destination tuple");
        }

        double length;
        double recip;

        length = tuple.length();

        if (length == 0) {
            this.data[0] = 1;

            for (int i = 1; i < getDimension(); i++) {
                this.data[i] = 0;
            }
        } else if (length != 1.0) {
            recip = 1.0 / length;

            for (int i = 1; i < getDimension(); i++) {
                set(i, tuple.get(i) * recip);
            }
        }

        this.len = 1.0;
    }
}
```

E.4.4 Solvers (com.srbenoit.math.solvers)

This package supports solving equations of systems of equations.

```
package com.srbenoit.math.solvers;

import com.srbenoit.log.LoggedObject;
import com.srbenoit.math.linear.TupleN;

/**
 * A base class for functions.
 */
public abstract class AbstractFunction extends LoggedObject {

    /**
     * Evaluate the function at a particular set of coordinates.
     *
     * @param   system       the system of functions to which the function belongs
     * @param   coordinates  the coordinates
     * @return  the value of the function at those coordinates
     */
    public abstract double evaluate(SystemOfFunctions system, TupleN coordinates);

    /**
```

```
     * Evaluate the partial derivative of the function with respect to a particular coordinate at a
     * particular set of coordinates.
     *
     * @param   system        the system of functions to which the function belongs
     * @param   coordinates   the coordinates
     * @param   index         the index of the coordinate that we want the derivative with respect to
     * @return  the value of the function at those coordinates
     */
    public abstract double derivative(SystemOfFunctions system, TupleN coordinates, int index);
}

package com.srbenoit.math.solvers;

import com.srbenoit.math.linear.DMatrix;
import com.srbenoit.math.linear.MatrixNotCompleteException;
import com.srbenoit.math.linear.SingularMatrixException;
import com.srbenoit.math.linear.TupleN;
import com.srbenoit.util.EvaluationException;

/**
 * This solver can take a system of nonlinear equations and find local solutions given a starting
 * guess, using the Newton-Raphson method. We are given a set of functions that take a vector of
 * values to generate an output. We seek to find the set of values such that all functions yield
 * zero.
 */
public class NewtonRaphsonMethod {

    /** tolerance used to detect when the solver has converged */
    public static final double TOLERANCE = 1E-16;

    /** tolerance used to detect when changes have grown too small */
    public static final double DELTA_TOLERANCE = 1E-100;

    /** the list of functions to be solved */
    private final transient SystemOfFunctions functions;

    /**
     * Constructs a new <code>NewtonRaphsonMethod</code>.
     *
     * @param  functionsToSolve  the list of functions to be solved.\
     */
    public NewtonRaphsonMethod(final SystemOfFunctions functionsToSolve) {

        this.functions = functionsToSolve;
    }

    /**
     * Finds a solution based on an initial guess.
     *
     * @param   maxIterations  the maximum number of iterations
     * @param   initialGuess   the initial guess
     * @return  the solution, if found
     * @throws  SolverFailedException  if the solver failed to find a solution
     */
    public TupleN solve(final int maxIterations, final TupleN initialGuess)
        throws SolverFailedException {

        TupleN coords;
        double err;
        TupleN rhs;
        DMatrix jacobian;
        DMatrix inverse;
        TupleN deltas;

        coords = initialGuess.copy();
        rhs = new TupleN(initialGuess.getDimension(), false);
        jacobian = new DMatrix(initialGuess.getDimension(), initialGuess.getDimension());

        for (int iter = 0; iter < maxIterations; iter++) {

            // Do any pre-computation based on current coordinates
            this.functions.precompute(coords);

            // See if we are done based on error being below tolerance.
            err = computeError(coords);

            if (err < TOLERANCE) {
                break; // Solver has finished.
            }

            // Compute the approximate Jacobian at the current point
            computeJacobian(coords, jacobian);

            // Invert the Jacobian matrix
            try {
                inverse = jacobian.inverse();
            } catch (EvaluationException e1) {
```

```
            throw new SolverFailedException("Unable to invert matrix", e1);
        } catch (SingularMatrixException e2) {
            throw new SolverFailedException("Unable to invert matrix", e2);
        } catch (MatrixNotCompleteException e2) {
            throw new SolverFailedException("Unable to invert matrix", e2);
        }

        // Set up right-hand side of dX = - J^-1 dot F(X)
        for (int n = 0; n < coords.getDimension(); n++) {
            rhs.set(n, -this.functions.getFunction(n).evaluate(this.functions, coords));
        }

        // Compute dX
        try {
            deltas = inverse.transform(rhs);
        } catch (EvaluationException e1) {
            throw new SolverFailedException("Unable to compute deltas", e1);
        }

        // Shrink deltas to avoid overshoot, check for convergence based on
        // Deltas being to small, and at the same time, update the values
        // based on the deltas.
        err = 0;

        for (int i = 0; i < coords.getDimension(); i++) {
            deltas.set(i, deltas.get(i) * 0.1);
            err += Math.abs(deltas.get(i));
            coords.set(i, coords.get(i) + deltas.get(i));
        }

        if (err < DELTA_TOLERANCE) {
            break; // Solver has finished.
        }
    }

    return coords;
}

/**
 * Computes the error at the current set of coordinates.
 *
 * @param   coords  the coordinates
 * @return  the error
 * @throws  SolverFailedException  if the error cannot be evaluated or is infinite
 */
private double computeError(final TupleN coords) throws SolverFailedException {

    double err;

    err = 0;

    for (int i = 0; i < coords.getDimension(); i++) {
        err += Math.abs(this.functions.getFunction(i).evaluate(this.functions, coords));
    }

    if (Double.isNaN(err)) {
        throw new SolverFailedException("Failed to evaluate");
    }

    if (Double.isInfinite(err)) {
        throw new SolverFailedException("Error is infinite");
    }

    return err;
}

/**
 * Computes the Jacobian matrix of the functions at the current point.
 *
 * @param  coordinates  the point at which to evaluate the Jacobian
 * @param  jacobian     the resulting Jacobian
 */
private void computeJacobian(final TupleN coordinates, final DMatrix jacobian) {

    AbstractFunction func;
    double partial;

    // Go through each function, get its value at the current point
    for (int i = 0; i < this.functions.numFunctions(); i++) {
        func = this.functions.getFunction(i);

        // Now loop through each value and take the partial derivative in
        // that value's direction
        for (int j = 0; j < coordinates.getDimension(); j++) {

            partial = func.derivative(this.functions, coordinates, j);
            jacobian.set(i, j, partial);
```

```
            }
        }
    }
}

package com.srbenoit.math.solvers;

/**
 * An exception thrown when a solver fails to find a solution.
 */
public class SolverFailedException extends Exception {

    /** version number for serialization */
    private static final long serialVersionUID = 587807358367525183L;

    /**
     * Constructs a new <code>SolverFailedException</code>.
     *
     * @param  message  the message
     */
    public SolverFailedException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>SolverFailedException</code>.
     *
     * @param  message  the message
     * @param  cause    the exception that caused the solver to fail
     */
    public SolverFailedException(final String message, final Throwable cause) {

        super(message, cause);
    }
}

package com.srbenoit.math.solvers;

import com.srbenoit.math.linear.TupleN;

/**
 * A system of functions to be evaluated.
 */
public class SystemOfFunctions {

    /** the functions. */
    private transient AbstractFunction[] functions;

    /**
     * Constructs a new <code>SystemOfFunctions</code>.
     *
     * @param  numFunctions  the number of functions in the system
     */
    public SystemOfFunctions(final int numFunctions) {

        this.functions = new AbstractFunction[numFunctions];
    }

    /**
     * Get the number of functions in the list.
     *
     * @return  the number of functions
     */
    public final int numFunctions() {

        return this.functions.length;
    }

    /**
     * Install a function in the system.
     *
     * @param  index     the index of the function
     * @param  function  the function
     */
    public final void setFunction(final int index, final AbstractFunction function) {

        this.functions[index] = function;
    }

    /**
     * Get a particular function.
     *
     * @param   index  the index of the function to get
     * @return  the function
     */
    public final AbstractFunction getFunction(final int index) {
```

```
        return this.functions[index];
    }

    /**
     * Pre-computes values used by all functions for one particular set of coordinates. This
     * implementation does nothing. Subclasses can override to perform problem-specific
     * pre-computation.
     *
     * @param  coordinates  the coordinates at which to compute values
     */
    public void precompute(@SuppressWarnings("unused") final TupleN coordinates) {

        // Empty
    }
}
```

E.4.5 Optimization (com.srbenoit.math.optimizers)

This package collects many one-off general utilities to simplify common functions or provide functionality missing in the JDK.

```
package com.srbenoit.math.optimizers;

/**
 * An exception that indicates the optimizer failed to converge.
 */
public class FailedToConvergeException extends Exception {

    /** version number for serialization */
    private static final long serialVersionUID = -3879758989507065943L;

    /**
     * Constructs a new <code>FailedToConvergeException</code> with the specified detail message.
     *
     * @param  message  the detail message
     */
    public FailedToConvergeException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>FailedToConvergeException</code> with the specified detail message and
     * cause.
     *
     * @param  message  the detail message
     * @param  cause    the cause
     */
    public FailedToConvergeException(final String message, final Throwable cause) {

        super(message, cause);
    }
}
```

```
package com.srbenoit.math.optimizers;

/**
 * An interface for classes that represent optimizable functions.
 */
public interface Optimizable {

    /**
     * Gets the dimension of the function.
     *
     * @return  the dimension
     */
    int dimension();

    /**
     * Evaluates the function for a given set of parameters that are to be optimized.
     *
     * @param   parameters  the parameters
     * @return  the function value based on the parameters
     */
    OutputValue evaluate(double[] parameters);
```

```
    /**
     * Called when the optimizer has found a new set of parameters that result in a lower value
     * during its search for the optimal parameters.
     *
     * @param  parameters  the newly found parameters
     */
    void updatedParameters(double[] parameters);
}

package com.srbenoit.math.optimizers;

import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * An optimizer based on the Nelder-Mead Simplex method to locate a local minimum in a function.
 */
public class Optimizer extends LoggedObject {

    /** maximum allowed iterations of the algorithm */
    public static final int MAXITERATIONS = 100000;

    /** fractional change in the evaluated function to stop algorithm */
    public static final double TOLERANCE = 1E-14;

    /** the optimizable function to be optimized */
    private final transient Optimizable function;

    /** the dimension of the optimizable function */
    private final transient int dimension;

    /** the scale in each dimension of the original simplex */
    private final transient double[] scale;

    /** preallocated double array for a point being tested */
    private final transient double[] pointToTry;

    /** flag indicating optimizer should favor integers in its output */
    private final transient boolean favorInts;

    /**
     * Constructs a new <code>Optimizer</code>.
     *
     * @param  theFunction     the optimizable function to be optimized
     * @param  theScale        the approximate scale of the function - a step size over which the
     *                         function will vary appreciably
     * @param  favorIntegers   if <code>true</code>, the optimizer will check the optimal solution
     *                         and if any values are very close to integers, it will check the
     *                         function value at the integer and if the same as the minimum, it will
     *                         return the integer; if <code>false</code>, it will return the optimum
     *                         is initially found
     */
    public Optimizer(final Optimizable theFunction, final double[] theScale,
        final boolean favorIntegers) {

        this.function = theFunction;
        this.dimension = theFunction.dimension();
        this.scale = theScale.clone();
        this.pointToTry = new double[theFunction.dimension()];
        this.favorInts = favorIntegers;
    }

    /**
     * Generates optimal parameter values.
     *
     * @param   logLevel  the level of logging to do
     * @param   guess     the point at which to start optimization
     * @return  an array of doubles with the best-fit values for the parameters
     * @throws  FailedToConvergeException  if the optimizer failed to converge
     */
    public double[] optimize(final double[] guess, final Level logLevel)
        throws FailedToConvergeException {

        double[][] simplex;
        OutputValue value;
        double[] values;

        LOG.setLevel(logLevel);

        LOG.log(Level.INFO, "Optimizing in dimension {0}", this.dimension);

        simplex = new double[this.dimension + 1][this.dimension];
        values = new double[this.dimension + 1];

        // Establish an initial simplex
        for (int i = 0; i <= this.dimension; i++) {
            System.arraycopy(guess, 0, simplex[i], 0, this.dimension);
```

```
        }

        for (int j = 0; j < this.dimension; j++) {
            simplex[j + 1][j] += this.scale[j];
        }

        // Compute the values at simplex vertices.
        for (int i = 0; i <= this.dimension; i++) {
            value = this.function.evaluate(simplex[i]);
            values[i] = value.getValue();
        }

        // Now, run the simplex algorithm...
        if (!simplex(simplex, values)) {
            throw new FailedToConvergeException("Simplex Optimizer failed to converge");
        }

        // The result is now the first simplex vertex
        return simplex[0];
    }

    /**
     * Performs simplex optimization.
     *
     * @param   simplex  on input, the starting simplex; on output, all points of the simplex will
     *                   be within TOLERANCE of the local minimum found
     * @param   values   on input, the initial values at the vertices of the simplex; on output,
     *                   the values at the new vertices
     * @return  <code>true</code> if algorithm converged on a solution before reaching maximum
     *          iterations; <code>false</code> if not
     */
    private boolean simplex(final double[][] simplex, final double[] values) {

        boolean converged;
        int ihi, ilo, inhi, mpts;
        double sum;
        double swap;
        double[] swapvertex;
        double ysave;
        double ytry;
        double rtol;
        double[] infNorms;
        double[] tryInts;
        int loops = 0;
        double ceil;
        double floor;
        OutputValue value;
        StringBuilder str;

        converged = false;
        infNorms = new double[this.dimension];

        mpts = this.dimension + 1;

        // Load partial sums (this is the sum of the infinity norms of all
        // vectors in the simplex).
        for (int j = 0; j < this.dimension; j++) {
            sum = 0.0;

            for (int i = 0; i < mpts; i++) {
                sum += simplex[i][j];
            }

            infNorms[j] = sum;
        }

        str = new StringBuilder(200);

        for (;;) {

            // First, find the highest, next highest, and lowest vertices.
            ilo = 0;

            if (values[0] > values[1]) {
                ihi = 0;
                inhi = 1;
            } else {
                ihi = 1;
                inhi = 0;
            }

            for (int i = 0; i < mpts; i++) {

                if (values[i] <= values[ilo]) {
                    ilo = i;
                }
```

```
        if (values[i] > values[ihi]) {
            inhi = ihi;
            ihi = i;
        } else if ((values[i] > values[inhi]) && (i != inhi)) {
            inhi = i;
        }
    }

    // Compute tolerance between values, see if we're done.
    if ((values[ihi] - values[ilo]) == 0) {
        rtol = 0;
    } else {
        rtol = 2.0 * Math.abs(values[ihi] - values[ilo])
            / (Math.abs(values[ihi]) + Math.abs(values[ilo]));
    }

    if (loops > (MAXITERATIONS - 10)) {
        str.setLength(0);
        str.append("FAILING: lo=");
        str.append(values[ilo]);
        str.append(", hi=");
        str.append(values[ihi]);
        str.append(", tol=");
        str.append(rtol);
        LOG.warning(str.toString());
    }

    if (rtol < TOLERANCE) {

        // Put the lowest value in values[0];
        swap = values[0];
        values[0] = values[ilo];
        values[ilo] = swap;

        // Put the vertex with the lowest error in simplex[0]
        swapvertex = simplex[0];
        simplex[0] = simplex[ilo];
        simplex[ilo] = swapvertex;

        converged = true;
        LOG.log(Level.INFO, "converged after {0} iterations: {1}",
            new Object[] { loops, rtol });

        break;
    }

    // If we've looped too many times, bail out.
    if (loops > MAXITERATIONS) {
        LOG.log(Level.WARNING,
            "Simplex optimizer exceeded {0} iterations - failed to converge",
            MAXITERATIONS);

        break;
    }

    loops += 2;

    // Extrapolate the simplex by a factor of -1 through the face of
    // the simplex across from the high point.
    ytry = attempt(simplex, values, infNorms, ihi, -1.0);

    if (ytry <= values[ilo]) {

        // This is better than our best point, so try increasing
        // extrapolation to a factor of -2
        ytry = attempt(simplex, values, infNorms, ihi, 2.0);

        this.function.updatedParameters(simplex[ilo]);
    } else if (ytry >= values[inhi]) {

        // reflected point is worse than the second-highest, so look
        // for an intermediate lower point (contract simplex)
        ysave = values[ihi];
        ytry = attempt(simplex, values, infNorms, ihi, 0.5);

        if (ytry >= ysave) {

            // Can't get rid of high point, better contract around
            // the lowest point
            for (int i = 0; i < mpts; i++) {

                if (i != ilo) {

                    for (int j = 0; j < this.dimension; j++) {
                        simplex[i][j] = 0.5 * (simplex[i][j] + simplex[ilo][j]);
                    }
```

```
                        value = this.function.evaluate(simplex[i]);
                        values[i] = value.getValue();
                    }
                }

                loops++;

                // Recompute infinity norms
                for (int j = 0; j < this.dimension; j++) {
                    sum = 0.0;

                    for (int i = 0; i < mpts; i++) {
                        sum += simplex[i][j];
                    }

                    infNorms[j] = sum;
                }
            }
        } else {
            loops--;
        }
    }

    // One last thing - if any of the values are very near an integer, we
    // try adjusting them to the integer value and evaluating. That way
    // we can find integer minima exactly.
    if (converged && this.favorInts) {
        tryInts = simplex[0].clone();
        value = this.function.evaluate(tryInts);
        ysave = value.getValue();

        for (int i = 0; i < tryInts.length; i++) {
            ceil = Math.ceil(tryInts[i]);
            floor = Math.floor(tryInts[i]);

            if ((ceil - tryInts[i]) < 1E-4) {
                tryInts[i] = ceil;
            } else if ((tryInts[i] - floor) < 1E-4) {
                tryInts[i] = floor;
            }

            // Get rid of weird "negative zero" value.
            if (tryInts[i] == -0.0) {
                tryInts[i] = 0.0;
            }
        }

        value = this.function.evaluate(tryInts);
        ytry = value.getValue();

        if (ytry <= ysave) {
            System.arraycopy(tryInts, 0, simplex[0], 0, tryInts.length);
        }
    }

    return converged;
}

/**
 * Extrapolates through the face of the simplex across from the high point, checking the value
 * there, and replacing the high point if the new value is lower.
 *
 * @param   simplex     on input, the starting simplex; on output, the adjusted simplex
 * @param   values      on input, the initial values at the vertices of the simplex; on output,
 *                      the values at the new vertices
 * @param   infNorms    the infinity norms of the vertex points
 * @param   highIndex   the index of the vertex with the highest node
 * @param   fac         the factor by which to extrapolate
 * @return  the function value at the trial point
 */
private double attempt(final double[][] simplex, final double[] values,
    final double[] infNorms, final int highIndex, final double fac) {

    double fac1;
    double fac2;
    double result;
    OutputValue value;

    fac1 = (1 - fac) / this.dimension;
    fac2 = fac1 - fac;

    for (int j = 0; j < this.dimension; j++) {
        this.pointToTry[j] = (infNorms[j] * fac1) - (simplex[highIndex][j] * fac2);
    }

    // Evaluate the function at the new point.
    value = this.function.evaluate(this.pointToTry);
```

```
            result = value.getValue();

            // If the value is smaller, move the simplex.
            if (result < values[highIndex]) {
                values[highIndex] = result;

                for (int j = 0; j < this.dimension; j++) {
                    infNorms[j] += this.pointToTry[j] - simplex[highIndex][j];
                }

                System.arraycopy(this.pointToTry, 0, simplex[highIndex], 0, this.dimension);
            }

            return result;
        }
    }

package com.srbenoit.math.optimizers;

/**
 * The computed value of an optimizable object, along with a flag indicating whether the parameters
 * were in range or not.
 */
public class OutputValue {

    /** the computed value */
    private double value = 0;

    /** true if the parameters were out of range; false if not */
    private boolean outOfRange = false;

    /**
     * Gets the current value.
     *
     * @return  the value
     */
    public double getValue() {

        return this.value;
    }

    /**
     * Sets the current value.
     *
     * @param  newValue  the new value
     */
    public void setValue(final double newValue) {

        this.value = newValue;
    }

    /**
     * Tests whether or not the value is out of range.
     *
     * @return  <code>true</code> if the value is out of range; <code>false</code> otherwise
     */
    public boolean isOutOfRange() {

        return this.outOfRange;
    }

    /**
     * Sets the flag that indicates whether or not the value is out of range.
     *
     * @param  outOfRange  <code>true</code> if the value is out of range; <code>false</code>
     *                     otherwise
     */
    public void setOutOfRange(final boolean outOfRange) {

        this.outOfRange = outOfRange;
    }

    /**
     * Generates the string representation of the value.
     *
     * @return  the <code>String</code>
     */
    @Override public String toString() {

        return Double.toString(this.value);
    }
}

package com.srbenoit.math.optimizers;

/**
 * A point in an N-dimensional space along with the value of a function at that point.
```

```
 */
public class PointAndValue {

    /** the point coordinates */
    private final transient double[] point;

    /** the point value */
    private final transient double value;

    /**
     * Constructs a new <code>PointAndValue</code>.
     *
     * @param  coordinates    the point coordinates
     * @param  functionValue  the function value
     */
    public PointAndValue(final double[] coordinates, final double functionValue) {

        this.point = coordinates.clone();
        this.value = functionValue;
    }

    /**
     * Get the point coordinates as an array.
     *
     * @return  the point coordinates
     */
    public double[] getPoint() {

        return this.point.clone();
    }

    /**
     * Get the value at the point.
     *
     * @return  the value
     */
    public double getValue() {

        return this.value;
    }

    /**
     * Generates the string representation of the point.
     *
     * @return  the <code>String</code>
     */
    @Override public String toString() {

        StringBuilder str;

        str = new StringBuilder(40);

        str.append('(');

        for (int i = 0; i < this.point.length; i++) {

            if (i > 0) {
                str.append(',');
            }

            str.append(Double.toString(this.point[i]));
        }

        str.append(") --> ");
        str.append(Double.toString(this.value));

        return str.toString();
    }
}

package com.srbenoit.math.optimizers;

/**
 * An interface for classes that represent functions that can be optimized over a region.
 */
public interface RegionalOptimizable extends Optimizable {

    /**
     * Called by the regional optimizer when it locates a regional minimum.
     *
     * @param  parameters  the parameters at the minimum
     */
    void foundAMinium(double[] parameters);
}

package com.srbenoit.math.optimizers;
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * An optimizer that scans a region of solution space for local minima by partitioning the space,
 * evaluating the function being minimized on grid points, and initializing a minimizer at each
 * grid point that has a smaller value than all its neighbors.
 */
public class RegionalOptimizer extends LoggedObject {

    /** the optimizable function to be optimized */
    private final transient RegionalOptimizable function;

    /** the range being evaluated in each dimension */
    private final transient SearchRange ranges;

    /**
     * the number of grid subdivisions to use in each dimension (the product of these values will
     * give the number of function evaluations performed to identify starting points for
     * optimizations)
     */
    private final transient int[] numSubdivisions;

    /**
     * Constructs a new <code>RegionalOptimizer</code>.
     *
     * @param  theFunction     the optimizable function to be optimized
     * @param  theRanges       the range being evaluated in each dimension
     * @param  theNumSubdivs   the number of grid subdivisions to use in each dimension (the product
     *                         of these values will give the number of function evaluations
     *                         performed to identify starting points for optimizations)
     */
    public RegionalOptimizer(final RegionalOptimizable theFunction, final SearchRange theRanges,
        final int[] theNumSubdivs) {

        int dim;

        dim = theFunction.dimension();

        this.function = theFunction;
        this.ranges = theRanges;
        this.numSubdivisions = theNumSubdivs.clone();

        if (this.ranges.getDimension() != dim) {
            throw new IllegalArgumentException("Ranges dimension must match function dimension");
        }

        if (this.numSubdivisions.length != dim) {
            throw new IllegalArgumentException(
                "Length of number of Subdivisions array must match function dimension");
        }

        // Ensure ranges are in proper order
        for (int i = 0; i < dim; i++) {

            if (this.ranges.getMin(i) == this.ranges.getMax(i)) {
                throw new IllegalArgumentException("Invalid range in dimension " + i);
            }

            if (this.numSubdivisions[i] <= 0) {
                throw new IllegalArgumentException("Invalid number of subdivisions in dimension "
                    + i);
            }
        }
    }

    /**
     * Finds all local optimal parameter values within the region.
     *
     * @param   logLevel  the level of logging to do
     * @return  a list containing the located minima. Each is an array of doubles with the (local)
     *          best-fit values for the parameters
     */
    public List<double[]> optimize(final Level logLevel) {

        int dim;
        List<double[]> accumulator;
        Optimizer opt;
        double[] scale;
        int[] point;
        double[] guess;
        int numPoints;
        double value;
        double delta;
        boolean isMin;
```

```
boolean isDuplicate;
double dist;
double[] optimum;
List<double[]> result;
double[] evaluations;
int step;
OutputValue output;

LOG.setLevel(logLevel);

accumulator = new ArrayList<double[]>(30);
result = new ArrayList<double[]>(20);

dim = this.function.dimension();
guess = new double[dim];
scale = new double[dim];
point = new int[dim];

// Determine the scale for each dimension and build an optimizer
for (int i = 0; i < dim; i++) {
    scale[i] = (this.ranges.getMax(i) - this.ranges.getMin(i)) / this.numSubdivisions[i];
}

opt = new Optimizer(this.function, scale, true);

// Now march through our grid points. At each grid point, evaluate
// the function at the point and at the surrounding points. If the
// function value at the point is the lowest, run an optimization
// there.
numPoints = 1;

for (int i = 0; i < dim; i++) {
    numPoints *= this.numSubdivisions[i];
    point[i] = 0;
    guess[i] = this.ranges.getMin(i) + (scale[i] / 2);
}

LOG.log(Level.INFO, "A total of {0} points to test", numPoints);

evaluations = new double[numPoints];

// Evaluate the function at all grid points
for (int j = 0; j < numPoints; j++) {

    output = this.function.evaluate(guess);
    evaluations[j] = output.isOutOfRange() ? Double.POSITIVE_INFINITY : output.getValue();

    // Move to the next point.
    for (int i = 0; i < dim; i++) {
        point[i]++;
        guess[i] += scale[i];

        if (point[i] < this.numSubdivisions[i]) {
            break;
        }

        point[i] = 0;
        guess[i] = this.ranges.getMin(i) + (scale[i] / 2);
    }
}

// Reset point
for (int i = 0; i < dim; i++) {
    point[i] = 0;
    guess[i] = this.ranges.getMin(i) + (scale[i] / 2);
}

LOG.info("Optimizing around minima ...");

for (int j = 0; j < numPoints; j++) {

    // Evaluate at 'guess' and at all surrounding points. If any are
    // smaller than the value at 'guess', this is NOT a minimum.
    value = evaluations[j];

    isMin = true;
    step = 1;

    for (int i = 0; i < dim; i++) {

        if ((point[i] > 0) && (evaluations[j - step] < value)) {
            isMin = false;

            break;
        }

        if ((point[i] < (this.numSubdivisions[i] - 1))
```

```
                    && (evaluations[j + step] < value)) {
                isMin = false;

                break;
            }

            step *= this.numSubdivisions[i];
        }

        // If this point is a minimum, optimize here and store result
        if (isMin) {

            try {
                optimum = opt.optimize(guess, logLevel);

                // See if the optimum lies within the ranges
                isMin = true;

                for (int i = 0; i < dim; i++) {

                    if (optimum[i] < this.ranges.getMin(i)) {
                        isMin = false;

                        break;
                    }

                    if (optimum[i] > this.ranges.getMax(i)) {
                        isMin = false;

                        break;
                    }
                }

                if (isMin) {
                    accumulator.add(optimum);
                    this.function.foundAMinium(optimum);
                }
            } catch (FailedToConvergeException e) {
                LOG.log(Level.WARNING, "Failed to locate minima: {0}", e.getMessage());
            }
        }

        // Move to the next point.
        for (int i = 0; i < dim; i++) {
            point[i]++;
            guess[i] += scale[i];

            if (point[i] < this.numSubdivisions[i]) {
                break;
            }

            point[i] = 0;
            guess[i] = this.ranges.getMin(i) + (scale[i] / 2);
        }
    }

    // Finally, we copy records into result omitting duplicates and results
    // that fall outside the allowed range.
    for (double[] minimum : accumulator) {
        isDuplicate = false;

        for (double[] test : result) {

            for (int i = 0; i < test.length; i++) {

                if (test[i] < this.ranges.getMin(i)) {
                    isDuplicate = true;

                    break;
                }

                if (test[i] > this.ranges.getMax(i)) {
                    isDuplicate = true;

                    break;
                }
            }

            if (isDuplicate) {
                break;
            }

            dist = 0;

            for (int i = 0; i < test.length; i++) {
                delta = (test[i] - minimum[i]) / scale[i];
                dist += delta * delta;
```

```
                }

                dist = Math.sqrt(dist);

                if (dist < 0.001) {
                    isDuplicate = true;

                    break;
                }
            }

            if (!isDuplicate) {
                result.add(minimum);
            }
        }

        return result;
    }
}

package com.srbenoit.math.optimizers;

/**
 * A range of data values over which to search for minima. This represents an open box in R^n.
 */
public class SearchRange {

    /** the ranges */
    private final transient double[][] ranges;

    /**
     * Constructs a new <code>SearchRange</code>.
     *
     * @param  dimension  the dimension of the data
     */
    public SearchRange(final int dimension) {

        this.ranges = new double[dimension][2];
    }

    /**
     * Gets the dimension of the search range.
     *
     * @return  the dimension
     */
    public int getDimension() {

        return this.ranges.length;
    }

    /**
     * Sets a range along a particular variable.
     *
     * @param  index  the variable index
     * @param  min    the minimum value
     * @param  max    the maximum value
     */
    public void setRange(final int index, final double min, final double max) {

        if (max >= min) {
            this.ranges[index][0] = min;
            this.ranges[index][1] = max;
        } else {
            this.ranges[index][0] = max;
            this.ranges[index][1] = min;
        }
    }

    /**
     * Gets the minimum value for a particular variable.
     *
     * @param   index  the variable index
     * @return  the minimum value
     */
    public double getMin(final int index) {

        return this.ranges[index][0];
    }

    /**
     * Gets the maximum value for a particular variable.
     *
     * @param   index  the variable index
     * @return  the maxSimum value
     */
    public double getMax(final int index) {
```

```
        return this.ranges[index][1];
    }

    /**
     * Tests whether a point is in the range.
     *
     * @param   point  the point to test
     * @return  <code>true</code> if the point is within the box in every dimension; <code>
     *          false</code> otherwise
     */
    public boolean containsPoint(final double[] point) {

        boolean inRange = true;

        for (int i = 0; i < this.ranges.length; i++) {

            if (point[i] < this.ranges[i][0]) {
                inRange = false;

                break;
            }

            if (point[i] > this.ranges[i][1]) {
                inRange = false;

                break;
            }
        }

        return inRange;
    }
}
```

E.4.6 Delaunay Triangulation (com.srbenoit.math.delaunay)

This package generates Delaunay triangulations and Voronoi diagrams of 2-dimensional point sets. *This work was adapted from other authors. Since some of the other code in this document uses this module, I include it*

```
package com.srbenoit.math.delaunay;

import java.awt.geom.Path2D;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * A subdivision of the plane by edges to form a Delaunay triangulation.
 */
public class Delaunay extends LoggedObject {

    /** the number of points to generate in the random set */
    private static final int NUM_POINTS = 250;

    /** a visualizer that will display results as triangulation proceeds */
    private final DelaunayVisualizerInt visualizer;

    /** the original list of points */
    private final Vertex[] origPoints;

    /** a starting edge for the subdivision */
    private Edge startingEdge;

    /** the list of all edges in the triangulation */
    public final List<Edge> delEdges;

    /** the list of all edges in the Voronoi diagram */
    public final List<GraphEdge> vorEdges;

    /** a map from vertex to the Voronoi region containing it */
    public Map<Vertex, Path2D> vorMap;

    /**
     * Constructs a new subdivision consisting of a single triangle that contains all points in a
     * points list.
```

```
     *
     * @param  points  the points list
     * @param  vis     a visualizer that will display results as triangulation proceeds (may be
     *                 <code>null</code>)
     */
    public Delaunay(final Vertex[] points, final DelaunayVisualizerInt vis) {

        double max;
        double xPos;
        double yPos;

        max = 0;

        for (int i = 0; i < points.length; i++) {
            xPos = Math.abs(points[i].xPos);
            yPos = Math.abs(points[i].yPos);

            if (xPos > max) {
                max = xPos;
            }

            if (yPos > max) {
                max = yPos;
            }
        }

        this.origPoints = points.clone();
        this.visualizer = vis;
        this.delEdges = new ArrayList<Edge>(points.length + 10);
        this.vorEdges = new ArrayList<GraphEdge>(points.length + 10);

        // We mark these three points as with asPoint = false so we can
        // recognize which points belong to this surrounding triangle later
        // Note: we use 3.1 rather than 3.0 since 3.0 admits the possibility
        // that a point falls on the edge of the surrounding triangle, and
        // we can have all points inside at no cost.
        set(new Vertex(3000 * max, 0, true), new Vertex(0, 3000 * max, true),
            new Vertex(-3000 * max, -3000 * max, true));
    }

    /**
     * Initializes the subdivision to contain a single triangle formed by <code>point1</code>,
     * <code>point2</code>, and <code>point3</code>.
     *
     * @param  point1  the first point
     * @param  point2  the second point
     * @param  point3  the third point
     */
    private void set(final Vertex point1, final Vertex point2, final Vertex point3) {

        Edge ea;
        Edge eb;
        Edge ec;

        ea = new QuadEdge().edge;
        ea.setEndPoints(point1, point2);

        eb = new QuadEdge().edge;
        Edge.splice(ea.sym(), eb);
        eb.setEndPoints(point2, point3);

        ec = new QuadEdge().edge;
        Edge.splice(eb.sym(), ec);
        ec.setEndPoints(point3, point1);

        Edge.splice(ec.sym(), ea);
        this.startingEdge = ea;

        this.delEdges.add(ea);
        this.delEdges.add(eb);
        this.delEdges.add(ec);
    }

    /**
     * Computes the delaunay triangulation of the points used when constructing the subdivision.
     */
    public void compute() {

        Edge edge;
        Voronoi vor;

        notify("Beginning triangulation");

        for (Vertex vert : this.origPoints) {
            this.insertSite(vert);
        }
```

```
        notify("Pruning bogus edges");

        // Remove all edges that touch a bogus vertex
        for (int i = this.delEdges.size() - 1; i >= 0; i--) {
            edge = this.delEdges.get(i);

            if (edge.org().isBogus || edge.dest().isBogus) {
                edge.delete();
                this.delEdges.remove(i);
            }
        }

        notify("Computing Voronoi diagram");

        vor = new Voronoi(1e-6);
        this.vorEdges.addAll(vor.generateVoronoi(this.origPoints));
        this.vorMap = vor.makeVoronoiPolygons(this.origPoints, this.vorEdges);

        notify("Finished");
    }

    /**
     * Inserts a new vertex into a subdivision representing a Delaunay triangulation, and fixes the
     * affected edges so the result is still a Delaunay triangulation.
     *
     * @param  vert  the vertex to insert
     */
    public void insertSite(final Vertex vert) {

        Edge edge;
        Edge base;
        Edge test;
        Vertex first;

        edge = locate(vert);

        if ((vert != edge.org()) && (vert != edge.dest())) {

            if (vert.isOnEdge(edge)) {
                test = edge.oPrev();
                edge.delete();
                this.delEdges.remove(edge);
                edge = test;
            }

            // Connect the new point to the vertices of the containing triangle
            // (or quadrilateral, if the new point fell on an existing edge)
            base = new QuadEdge().edge;
            this.delEdges.add(base);
            first = edge.org();
            base.setEndPoints(first, vert);
            Edge.splice(base, edge);

            base = edge.connect(base.sym());
            this.delEdges.add(base);
            edge = base.oPrev();

            while (edge.dest() != first) {
                base = edge.connect(base.sym());
                this.delEdges.add(base);
                edge = base.oPrev();
            }

            // Examine suspect edges to ensure that the Delaunay condition is
            // satisfied
            for (;;) {
                test = edge.oPrev();

                if (test.dest().isRightOf(edge)
                        && vert.isInCircle(edge.org(), test.dest(), edge.dest())) {
                    edge.swap();

                    // edge = test; // Documented bug
                    edge = edge.oPrev();
                } else if (edge.org() == first) {
                    break;
                }

                edge = edge.oNext().lPrev();
            }
        }
    }

    /**
     * Returns an edge <code>e</code> such that either <code>point</code> is on <code>e</code> or
     * <code>e</code> is an edge of a triangle containing <code>point</code>. The search starts
     * from <code>startingEdge</code> and proceeds in the general direction of <code>point</code>.
```

```
     *
     * @param   point  the point
     * @return  the edge
     */
    public Edge locate(final Vertex point) {

        Edge edge;

        edge = this.startingEdge;

        for (;;) {

            if ((point == edge.org()) || (point == edge.dest())) {
                break;
            }

            if (point.isRightOf(edge)) {
                edge = edge.sym();
            } else if (point.isLeftOf(edge.oNext())) {
                edge = edge.oNext();
            } else if (point.isLeftOf(edge.dPrev())) {
                edge = edge.dPrev();
            } else {
                break;
            }
        }

        return edge;
    }

    /**
     * If there is a visualizer registered, sends a notification to that visualizer to update its
     * view, and optionally pause before proceeding.
     *
     * @param  phase  a string description of the phase just completed
     */
    private void notify(final String phase) {

        if (this.visualizer != null) {
            this.visualizer.update(this.origPoints, this, phase);
        }
    }

    /**
     * Main method that generates a random set of points, builds a visualizer and a Delaunay
     * triangulator, then steps through the process asking the user for a button press after each
     * step to move on.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        Random rnd;
        Vertex[] points;
        DelaunayViewer viewer;
        Delaunay triangulator;
        double xPos;
        double yPos;

        rnd = new Random(System.currentTimeMillis());
        points = new Vertex[NUM_POINTS];

        for (int i = 0; i < NUM_POINTS; i++) {
            xPos = (rnd.nextDouble() - 0.5);
            yPos = (rnd.nextDouble() - 0.5);
            points[i] = new Vertex(xPos, yPos); // NOPMD SRB
        }

        try {
            viewer = new DelaunayViewer();

            triangulator = new Delaunay(points, viewer);
            triangulator.compute();
        } catch (Exception e) {
            LOG.log(Level.WARNING, "Exception while creating Delaunay triangulation", e);
        }
    }
}

package com.srbenoit.math.delaunay;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.geom.Line2D;
```

```
import java.awt.geom.Path2D;
import java.awt.geom.PathIterator;
import java.awt.image.BufferedImage;
import java.util.Map;
import java.util.Random;
import javax.swing.JPanel;
import com.srbenoit.color.Gradient;

/**
 * A panel that can visualize a Delaunay triangulation at any point in the process.
 */
public class DelaunayPanel extends JPanel {

    /** version number for serialization */
    private static final long serialVersionUID = -4125801564718139263L;

    /** the width of the offscreen image */
    private static final int IMGWIDTH = 920;

    /** the height of the offscreen image */
    private static final int IMGHEIGHT = 920;

    /** the portion of window to leave as margin on each side */
    private static final double MARGIN = 0.05;

    /** green color for leaf triangles */
    private static final Color LINES = new Color(0, 255, 0);

    /** the left edge of a bounding box that contains the points */
    private double minX;

    /** the right edge of a bounding box that contains the points */
    private double maxX;

    /** the bottom edge of a bounding box that contains the points */
    private double minY;

    /** the top edge of a bounding box that contains the points */
    private double maxY;

    /** X axis scale factor to take points to pixels */
    private double xScale;

    /** X axis offset to take points to pixels */
    private double xOffset;

    /** Y axis scale factor to take points to pixels */
    private double yScale;

    /** Y axis offset to take points to pixels */
    private double yOffset;

    /** the offscreen image */
    private final BufferedImage offscreen;

    /** a gradient over hue with 100 steps */
    private final Gradient gradient;

    /**
     * Constructs a new <code>DelaunayPanel</code>.
     */
    public DelaunayPanel() {

        super();

        setBackground(Color.BLACK);
        setPreferredSize(new Dimension(IMGWIDTH, IMGHEIGHT));

        this.offscreen = new BufferedImage(IMGWIDTH, IMGHEIGHT, BufferedImage.TYPE_INT_RGB);

        this.gradient = new Gradient(100, 1);
    }

    /**
     * Notifies that another step in the process has completed, and visualization can be updated.
     * Processing will not continue until this method returns, allowing a visualizer to show a
     * slow, step-by-step process.
     *
     * @param  vertices  the list of points being triangulated
     * @param  sub       the subdivision to draw
     */
    public void update(final Vertex[] vertices, final Delaunay sub) {

        Random rnd;
        Graphics2D grx;
        int pixX1;
        int pixY1;
```

```
int pixX2;
int pixY2;
Map<Vertex, Path2D> map;
double[] seg;
double lastX = 0;
double lastY = 0;
Path2D scaled;
Line2D newLine;
Color color;
PathIterator iter;

seg = new double[] { 0, 0, 0, 0, 0, 0 };
rnd = new Random();

// Compute the bounding box based on the points
getBoundingBox(vertices);

// Compute the transform to convert points to pixels
computeTransform();

// Do all drawing in a synchronized block
synchronized (this.offscreen) {
    grx = (Graphics2D) this.offscreen.getGraphics();
    grx.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);

    // Clear the window
    grx.setColor(Color.BLACK);
    grx.fillRect(0, 0, IMGWIDTH, IMGHEIGHT);

    // Draw Voronoi regions
    map = sub.vorMap;

    if (map != null) {

        for (Path2D path : map.values()) {
            color = this.gradient.getColor(rnd.nextInt(100));
            color = new Color(color.getRed() / 2, color.getGreen() / 2, // NOPMD SRB
                    color.getBlue() / 2);
            grx.setColor(color);

            scaled = new Path2D.Double(); // NOPMD SRB

            iter = path.getPathIterator(null);

            while (!iter.isDone()) {

                switch (iter.currentSegment(seg)) {

                case PathIterator.SEG_MOVETO:
                    lastX = seg[0];
                    lastY = seg[1];
                    break;

                case PathIterator.SEG_LINETO:
                    newLine = new Line2D.Double(scaleX(lastX), scaleY(lastY), // NOPMD SRB
                            scaleX(seg[0]), scaleY(seg[1]));
                    scaled.append(newLine, true);
                    lastX = seg[0];
                    lastY = seg[1];
                    break;

                default:
                    break;
                }

                iter.next();
            }

            grx.fill(scaled);
        }
    }

    // Draw all the edges in green
    grx.setColor(LINES);

    for (Edge edge : sub.delEdges) {
        pixX1 = scaleX(edge.org().xPos);
        pixY1 = scaleY(edge.org().yPos);
        pixX2 = scaleX(edge.dest().xPos);
        pixY2 = scaleY(edge.dest().yPos);
        grx.drawLine(pixX1, pixY1, pixX2, pixY2);
    }

    // Draw all the voronoi boundaries in dark gray
    grx.setColor(Color.RED);
```

```
            for (GraphEdge edge : sub.vorEdges) {

                pixX1 = scaleX(edge.xPos1);
                pixY1 = scaleY(edge.yPos1);
                pixX2 = scaleX(edge.xPos2);
                pixY2 = scaleY(edge.yPos2);
                grx.drawLine(pixX1, pixY1, pixX2, pixY2);
            }

            // Plot the points
            grx.setColor(Color.YELLOW);

            for (Edge edge : sub.delEdges) {

                if (edge.data != null) {
                    pixX1 = scaleX(edge.data.xPos);
                    pixY1 = scaleY(edge.data.yPos);
                    grx.fillOval(pixX1 - 1, pixY1 - 1, 3, 3);
                }

                if (edge.sym().data != null) {
                    pixX1 = scaleX(edge.sym().data.xPos);
                    pixY1 = scaleY(edge.sym().data.yPos);
                    grx.fillOval(pixX1 - 1, pixY1 - 1, 3, 3);
                }
            }
        }

        repaint();
    }

    /**
     * Paints the panel.
     *
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    @Override public void paintComponent(final Graphics grx) {

        super.paintComponent(grx);

        synchronized (this.offscreen) {
            grx.drawImage(this.offscreen, 0, 0, getWidth(), getHeight(), null);
        }
    }

    /**
     * Computes the bounding box of a set of points. The box will be just large enough to contain
     * the points, which could possibly mean having zero width or height if points are colinear.
     * The bounding rectangle is stored in the member variable <code>bounds</code>.
     *
     * @param  points  the set of points
     */
    private void getBoundingBox(final Vertex[] points) {

        Vertex vert;

        vert = points[0];

        this.minX = vert.xPos;
        this.maxX = this.minX;
        this.minY = vert.yPos;
        this.maxY = this.minY;

        for (int i = 1; i < points.length; i++) {
            vert = points[i];

            if (vert.xPos < this.minX) {
                this.minX = vert.xPos;
            }

            if (vert.xPos > this.maxX) {
                this.maxX = vert.xPos;
            }

            if (vert.yPos < this.minY) {
                this.minY = vert.yPos;
            }

            if (vert.yPos > this.maxY) {
                this.maxY = vert.yPos;
            }
        }

        // Ensure that no edge is zero, to make matrix computation easier
        if (Math.abs(this.minX) < 0.0000001) {
            this.minX = -0.0000001;
        }
```

```
            if (Math.abs(this.maxX) < 0.0000001) {
                this.maxX = 0.0000001;
            }

            if (Math.abs(this.minY) < 0.0000001) {
                this.minY = -0.0000001;
            }

            if (Math.abs(this.maxY) < 0.0000001) {
                this.maxY = 0.0000001;
            }
        }

        /**
         * Computes the transformation matrix to map a rectangle to an area in the panel covering 80%
         * of the panel in each dimension (that is, the panel will have a 10% margin on each edge).
         *
         * <pre>
         * [ . . ] [x] = [X]
         * [ . . ] [y]   [Y]
         * </pre>
         *
         * where (x,y) is the real-valued point space, (X, Y) is the pixel, and (minX,minY) maps to
         * (w/10,9h/10) and (maxX,maxY) maps to (9w/10,h/10).
         */
        private void computeTransform() {

            // Compute scale factors for X and Y axis (flipping Y axis)
            if (this.maxX == this.minX) {
                this.xScale = 1;
            } else {
                this.xScale = (1 - (2 * MARGIN)) * IMGWIDTH / (this.maxX - this.minX);
            }

            if (this.maxY == this.minY) {
                this.yScale = -1;
            } else {
                this.yScale = -(1 - (2 * MARGIN)) * IMGHEIGHT / (this.maxY - this.minY);
            }

            if (-this.yScale > this.xScale) {
                this.yScale = -this.xScale;
            } else if (this.xScale > -this.yScale) {
                this.xScale = -this.yScale;
            }

            // Now set offset so (minX, minY) maps to (0.1W, 0.9H)
            this.xOffset = (MARGIN * IMGWIDTH) - (this.xScale * this.minX);
            this.yOffset = ((1 - MARGIN) * IMGHEIGHT) - (this.yScale * this.minY);
        }

        /**
         * Generates an X pixel from a real X coordinate in point space.
         *
         * @param   coord  the X coordinate
         * @return  the X pixel value
         */
        private int scaleX(final double coord) {

            return (int) ((this.xScale * coord) + this.xOffset + 0.5);
        }

        /**
         * Generates a Y pixel from a real Y coordinate in point space.
         *
         * @param   coord  the Y coordinate
         * @return  the Y pixel value
         */
        private int scaleY(final double coord) {

            return (int) ((this.yScale * coord) + this.yOffset + 0.5);
        }
    }

    package com.srbenoit.math.delaunay;

    import java.awt.BorderLayout;
    import java.lang.reflect.InvocationTargetException;
    import javax.swing.JFrame;
    import javax.swing.JLabel;
    import javax.swing.JPanel;
    import javax.swing.SwingUtilities;
    import javax.swing.WindowConstants;
    import com.srbenoit.ui.UIUtilities;

    /**
```

```
 * A viewer that will display the that can show a visualization of the triangulation.
 */
public class DelaunayViewer implements DelaunayVisualizerInt, Runnable {

    /** flag that controls whether we pause after each step */
    private static final boolean PAUSE = false;

    /** the panel that will display the triangulation */
    private DelaunayPanel panel;

    /** a label that will show the description of the phase just completed */
    private JLabel label;

    /**
     * Constructs a new <code>DelaunayViewer</code>.
     *
     * @throws  InvocationTargetException  if there is an error constructing the interface
     * @throws  InterruptedException       if there is an interruption while constructing the
     *                                     interface
     */
    public DelaunayViewer() throws InterruptedException, InvocationTargetException {

        SwingUtilities.invokeAndWait(this);
    }

    /**
     * Notifies that another step in the process has completed, and visualization can be updated.
     * Processing will not continue until this method returns, allowing a visualizer to show a
     * slow, step-by-step process.
     *
     * @param  points  the list of points being triangulated
     * @param  sub     the subdivision to draw
     * @param  phase   a string description of the phase just completed
     */
    public void update(final Vertex[] points, final Delaunay sub, final String phase) {

        this.label.setText(phase);
        this.panel.update(points, sub);

        if (PAUSE) {

            while (this.panel.isVisible()) {

                synchronized (this) {

                    try {
                        wait(1000);
                    } catch (InterruptedException e) {
                        // No action
                    }
                }
            }
        }
    }

    /**
     * Constructs the user interface in the AWT event dispatcher thread.
     */
    public void run() {

        JFrame frame;
        JPanel content;

        frame = new JFrame();
        frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

        content = new JPanel(new BorderLayout());
        frame.setContentPane(content);

        this.panel = new DelaunayPanel();
        content.add(this.panel, BorderLayout.CENTER);

        this.label = new JLabel("Initializing...");
        content.add(this.label, BorderLayout.NORTH);

        frame.pack();
        UIUtilities.positionFrame(frame, 0.5, 0.1);
        frame.setVisible(true);
    }
}

package com.srbenoit.math.delaunay;

/**
 * An interface for classes that can display a visualization of the process as it occurs.
 */
public interface DelaunayVisualizerInt {
```

```
    /**
     * Notifies that another step in the process has completed, and visualization can be updated.
     * Processing will not continue until this method returns, allowing a visualizer to show a slow,
     * step-by-step process.
     *
     * @param  points  the list of points being triangulated
     * @param  sub     the subdivision to draw
     * @param  phase   a string description of the phase just completed
     */
    void update(Vertex[] points, Delaunay sub, String phase);
}

package com.srbenoit.math.delaunay;

/**
 * A single edge in a QuadEdge data structure.
 */
public class Edge {

    /** the next edge of 4 (counterclockwise) in the edge QuadEdge */
    public Edge rotEdge;

    /** the next edge of 4 (clockwise) in the edge QuadEdge */
    public Edge invrotEdge;

    /** the next edge counterclockwise from the origin vertex */
    public Edge next;

    /** data associated with the edge (origin vertex point) */
    public Vertex data;

    /** data associated with the edge (origin vertex point) */
    public boolean voronoi = false;

    /**
     * Gets the edge from the right face to the left face relative to this edge (think of this as
     * the edge that is rotated 90 degrees from this edge).
     *
     * @return  the edge
     */
    public Edge rot() {

        return this.rotEdge;
    }

    /**
     * Gets the edge from the left face to the right face relative to this edge (think of this as
     * the edge that is rotated -90 degrees from this edge).
     *
     * @return  the edge
     */
    public Edge invRot() {

        return this.invrotEdge;
    }

    /**
     * Gets the edge that connects the same two vertices as this edge, but with opposite
     * orientation.
     *
     * @return  the edge
     */
    public Edge sym() {

        return this.rotEdge.rotEdge;
    }

    /**
     * Gets the next edge we find that leaves this edge's origin vertex as we go around that vertex
     * counterclockwise from this edge.
     *
     * @return  the edge
     */
    public Edge oNext() {

        return this.next;
    }

    /**
     * Gets the next edge we find that leaves this edge's origin vertex as we go around that vertex
     * clockwise from this edge.
     *
     * @return  the edge
     */
    public Edge oPrev() {
```

```
        return this.rotEdge.next.rotEdge;
    }

    /**
     * Gets the next edge we find that enters this edge's destination vertex as we go around that
     * vertex counterclockwise from this edge.
     *
     * @return  the edge
     */
    public Edge dNext() {

        return this.rotEdge.rotEdge.next.rotEdge.rotEdge;
    }

    /**
     * Gets the next edge we find that enters this edge's destination vertex as we go around that
     * vertex clockwise from this edge.
     *
     * @return  the edge
     */
    public Edge dPrev() {

        return this.invrotEdge.next.invrotEdge;
    }

    /**
     * Gets the edge that follows this edge in a counterclockwise traversal of the face to the left
     * of this edge.
     *
     * @return  the edge
     */
    public Edge lNext() {

        return this.invrotEdge.next.rotEdge;
    }

    /**
     * Gets the edge that precedes this edge in a counterclockwise traversal of the face to the
     * left of this edge.
     *
     * @return  the edge
     */
    public Edge lPrev() {

        return this.next.rotEdge.rotEdge;
    }

    /**
     * Gets the edge that follows this edge in a counterclockwise traversal of the face to the
     * right of this edge (note that such a traversal goes opposite the direction of this edge).
     *
     * @return  the edge
     */
    public Edge rNext() {

        return this.rotEdge.next.invrotEdge;
    }

    /**
     * Gets the edge that precedes this edge in a counterclockwise traversal of the face to the
     * right of this edge (note that such a traversal goes opposite the direction of this edge).
     *
     * @return  the edge
     */
    public Edge rPrev() {

        return this.rotEdge.rotEdge.next;
    }

    /**
     * Gets the data associated with this edge's origin vertex.
     *
     * @return  the origin vertex data
     */
    public Vertex org() {

        return this.data;
    }

    /**
     * Gets the data associated with this edge's destination vertex.
     *
     * @return  the destination vertex data
     */
    public Vertex dest() {

        return this.rotEdge.rotEdge.data;
```

```
}

/**
 * Gets the data associated with the face to the left of this edge.
 *
 * @return  the left face data
 */
public Vertex left() {

    return this.rotEdge.data;
}

/**
 * Gets the data associated with the face to the right of this edge.
 *
 * @return  the right face data
 */
public Vertex right() {

    return this.invrotEdge.data;
}

/**
 * Sets the end points of this edge. The points of the dual space (faces) can be set using
 * <code>rotEdge().setEndPoints</code>.
 *
 * @param  org   the origin vertex
 * @param  dest  the destination vertex
 */
public void setEndPoints(final Vertex org, final Vertex dest) {

    this.data = org;
    sym().data = dest;
}

/**
 * Generates the string representation of the edge.
 *
 * @return  the string representation
 */
@Override public String toString() {

    return "{" + org() + ":" + dest() + "}";
}

/**
 * Implementation of the Splice topological primitive from Guibas & Stolfi (1985). If the two
 * edges have the same origin vertex (they are parts of different loops that leave that
 * vertex), the vertex is split. If the two edges have different origin vertices, those
 * vertices are merged.
 *
 * @param  edge1  the first edge
 * @param  edge2  the second edge
 */
public static void splice(final Edge edge1, final Edge edge2) {

    Edge alpha;
    Edge beta;
    Edge tt1;
    Edge tt2;
    Edge tt3;
    Edge tt4;

    alpha = edge1.next.rotEdge;
    beta = edge2.next.rotEdge;

    tt1 = edge2.next;
    tt2 = edge1.next;
    tt3 = beta.next;
    tt4 = alpha.next;

    edge1.next = tt1;
    edge2.next = tt2;
    alpha.next = tt3;
    beta.next = tt4;
}

/**
 * Deletes the edge, adjusting the references of surrounding edges as needed.
 */
public void delete() {

    splice(this, oPrev());
    splice(sym(), sym().oPrev());
}

/**
```

```
     * Add a new edge connecting the destination of this edge to the origin of <code>edge2</code>
     * in such a way that all three have the same left face after the connection is complete. The
     * data pointers of the new edge are set to the appropriate vertices from the existing edges.
     *
     * @param   edge2  the second edge
     * @return  the new edge
     */
    public Edge connect(final Edge edge2) {

        Edge edge;

        edge = new QuadEdge().edge;
        Edge.splice(edge, lNext());
        Edge.splice(edge.sym(), edge2);
        edge.setEndPoints(dest(), edge2.org());

        return edge;
    }

    /**
     * Turns the edge counterclockwise inside its enclosing quadrilateral, updating data pointers
     * appropriately.
     */
    public void swap() {

        Edge aEdge;
        Edge bEdge;

        aEdge = oPrev();
        bEdge = sym().oPrev();

        Edge.splice(this, aEdge);
        Edge.splice(sym(), bEdge);
        Edge.splice(this, aEdge.lNext());
        Edge.splice(sym(), bEdge.lNext());

        setEndPoints(aEdge.dest(), bEdge.dest());
    }
}

package com.srbenoit.math.delaunay;

/**
 */
public class EL {

    /** array index of left edge */
    private final static int LEFTEDGE = 0;

    /** array index of right edge */
    private final static int RIGHTEDGE = 1;

    /** the minimum Y coordinate of any site */
    private final double minX;

    /** the height of the bounding box */
    private final double width;

    /** the length of the hash table */
    private final int hashsize;

    /** the hash table */
    private Halfedge[] hash;

    /** the leftmost element in the list */
    public Halfedge leftEnd;

    /** the rightmost element in the list */
    public Halfedge rightEnd;

    /**
     * Constructs a new <code>EL</code>.
     *
     * @param  sqrtNsites  the square root of (the number of sites + 4)
     * @param  minSiteX    the minimum X value of any site
     * @param  siteXspan   the total X width spanned by all sites
     */
    public EL(final int sqrtNsites, final double minSiteX, final double siteXspan) {

        this.minX = minSiteX;
        this.width = siteXspan;

        this.hashsize = 2 * sqrtNsites;
        this.hash = new Halfedge[this.hashsize];

        for (int i = 0; i < this.hashsize; i += 1) {
            this.hash[i] = null;
```

```
        }

        this.leftEnd = new Halfedge(null, 0);
        this.rightEnd = new Halfedge(null, 0);

        this.leftEnd.elLeft = null;
        this.leftEnd.elRight = this.rightEnd;

        this.rightEnd.elLeft = this.leftEnd;
        this.rightEnd.elRight = null;

        this.hash[0] = this.leftEnd;
        this.hash[this.hashsize - 1] = this.rightEnd;
    }

    /**
     * Inserts an edge in the linked list to the right of a specified edge.
     *
     * @param  lb      the edge that should be to the left of the inserted edge
     * @param  newHe   the edge to insert
     */
    public void insert(final Halfedge lb, final Halfedge newHe) {

        newHe.elLeft = lb;
        newHe.elRight = lb.elRight;
        lb.elRight.elLeft = newHe;
        lb.elRight = newHe;
    }

    /**
     * Deletes an edge from the linked list.
     *
     * @param  edge   the edge to delete
     */
    public void delete(final Halfedge edge) {

        edge.elLeft.elRight = edge.elRight;
        edge.elRight.elLeft = edge.elLeft;
        edge.deleted = true;
    }

    /**
     * Gets entry from hash table, pruning any deleted nodes.
     *
     * @param    index  the index in the hash table
     * @return   the found entry
     */
    public Halfedge gethash(final int index) {

        Halfedge edge;

        if ((index < 0) || (index >= this.hashsize)) {
            edge = null;
        } else {
            edge = this.hash[index];

            if ((edge != null) && edge.deleted) {
                this.hash[index] = null;
                edge = null;
            }
        }

        return edge;
    }

    /**
     * ???
     *
     * @param    vert  the vertex
     * @return   the located half-edge
     */
    public Halfedge leftbnd(final Vertex vert) {

        int bucket;
        Halfedge he;

        // identify the bucket near the point based on it's X coordinate
        bucket = (int) ((vert.xPos - this.minX) / this.width * this.hashsize);

        // check the bucket range
        if (bucket < 0) {
            bucket = 0;
        } else if (bucket >= this.hashsize) {
            bucket = this.hashsize - 1;
        }

        he = gethash(bucket);
```

```
    if (he == null) {

        // if the HE isn't found, search backwards and forwards in the hash
        // map for the first non-null entry
        for (int i = 1; i < this.hashsize; i += 1) {

            if ((he = gethash(bucket - i)) != null) {
                break;
            }

            if ((he = gethash(bucket + i)) != null) {
                break;
            }
        }
    }

    if (he != null) {

        // search linear list of half edges for the correct one
        if ((he == this.leftEnd) || ((he != this.rightEnd) && rightOf(he, vert))) {

            // keep going right on the list until either the end is
            // reached, or you find the 1st edge which the point isn't to
            // the right of
            do {
                he = he.elRight;
            } while ((he != this.rightEnd) && rightOf(he, vert));

            he = he.elLeft;
        } else {

            // if the point is to the left of the HalfEdge, then search
            // left for the HE just to the left of the point
            do {
                he = he.elLeft;
            } while ((he != this.leftEnd) && !rightOf(he, vert));
        }
    }

    // update hash table and reference counts
    if ((bucket > 0) && (bucket < (this.hashsize - 1))) {
        this.hash[bucket] = he;
    }

    return he;
}

/**
 * Tests if p is to right of half edge e.
 *
 * @param    el      the half sedge
 * @param    point   the point
 * @return  <code>true</code> if <code>point</code> is to the right of <code>e</code>
 */
private boolean rightOf(final Halfedge el, final Vertex point) {

    VorEdge e;
    Site topsite;
    boolean rightOfSite;
    boolean above;
    boolean fast;
    double dxp;
    double dyp;
    double dxs;
    double t1;
    double t2;
    double t3;
    double yl;
    boolean result;

    e = el.elEdge;
    topsite = e.reg[1];
    rightOfSite = (point.xPos > topsite.xPos);

    if (rightOfSite && (el.elPm == LEFTEDGE)) {
        result = true;
    } else if (!rightOfSite && (el.elPm == RIGHTEDGE)) {
        result = false;
    } else {

        if (e.aParam == 1.0) {
            dyp = point.yPos - topsite.yPos;
            dxp = point.xPos - topsite.xPos;
            fast = false;

            if ((!rightOfSite & (e.bParam < 0.0)) | (rightOfSite & (e.bParam >= 0.0))) {
```

```
                    above = dyp >= (e.bParam * dxp);
                    fast = above;
                } else {
                    above = (point.xPos + (point.yPos * e.bParam)) > e.cParam;

                    if (e.bParam < 0.0) {
                        above = !above;
                    }

                    if (!above) {
                        fast = true;
                    }
                }

                if (!fast) {
                    dxs = topsite.xPos - (e.reg[0]).xPos;
                    above = (e.bParam * ((dxp * dxp) - (dyp * dyp)))
                        < (dxs * dyp * (1.0 + (2.0 * dxp / dxs) + (e.bParam * e.bParam)));

                    if (e.bParam < 0.0) {
                        above = !above;
                    }
                }
            } else {
                yl = e.cParam - (e.aParam * point.xPos);
                t1 = point.yPos - yl;
                t2 = point.xPos - topsite.xPos;
                t3 = yl - topsite.yPos;
                above = (t1 * t1) > ((t2 * t2) + (t3 * t3));
            }

            result = (el.elPm == LEFTEDGE) ? above : (!above);
        }

        return result;
    }
}
```

```
package com.srbenoit.math.delaunay;

/**
 * An edge in the Voronoi graph.
 */
public class GraphEdge {

    /** The start X coordinate */
    public double xPos1;

    /** The start Y coordinate */
    public double yPos1;

    /** The end X coordinate */
    public double xPos2;

    /** The endY coordinate */
    public double yPos2;

    /** the index of the site to the left of the edge */
    public int site1;

    /** the index of the site to the right of the edge */
    public int site2;
}
```

```
package com.srbenoit.math.delaunay;

/**
 */
public class Halfedge {

    /**  */
    public Halfedge elLeft;

    /**  */
    public Halfedge elRight;

    /**  */
    public VorEdge elEdge;

    /**  */
    public boolean deleted;

    /**  */
    public int elPm;

    /**  */
    public Site vertex;
```

```
    /**  */
    public double ystar;

    /** the next half-edge in the hash table */
    public Halfedge pqNext;

    /**
     * Constructs a new <code>Halfedge</code>.
     */
    public Halfedge() {

        this.pqNext = null;
    }

    /**
     * Constructs a new <code>Halfedge</code>.
     *
     * @param  edge  ???
     * @param  pm    ???
     */
    public Halfedge(final VorEdge edge, final int pm) {

        this();

        this.elEdge = edge;
        this.elPm = pm;
        this.vertex = null;
    }

}

package com.srbenoit.math.delaunay;

/**
 * A hash table to store half-edges.
 */
public class PQ {

    /** the minimum Y coordinate of any site */
    private final double minY;

    /** the height of the bounding box */
    private final double height;

    /** the number of half-edges in the hash table */
    private int count; // NOPMD

    /** the minimum hash bucket */
    private int minBucket;

    /**  */
    private final int hashsize;

    /**  */
    private final Halfedge[] hash;

    /**
     * Constructs a new <code>PQ</code>.
     *
     * @param  sqrtNsites  the square root of (the number of sites + 4)
     * @param  minSiteY    the minimum Y value of any site
     * @param  siteYspan   the total Y height spanned by all sites
     */
    public PQ(final int sqrtNsites, final double minSiteY, final double siteYspan) {

        this.minY = minSiteY;
        this.height = siteYspan;

        this.count = 0;
        this.minBucket = 0;
        this.hashsize = 4 * sqrtNsites;
        this.hash = new Halfedge[this.hashsize];

        for (int i = 0; i < this.hashsize; i += 1) {
            this.hash[i] = new Halfedge(); // NOPMD SRB
        }
    }

    /**
     * Determines the bucket where a half-edge should be stored. The bucket is based on the y*
     * value, and runs from 0 (y* is at the minimum y) to the hash-size (y* is at the maximum y).
     *
     * <p>The minimum hash bucket is updated if the result is smaller than the current minimum.
     *
     * @param   he  the half-edge
     * @return  the bucket
```

```
 */
public int bucket(final Halfedge he) {

    int bucket;

    bucket = (int) ((he.ystar - this.minY) / this.height * this.hashsize);

    if (bucket < 0) {
        bucket = 0;
    }

    if (bucket >= this.hashsize) {
        bucket = this.hashsize - 1;
    }

    if (bucket < this.minBucket) {
        this.minBucket = bucket;
    }

    return bucket;
}

/**
 * Adds the HalfEdge to the ordered linked list of vertices in its appropriate bucket.
 *
 * @param  he       the half edge to add
 * @param  v        the vertex the half edge is associated with
 * @param  offset   the offset from y to y* (distance to nearest site)
 */
public void insert(final Halfedge he, final Site v, final double offset) {

    Halfedge last;
    Halfedge next;

    he.vertex = v;
    he.ystar = v.yPos + offset;

    // Get the first entry in the bucket this half edge should be in
    // (based on its y-start value)
    last = this.hash[bucket(he)];

    // Insert the half edge in the bucket's linked list such that the
    // items in the linked list are sorted by increasing y-star
    next = last.pqNext;

    while ((next != null)
            && ((he.ystar > next.ystar)
                    || ((he.ystar == next.ystar) && (v.xPos > next.vertex.xPos)))) {
        last = next;
        next = last.pqNext;
    }

    he.pqNext = last.pqNext;
    last.pqNext = he;

    this.count += 1;
}

/**
 * Removes the HalfEdge from the list of vertices.
 *
 * @param  he  the half edge to remove
 */
public void delete(final Halfedge he) {

    Halfedge last;

    if (he.vertex != null) {
        last = this.hash[bucket(he)];

        while (last.pqNext != he) {
            last = last.pqNext;
        }

        last.pqNext = he.pqNext;
        this.count -= 1;
        he.vertex = null;
    }
}

/**
 * Tests whether the hashtable is empty.
 *
 * @return  <code>true</code> if the hash table is empty
 */
public boolean empty() {
```

```
        return (this.count == 0);
    }

    /**
     * Gets the vertex in the hashtable with the minimum y-star value.
     *
     * @return  the vertex
     */
    public Vertex min() {

        while (this.hash[this.minBucket].pqNext == null) {
            this.minBucket += 1;
        }

        return new Vertex(this.hash[this.minBucket].pqNext.vertex.xPos,
                this.hash[this.minBucket].pqNext.ystar);
    }

    /**
     * Gets the half edge with the minimum y-star value and deletes that half edge from the
     * hashtable.
     *
     * @return  the half-edge
     */
    public Halfedge extractMin() {

        Halfedge curr;

        curr = this.hash[this.minBucket].pqNext;
        this.hash[this.minBucket].pqNext = curr.pqNext;
        this.count -= 1;

        return curr;
    }
}
```

```
package com.srbenoit.math.delaunay;

/**
 * A QuadEdge, consisting of four edges linked appropriately. The edges form a cyclical,
 * doubly-linked list, and each edge has a link to the next edge in a counterclockwise edge list
 * around its origin vertex.
 */
public class QuadEdge {

    /** the "canonical" edge in this QuadEdge structure */
    public Edge edge;

    /**
     * Constructs a new <code>QuadEdge</code>, implementing the "MakeEdge" topological operator from
     * Guibas & Stolfi (1985).
     */
    public QuadEdge() {

        Edge edge2;
        Edge edge3;
        Edge edge4;

        this.edge = new Edge();
        edge2 = new Edge();
        edge3 = new Edge();
        edge4 = new Edge();

        this.edge.rotEdge = edge2;
        edge2.rotEdge = edge3;
        edge3.rotEdge = edge4;
        edge4.rotEdge = this.edge;

        this.edge.invrotEdge = edge4;
        edge4.invrotEdge = edge3;
        edge3.invrotEdge = edge2;
        edge2.invrotEdge = this.edge;

        this.edge.next = this.edge;
        edge2.next = edge4;
        edge3.next = edge3;
        edge4.next = edge2;
    }
}
```

```
package com.srbenoit.math.delaunay;

/**
 * A site or vertex in the Voronoi diagram.
 */
public class Site extends Vertex implements Comparable<Site> {
```

```
    /** the index of the site */
    public int sitenbr;

    /**
     * Constructs a new <code>Site</code>.
     *
     * @param  xCoord  the X coordinate
     * @param  yCoord  the Y coordinate
     * @param  num     the site number
     */
    public Site(final double xCoord, final double yCoord, final int num) {

        super(xCoord, yCoord);

        this.sitenbr = num;
    }

    /**
     * Compares this object with the specified object for order. Returns a negative integer, zero,
     * or a positive integer as this object is less than, equal to, or greater than the specified
     * object.
     *
     * @param   obj  the object to be compared
     * @return  a negative integer, zero, or a positive integer as this object is less than, equal
     *          to, or greater than the specified object.
     */
    public int compareTo(final Site obj) {

        int result;

        if (this.yPos < obj.yPos) {
            result = -1;
        } else if (this.yPos > obj.yPos) {
            result = 1;
        } else if (this.xPos < obj.xPos) {
            result = -1;
        } else if (this.xPos > obj.xPos) {
            result = 1;
        } else {
            result = 0;
        }

        return result;
    }

    /**
     * Tests this object for equality with another object. To be equal, the other object must be a
     * site, and must have the same X and Y coordinates. The site number is not used in comparison.
     *
     * @param  obj  the object to compare
     */
    @Override public boolean equals(final Object obj) {

        Site site;
        boolean equal;

        if (obj instanceof Site) {
            site = (Site) obj;
            equal = (site.xPos == this.xPos) && (site.yPos == this.yPos);
        } else {
            equal = false;
        }

        return equal;
    }

    /**
     * Generates the hash code of the site.
     *
     * @return  the hash code
     */
    @Override public int hashCode() {

        return (int) Double.doubleToLongBits(this.xPos + this.yPos);
    }
}

package com.srbenoit.math.delaunay;

/**
 * A vertex in a Delaunay triangulation.
 */
public class Vertex {

    /** distance within which we consider a vertex to fall on an edge */
    public static final double EPSILON = 1e-6;
```

```
/** the X coordinate */
public final double xPos;

/** the Y coordinate */
public final double yPos;

/** flag indicating the vertex is bogus */
public final boolean isBogus;

/**
 * Constructs a new <code>Vertex</code>.
 *
 * @param  xCoord  the X coordinate
 * @param  yCoord  the Y coordinate
 */
public Vertex(final double xCoord, final double yCoord) {

    this(xCoord, yCoord, false);
}

/**
 * Constructs a new <code>Vertex</code>.
 *
 * @param  xCoord  the X coordinate
 * @param  yCoord  the Y coordinate
 * @param  bogus   <code>true</code> if the vertex is bogus
 */
public Vertex(final double xCoord, final double yCoord, final boolean bogus) {

    this.xPos = xCoord;
    this.yPos = yCoord;
    this.isBogus = bogus;
}

/**
 * Returns the square of the distance of this vertex from the origin.
 *
 * @return  the distance from the origin squared
 */
public double lengthSquared() {

    return (this.xPos * this.xPos) + (this.yPos * this.yPos);
}

/**
 * Tests whether a point lies to the right of an edge.
 *
 * @param   edge  the edge
 * @return  <code>true</code> if the point lies to the right of the edge; <code>false</code>
 *          otherwise
 */
public boolean isRightOf(final Edge edge) {

    return isCcw(this, edge.dest(), edge.org());
}

/**
 * Tests whether a point lies to the left of an edge.
 *
 * @param   edge  the edge
 * @return  <code>true</code> if the point lies to the left of the edge; <code>false</code>
 *          otherwise
 */
public boolean isLeftOf(final Edge edge) {

    return isCcw(this, edge.org(), edge.dest());
}

/**
 * Tests whether this point lies on an edge (within some small epsilon).
 *
 * @param   edge  the edge
 * @return  <code>true</code> if the point lies on the edge; <code>false</code> otherwise
 */
public boolean isOnEdge(final Edge edge) {

    Vertex org;
    Vertex dest;
    double tx;
    double ty;
    double len;
    double lineA;
    double lineB;
    double lineC;
    boolean onEdge;

    org = edge.org();
```

```
        dest = edge.dest();

        tx = dest.xPos - org.xPos;
        ty = dest.yPos - org.yPos;
        len = Math.sqrt((tx * tx) + (ty * ty));

        lineA = ty / len;
        lineB = -tx / len;
        lineC = -((lineA * org.xPos) + (lineB * org.yPos));

        onEdge = Math.abs((lineA * this.xPos) + (lineB * this.yPos) + lineC) < EPSILON;

        return onEdge;
    }

    /**
     * Tests whether this point lies within the circle circumscribing a triangle.
     *
     * @param   pt1  the first point defining the triangle
     * @param   pt2  the second point defining the triangle
     * @param   pt3  the third point defining the triangle
     * @return  <code>true</code> if this point is inside the circumscribing circle; <code>
     *          false</code> if not
     */
    public boolean isInCircle(final Vertex pt1, final Vertex pt2, final Vertex pt3) {

        return ((pt1.lengthSquared() * triArea(pt2, pt3, this))
                - (pt2.lengthSquared() * triArea(pt1, pt3, this))
                + (pt3.lengthSquared() * triArea(pt1, pt2, this))
                - (this.lengthSquared() * triArea(pt1, pt2, pt3))) > 0;
    }

    /**
     * Generates the string representation of the vertex.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        return "(" + (float) this.xPos + "," + (float) this.yPos + ")";
    }

    /**
     * Computes twice the area of the oriented triangle <code>(pt1, pt2, pt3)</code> (the area is
     * positive if the triangle is oriented counterclockwise).
     *
     * @param   pt1  the first point
     * @param   pt2  the second point
     * @param   pt3  the third point
     * @return  twice the oriented area
     */
    public static double triArea(final Vertex pt1, final Vertex pt2, final Vertex pt3) {

        return ((pt2.xPos - pt1.xPos) * (pt3.yPos - pt1.yPos))
            - ((pt2.yPos - pt1.yPos) * (pt3.xPos - pt1.xPos));
    }

    /**
     * Tests whether the points of a triangle are in counterclockwise order.
     *
     * @param   pt1  the first point
     * @param   pt2  the second point
     * @param   pt3  the third point
     * @return  <code>true</code> if triangle (pt1, pt2, pt3) is in counterclockwise order
     */
    public static boolean isCcw(final Vertex pt1, final Vertex pt2, final Vertex pt3) {

        return triArea(pt1, pt2, pt3) > 0;
    }

    /**
     * Creates a vertex that lies at the center of the circle that circumscribes a triangle.
     *
     * @param   pt1  the first vertex of the triangle
     * @param   pt2  the second vertex of the triangle
     * @param   pt3  the third vertex of the triangle
     * @return  the vertex at the circumcenter
     */
    public static Vertex circumcenter(final Vertex pt1, final Vertex pt2, final Vertex pt3) {

        double len1Sq;
        double len2Sq;
        double len3Sq;
        double denom;
        double xNumer;
        double yNumer;
        Vertex center;
```

```
            len1Sq = pt1.lengthSquared();
            len2Sq = pt2.lengthSquared();
            len3Sq = pt3.lengthSquared();

            denom = 2
                * ((pt1.xPos * (pt2.yPos - pt3.yPos)) + (pt2.xPos * (pt3.yPos - pt1.yPos))
                    + (pt3.xPos * (pt1.yPos - pt2.yPos)));

            xNumer = (len1Sq * (pt2.yPos - pt3.yPos)) + (len2Sq * (pt3.yPos - pt1.yPos))
                + (len3Sq * (pt1.yPos - pt2.yPos));
            yNumer = (len1Sq * (pt3.xPos - pt2.xPos)) + (len2Sq * (pt1.xPos - pt3.xPos))
                + (len3Sq * (pt2.xPos - pt1.xPos));

            center = new Vertex(xNumer / denom, yNumer / denom);

            return center;
        }
    }
```

```
package com.srbenoit.math.delaunay;

/**
 * An intermediate data structure used in building the edges between regions in a Voronoi
 * diagram.
 */
class VorEdge {

    /** 'a' parameter in ax + by = c equation of the line */
    public double aParam = 0;

    /** 'b' parameter in ax + by = c equation of the line */
    public double bParam = 0;

    /** 'c' parameter in ax + by = c equation of the line */
    public double cParam = 0;

    /** the end points of this edge */
    public final Site[] endPoints;

    /** the sites this edge is the bisector of */
    public final Site[] reg;

    /** the edge number */
    public int edgeNumber;

    /**
     * Constructs a new <code>VorEdge</code>.
     */
    VorEdge() {

        this.endPoints = new Site[2];
        this.reg = new Site[2];
    }
}
```

```
package com.srbenoit.math.delaunay;

import java.awt.geom.Line2D;
import java.awt.geom.Path2D;
import java.util.Arrays;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

/**
 * A class to compute Voronoi diagrams.
 */
public class Voronoi {

    /** array index of left edge */
    private final static int LEFTEDGE = 0;

    /** array index of right edge */
    private final static int RIGHTEDGE = 1;

    /** a distance within which we consider points equal */
    private final double epsilon;

    /** the minimum distance between sites */
    private final double minSiteDist;

    /** the minimum X coordinate of any site */
    private double minX;

    /** the maximum X coordinate of any site */
```

```
private double maxX;

/** the minimum Y coordinate of any site */
private double minY;

/** the maximum Y coordinate of any site */
private double maxY;

/** the width of the bounding box */
private double width;

/** the height of the bounding box */
private double height;

/** a hash table to store half-edges */
private PQ pq;

/** a hash table to store ??? */
private EL el;

/** the number of sites */
private int nsites;

/** the square root of (the number of sites + 4) */
private int sqrtNsites;

/** the sorted list of sites */
private Site[] sites;

/** the index of the site being processed */
private int siteidx;

/** the number of vertices */
private int nvertices; // NOPMD

/** the number of edges */
private int nedges; // NOPMD

/** ??? */
private Site bottomsite;

/** the set of all edges in the graph */
private final List<GraphEdge> allEdges;

/**
 * Constructs a new <code>Voronoi</code>.
 *
 * @param  minSiteSpacing  the minimum distance between sites - any sites closer than this
 *                         distance will be considered a single site
 */
public Voronoi(final double minSiteSpacing) {

    this.siteidx = 0;
    this.sites = null;

    this.minSiteDist = minSiteSpacing;
    this.epsilon = minSiteSpacing / 100.0;
    this.nvertices = 0;
    this.nedges = 0;
    this.nsites = 0;
    this.sqrtNsites = 0;

    this.allEdges = new LinkedList<GraphEdge>();
}

/**
 * Generates the Voronoi diagram using Fortune's algorithm.
 *
 * @param   points  an array of coordinates for each site
 * @return  the list of graph edges in the Voronoi diagram
 */
public List<GraphEdge> generateVoronoi(final Vertex[] points) {

    GraphEdge edge;

    // Remove any prior run's output
    this.allEdges.clear();

    // Compute bounding box (only used to clip edges in the future)
    computeBounds(points);

    // Create a list of sites from the input list of points that is sorted
    // by increasing Y coordinate (and increasing X for values of equal Y)
    makeSortedSiteList(points);

    // Build the Voronoi diagram
    voronoiBuild();
```

```
        // Remove any edges whose end and start points are the same
        for (int i = this.allEdges.size() - 1; i >= 0; i--) {
            edge = this.allEdges.get(i);

            if (isSame(edge.xPos1, edge.yPos1, edge.xPos2, edge.yPos2)) {
                this.allEdges.remove(i);
            }
        }

        return this.allEdges;
    }

    /**
     * Computes the bounding box of a set of points.
     *
     * @param  points  the points
     */
    private void computeBounds(final Vertex[] points) {

        this.minX = Double.MAX_VALUE;
        this.minY = Double.MAX_VALUE;
        this.maxX = -Double.MAX_VALUE;
        this.maxY = -Double.MAX_VALUE;

        for (Vertex vert : points) {

            if (vert.xPos < this.minX) {
                this.minX = vert.xPos;
            }

            if (vert.xPos > this.maxX) {
                this.maxX = vert.xPos;
            }

            if (vert.yPos < this.minY) {
                this.minY = vert.yPos;
            }

            if (vert.yPos > this.maxY) {
                this.maxY = vert.yPos;
            }
        }

        this.height = this.maxY - this.minY;
        this.width = this.maxX - this.minX;
    }

    /**
     * Creates a list of <code>Site</code> objects whose coordinates are a given set of points, and
     * that is sorted in order of increasing Y (then increasing X if Y coordinates are the same)
     *
     * @param  points  the list of points
     */
    private void makeSortedSiteList(final Vertex[] points) {

        this.nsites = points.length;
        this.sqrtNsites = (int) Math.sqrt(this.nsites + 4);

        this.sites = new Site[this.nsites];
        this.siteidx = 0;

        for (int i = 0; i < this.nsites; i++) {
            this.sites[i] = new Site(points[i].xPos, points[i].yPos, i); // NOPMD SRB
        }

        Arrays.sort(this.sites);
    }

    /**
     * Gets the next site to be processed.
     *
     * @return  the next site; <code>null</code> if there are no more sites
     */
    private Site nextOne() {

        Site site;

        if (this.siteidx < this.nsites) {
            site = this.sites[this.siteidx];
            this.siteidx += 1;
        } else {
            site = null;
        }

        return site;
    }
```

```
/**
 * Constructs the bisector between two sites, as a line with equation ax + by = cParam.
 *
 * <p>The midpoint of the line is (s1 + s2)/2.
 *
 * <p>For a line changing more in Y than in X, we seek a formula for the bisector of the form
 * ax + y = cParam, and for a line changing more in X than in Y, we seek a formula for the
 * bisector of the form x + by = cParam.
 *
 * <p>For |dy|>|dx|, the slope of the bisector is -dx/dy, and so the equation of the line is
 *
 * <pre>
 * y - (yPos1+yPos2)/2 = (-dx/dy)[x - (xPos1+xPos2)/2]
 * (dx/dy)x + y = [(dx/dy)(xPos1+xPos2) + (yPos1+yPos2)]/2
 * (dx/dy)x + y = [((xPos2-xPos1)(xPos1+xPos2)/(yPos2-yPos1)) + (yPos1+yPos2)]/2
 * (dx/dy)x + y = [xPos2^2 - xPos1^2 + yPos2^2 - yPos1^2] / 2dy
 * </pre>
 *
 * <p>For |dx|>|dy|, the y-x slope of the bisector is -dy/dx, and so the equation of the line
 * is
 *
 * <pre>
 * x - (xPos1+xPos2)/2 = (-dy/dx)[y - (yPos1+yPos2)/2]
 * x + (dy/dx)y = [(dy/dx)(yPos1+yPos2) + (xPos1+xPos2)]/2
 * x + (dy/dx)y = [((yPos2-yPos1)/(xPos2-xPos1))(yPos1+yPos2) + (xPos1+xPos2)]/2
 * x + (dy/dx)y = [xPos2^2 - xPos1^2 + yPos2^2 - yPos1^2] / 2dx
 * </pre>
 *
 * @param    site1   the first site
 * @param    site2   the second site
 * @return   the bisector
 */
private VorEdge bisect(final Site site1, final Site site2) {

    double distX;
    double distY;
    double adx;
    double ady;
    double temp;
    VorEdge newedge;

    newedge = new VorEdge();

    // store the sites that this edge is bisecting
    newedge.reg[0] = site1;
    newedge.reg[1] = site2;

    // to begin with, there are no endpoints on the bisector - it goes to
    // infinity
    newedge.endPoints[0] = null;
    newedge.endPoints[1] = null;

    // vector from site 1 to site 2
    distX = site2.xPos - site1.xPos;
    distY = site2.yPos - site1.yPos;

    // make sure that the difference in positive
    adx = (distX > 0) ? distX : -distX;
    ady = (distY > 0) ? distY : -distY;

    temp = ((site2.xPos * site2.xPos) - (site1.xPos * site1.xPos) + (site2.yPos * site2.yPos)
            - (site1.yPos * site1.yPos)) * 0.5;

    if (adx > ady) {
        newedge.aParam = 1.0f;
        newedge.bParam = distY / distX;
        newedge.cParam = temp / distX;
    } else {
        newedge.bParam = 1.0f;
        newedge.aParam = distX / distY;
        newedge.cParam = temp / distY;
    }

    newedge.edgeNumber = this.nedges;
    this.nedges += 1;

    return newedge;
}

/**
 * ???
 *
 * @param    he   ???
 * @return   ???
 */
private Site leftreg(final Halfedge he) {
```

```
        if (he.elEdge == null) {
            return this.bottomsite;
        }

        return (he.elPm == LEFTEDGE) ? he.elEdge.reg[LEFTEDGE] : he.elEdge.reg[RIGHTEDGE];
    }

    /**
     * ???
     *
     * @param   leftSite    ???
     * @param   rightSite   ???
     * @param   xPos1       ???
     * @param   yPos1       ???
     * @param   xPos2       ???
     * @param   yPos2       ???
     */
    private void pushGraphEdge(final Site leftSite, final Site rightSite, final double xPos1,
        final double yPos1, final double xPos2, final double yPos2) {

        GraphEdge newEdge;

        newEdge = new GraphEdge();
        this.allEdges.add(newEdge);

        newEdge.xPos1 = xPos1;
        newEdge.yPos1 = yPos1;
        newEdge.xPos2 = xPos2;
        newEdge.yPos2 = yPos2;

        newEdge.site1 = leftSite.sitenbr;
        newEdge.site2 = rightSite.sitenbr;
    }

    /**
     * Clips a line to the bounding box.
     *
     * @param   edge  the edge to clip
     */
    private void clipLine(final VorEdge edge) {

        double pxmin, pxmax, pymin, pymax;
        Site site1;
        Site site2;
        double xPos1 = 0;
        double xPos2 = 0;
        double yPos1 = 0;
        double yPos2 = 0;

        xPos1 = edge.reg[0].xPos;
        xPos2 = edge.reg[1].xPos;
        yPos1 = edge.reg[0].yPos;
        yPos2 = edge.reg[1].yPos;

        // if the distance between the two points this line was created from is
        // less than the square root of 2, then ignore it
        if (Math.sqrt(((xPos2 - xPos1) * (xPos2 - xPos1)) + ((yPos2 - yPos1) * (yPos2 - yPos1)))
                < this.minSiteDist) {
            return;
        }

        pxmin = this.minX;
        pxmax = this.maxX;
        pymin = this.minY;
        pymax = this.maxY;

        if ((edge.aParam == 1.0) && (edge.bParam >= 0.0)) {
            site1 = edge.endPoints[1];
            site2 = edge.endPoints[0];
        } else {
            site1 = edge.endPoints[0];
            site2 = edge.endPoints[1];
        }

        if (edge.aParam == 1.0) {
            yPos1 = pymin;

            if ((site1 != null) && (site1.yPos > pymin)) {
                yPos1 = site1.yPos;
            }

            if (yPos1 > pymax) {
                yPos1 = pymax;
            }

            xPos1 = edge.cParam - (edge.bParam * yPos1);
```

```
            yPos2 = pymax;

            if ((site2 != null) && (site2.yPos < pymax)) {
                yPos2 = site2.yPos;
            }

            if (yPos2 < pymin) {
                yPos2 = pymin;
            }

            xPos2 = (edge.cParam) - ((edge.bParam) * yPos2);

            if (((xPos1 > pxmax) & (xPos2 > pxmax)) | ((xPos1 < pxmin) & (xPos2 < pxmin))) {
                return;
            }

            if (xPos1 > pxmax) {
                xPos1 = pxmax;
                yPos1 = (edge.cParam - xPos1) / edge.bParam;
            }

            if (xPos1 < pxmin) {
                xPos1 = pxmin;
                yPos1 = (edge.cParam - xPos1) / edge.bParam;
            }

            if (xPos2 > pxmax) {
                xPos2 = pxmax;
                yPos2 = (edge.cParam - xPos2) / edge.bParam;
            }

            if (xPos2 < pxmin) {
                xPos2 = pxmin;
                yPos2 = (edge.cParam - xPos2) / edge.bParam;
            }
        } else {
            xPos1 = pxmin;

            if ((site1 != null) && (site1.xPos > pxmin)) {
                xPos1 = site1.xPos;
            }

            if (xPos1 > pxmax) {
                xPos1 = pxmax;
            }

            yPos1 = edge.cParam - (edge.aParam * xPos1);
            xPos2 = pxmax;

            if ((site2 != null) && (site2.xPos < pxmax)) {
                xPos2 = site2.xPos;
            }

            if (xPos2 < pxmin) {
                xPos2 = pxmin;
            }

            yPos2 = edge.cParam - (edge.aParam * xPos2);

            if (((yPos1 > pymax) & (yPos2 > pymax)) | ((yPos1 < pymin) & (yPos2 < pymin))) {
                return;
            }

            if (yPos1 > pymax) {
                yPos1 = pymax;
                xPos1 = (edge.cParam - yPos1) / edge.aParam;
            }

            if (yPos1 < pymin) {
                yPos1 = pymin;
                xPos1 = (edge.cParam - yPos1) / edge.aParam;
            }

            if (yPos2 > pymax) {
                yPos2 = pymax;
                xPos2 = (edge.cParam - yPos2) / edge.aParam;
            }

            if (yPos2 < pymin) {
                yPos2 = pymin;
                xPos2 = (edge.cParam - yPos2) / edge.aParam;
            }
        }

        pushGraphEdge(edge.reg[0], edge.reg[1], xPos1, yPos1, xPos2, yPos2);
    }
```

```
/**
 * ???
 *
 * @param   edge   ???
 * @param   lr     ???
 * @param   site   ???
 */
private void endpoint(final VorEdge edge, final int lr, final Site site) {

    edge.endPoints[lr] = site;

    if (edge.endPoints[RIGHTEDGE - lr] == null) {
        return;
    }

    clipLine(edge);
}

/**
 * ???
 *
 * @param    he   ???
 * @return   ???
 */
private Site rightreg(final Halfedge he) {

    if (he.elEdge == null) {

        // if this half edge has no edge, return the bottom site (whatever
        // that is)
        return this.bottomsite;
    }

    // if the elPm field is zero, return the site 0 that this edge bisects,
    // otherwise return site number 1
    return (he.elPm == LEFTEDGE) ? he.elEdge.reg[RIGHTEDGE] : he.elEdge.reg[LEFTEDGE];
}

/**
 * Computes the distance between two sites.
 *
 * @param    site1   the first site
 * @param    site2   the second site
 * @return   the distance between the sites
 */
private double dist(final Site site1, final Site site2) {

    double distX;
    double distY;

    distX = site1.xPos - site2.xPos;
    distY = site1.yPos - site2.yPos;

    return Math.sqrt((distX * distX) + (distY * distY));
}

/**
 * Creates a new site where the HalfEdges el1 and el2 intersect.
 *
 * @param    el1   ???
 * @param    el2   ???
 * @return   ???
 */
private Site intersect(final Halfedge el1, final Halfedge el2) {

    VorEdge edge1;
    VorEdge edge2;
    VorEdge edge3;
    Halfedge edge;
    double det;
    double xint;
    double yint;
    boolean right_of_site;

    edge1 = el1.elEdge;
    edge2 = el2.elEdge;

    if ((edge1 == null) || (edge2 == null)) {
        return null;
    }

    // if the two edges bisect the same parent, return null
    if (edge1.reg[1] == edge2.reg[1]) {
        return null;
    }

    det = (edge1.aParam * edge2.bParam) - (edge1.bParam * edge2.aParam);
```

```
    if ((-1.0e-10 < det) && (det < 1.0e-10)) {
        return null;
    }

    xint = ((edge1.cParam * edge2.bParam) - (edge2.cParam * edge1.bParam)) / det;
    yint = ((edge2.cParam * edge1.aParam) - (edge1.cParam * edge2.aParam)) / det;

    if ((edge1.reg[1].yPos < edge2.reg[1].yPos)
            || ((edge1.reg[1].yPos == edge2.reg[1].yPos)
                && (edge1.reg[1].xPos < edge2.reg[1].xPos))) {
        edge = el1;
        edge3 = edge1;
    } else {
        edge = el2;
        edge3 = edge2;
    }

    right_of_site = xint >= edge3.reg[1].xPos;

    if ((right_of_site && (edge.elPm == LEFTEDGE))
            || (!right_of_site && (edge.elPm == RIGHTEDGE))) {
        return null;
    }

    // create a new site at the point of intersection - this is a new
    // vector event waiting to happen
    return new Site(xint, yint, -1);
}

/**
 * ???
 */
private void voronoiBuild() {

    Site newsite;
    Site bot;
    Site top;
    Site temp;
    Site p;
    Site v;
    Vertex newintstar = null;
    int pm;
    Halfedge lbnd;
    Halfedge rbnd;
    Halfedge llbnd;
    Halfedge rrbnd;
    Halfedge bisector;
    VorEdge edge;

    this.pq = new PQ(this.sqrtNsites, this.minY, this.height);
    this.el = new EL(this.sqrtNsites, this.minX, this.width);

    this.bottomsite = nextOne();
    newsite = nextOne();

    for (;;) {

        if (!this.pq.empty()) {
            newintstar = this.pq.min();
        }

        // if the lowest site has a smaller y value than the lowest vector
        // intersection, process the site otherwise process the vector
        // intersection
        if ((newsite != null)
                && (this.pq.empty() || (newsite.yPos < newintstar.yPos)
                    || ((newsite.yPos == newintstar.yPos)
                        && (newsite.xPos < newintstar.xPos)))) {

            /* new site is smallest - this is a site event */
            // get the first HalfEdge to the LEFT of the new site
            lbnd = this.el.leftbnd((newsite));

            // get the first HalfEdge to the RIGHT of the new site
            rbnd = lbnd.elRight;

            // if this half edge has no edge,bot = bottom site (whatever
            // that is)
            bot = rightreg(lbnd);

            // create a new edge that bisects
            edge = bisect(bot, newsite);

            // create a new HalfEdge, setting its elPm field to 0
            bisector = new Halfedge(edge, LEFTEDGE); // NOPMD SRB
```

```
        // insert this new bisector edge between the left and right
        // vectors in a linked list
        this.el.insert(lbnd, bisector);

        // if the new bisector intersects with the left edge,
        // remove the left edge's vertex, and put in the new one
        p = intersect(lbnd, bisector);

        if (p != null) {
            this.pq.delete(lbnd);
            this.pq.insert(lbnd, p, dist(p, newsite));
        }

        lbnd = bisector;

        // create a new HalfEdge, setting its elPm field to 1
        bisector = new Halfedge(edge, RIGHTEDGE); // NOPMD SRB

        // insert the new HE to the right of the original bisector
        // earlier in the IF statement
        this.el.insert(lbnd, bisector);

        // if this new bisector intersects with the new HalfEdge
        p = intersect(bisector, rbnd);

        if (p != null) {

            // push the HE into the ordered linked list of vertices
            this.pq.insert(bisector, p, dist(p, newsite));
        }

        newsite = nextOne();
    } else if (this.pq.empty()) {
        break;
    } else {
        /* intersection is smallest - this is a vector event */

        // pop the HalfEdge with the lowest vector off the ordered list
        // of vectors
        lbnd = this.pq.extractMin();

        // get the HalfEdge to the left of the above HE
        llbnd = lbnd.elLeft;

        // get the HalfEdge to the right of the above HE
        rbnd = lbnd.elRight;

        // get the HalfEdge to the right of the HE to the right of the
        // lowest HE
        rrbnd = rbnd.elRight;

        // get the Site to the left of the left HE which it bisects
        bot = leftreg(lbnd);

        // get the Site to the right of the right HE which it bisects
        top = rightreg(rbnd);

        v = lbnd.vertex;

        // get the vertex that caused this event set the vertex number
        // - couldn't do this searlier since we didn't know when it
        // would be processed
        v.sitenbr = this.nvertices;
        this.nvertices += 1;

        endpoint(lbnd.elEdge, lbnd.elPm, v);

        // set the endpoint of the left HalfEdge to be this vector
        endpoint(rbnd.elEdge, rbnd.elPm, v);

        // set the endpoint of the right HalfEdge to be this vector
        this.el.delete(lbnd); // mark the lowest HE for

        // deletion - can't delete yet because there might be pointers
        // to it in Hash Map
        this.pq.delete(rbnd);

        // remove all vertex events to do with the right HE
        this.el.delete(rbnd); // mark the right HE for

        // deletion - can't delete yet because there might be pointers
        // to it in Hash Map
        pm = LEFTEDGE; // set the pm variable to zero

        if (bot.yPos > top.yPos) {

            // if the site to the left of the event is higher than the
```

```
                    // Site to the right of it, then swap them and set the 'pm'
                    // variable to 1
                    temp = bot;
                    bot = top;
                    top = temp;
                    pm = RIGHTEDGE;
                }

                edge = bisect(bot, top);

                // create an Edge (or line) that is between the two Sites. This
                // creates the formula of the line, and assigns a line number
                // to it
                bisector = new Halfedge(edge, pm); // NOPMD SRB

                // create a HE from the Edge 'e', and make it point to that
                // edge with its elEdge field
                this.el.insert(llbnd, bisector);

                // insert the new bisector to the sright of the left HE
                endpoint(edge, RIGHTEDGE - pm, v);
                // set one endpoint to the new edge to be the vector point 'v'. If the site to the
                // left of this bisector is higher than the right Site, then this endpoint is put
                // in position 0; otherwise in pos 1

                // if left HE and the new bisector intersect, then delete sthe
                // left HE, and reinsert it
                p = intersect(llbnd, bisector);

                if (p != null) {
                    this.pq.delete(llbnd);
                    this.pq.insert(llbnd, p, dist(p, bot));
                }

                // if right HE and the new bisector intersect, then reinsert it
                p = intersect(bisector, rrbnd);

                if (p != null) {
                    this.pq.insert(bisector, p, dist(p, bot));
                }
            }
        }

        for (lbnd = this.el.leftEnd.elRight; lbnd != this.el.rightEnd; lbnd = lbnd.elRight) {
            edge = lbnd.elEdge;
            clipLine(edge);
        }
    }

    /**
     * Given a list of vertices and a list of voronoi edges, computes a set of polygons, each
     * representing a Voronoi region, then finds the vertex in that region and constructs a map
     * from vertex to region.
     *
     * @param   vertices  the list of vertices
     * @param   edges     the list of Voronoi edges
     * @return  the constructed map
     */
    public Map<Vertex, Path2D> makeVoronoiPolygons(final Vertex[] vertices,
        final List<GraphEdge> edges) {

        Map<Vertex, Path2D> map;

        map = new HashMap<Vertex, Path2D>(edges.size());

        // Scan through all edges
        for (int onEdge = 0; onEdge < edges.size(); onEdge++) {
            findPolygon(onEdge, vertices, edges, map, true);
            findPolygon(onEdge, vertices, edges, map, false);
        }

        return map;
    }

    /**
     * Follows an edge around the polygon to its left, connecting subsequent edges to form a
     * polygon, then adds that polygon to the map based on the vertex it contains.
     *
     * @param  onEdge    the edge
     * @param  vertices  the list of vertices
     * @param  edges     the list of edges
     * @param  map       the map from vertex to containing polygon
     * @param  isLeft    <code>true <code>to look for edges that follow the polygon to the left of
     *                   the starting edge; <code>false</code> to follow the polygon to the right
     */
    private void findPolygon(final int onEdge, final Vertex[] vertices,
        final List<GraphEdge> edges, final Map<Vertex, Path2D> map, final boolean isLeft) {
```

```
        GraphEdge root;
        GraphEdge terminal;
        Vertex start;
        Vertex end;
        Vertex prior;
        Vertex current;
        List<Vertex> list;
        double angle;
        double maxAngle;
        double dx1;
        double dx2;
        double dy1;
        double dy2;
        GraphEdge best = null;
        double bestX = 0;
        double bestY = 0;
        boolean finished;
        boolean found;
        int len;
        Line2D line;
        Path2D poly;

        root = edges.get(onEdge);
        list = new LinkedList<Vertex>();
        finished = false;

        // Create the initial two points in the polygon
        start = new Vertex(root.xPos1, root.yPos1);
        list.add(start);
        end = new Vertex(root.xPos2, root.yPos2);
        list.add(end);

        prior = start;
        current = end;
        terminal = root;

outer:
        for (;;) {

            // Look for edges that start where the current edge ends, then
            // identify the one whose endpoint lies to the left (right) of the
            // current edge and which makes the largest angle with the current
            // edge
            maxAngle = 0;
            dx1 = current.xPos - prior.xPos;
            dy1 = current.yPos - prior.yPos;
            found = false;

            for (GraphEdge test : edges) {

                if ((test == root) || (test == terminal)) {

                    // edge would trivially link back to its start point
                    continue;
                }

                if ((Math.abs(test.xPos1 - current.xPos) < this.epsilon)
                        && (Math.abs(test.yPos1 - current.yPos) < this.epsilon)) {

                    // Found a link
                    if ((Math.abs(test.xPos2 - start.xPos) < this.epsilon)
                            && (Math.abs(test.yPos2 - start.yPos) < this.epsilon)) {

                        // Edge closed on the starting vertex - finished
                        finished = true;

                        break outer;
                    }

                    if (isLeft
                            == isCcw(prior.xPos, prior.yPos, current.xPos, current.yPos,
                                test.xPos2, test.yPos2)) {

                        dx2 = test.xPos2 - test.xPos1;
                        dy2 = test.yPos2 - test.yPos1;

                        angle = Math.abs(Math.acos(
                                    ((dx1 * dx2) + (dy1 * dy2))
                                    / (Math.sqrt((dx1 * dx1) + (dy1 * dy1)))
                                    / (Math.sqrt((dx2 * dx2) + (dy2 * dy2)))));

                        if (angle > maxAngle) {
                            found = true;
                            maxAngle = angle;
                            best = test;
                            bestX = test.xPos2;
```

```
                            bestY = test.yPos2;
                        }
                    }
                } else if ((Math.abs(test.xPos2 - current.xPos) < this.epsilon)
                        && (Math.abs(test.yPos2 - current.yPos) < this.epsilon)) {

                    // Found a link
                    if ((Math.abs(test.xPos1 - start.xPos) < this.epsilon)
                            && (Math.abs(test.yPos1 - start.yPos) < this.epsilon)) {

                        // Edge closed on the starting vertex - finished
                        finished = true;

                        break outer;
                    }

                    if (isLeft
                            == isCcw(prior.xPos, prior.yPos, current.xPos, current.yPos,
                                test.xPos1, test.yPos1)) {

                        dx2 = test.xPos1 - test.xPos2;
                        dy2 = test.yPos1 - test.yPos2;
                        angle = Math.abs(Math.acos(
                                    ((dx1 * dx2) + (dy1 * dy2))
                                    / (Math.sqrt((dx1 * dx1) + (dy1 * dy1)))
                                    / (Math.sqrt((dx2 * dx2) + (dy2 * dy2)))));

                        if (angle > maxAngle) {
                            found = true;
                            maxAngle = angle;
                            best = test;
                            bestX = test.xPos1;
                            bestY = test.yPos1;
                        }
                    }
                }
            }

            if (found) {
                prior = current;
                current = new Vertex(bestX, bestY); // NOPMD SRB
                list.add(current);
                terminal = best;

            } else {
                break;
            }
        }

        if (!finished) {

            prior = end;
            current = start;

outer:
            for (;;) {

                // Look for edges that end where the current edge starts, then
                // identify the one whose endpoint lies to the left of the
                // current edge and which makes the largest angle with the
                // current edge
                maxAngle = 0;
                dx1 = current.xPos - prior.xPos;
                dy1 = current.yPos - prior.yPos;
                found = false;

                for (GraphEdge test : edges) {

                    if ((test == root) || (test == terminal)) {

                        // edge would trivially link back to its start point
                        continue;
                    }

                    if ((Math.abs(test.xPos1 - current.xPos) < this.epsilon)
                            && (Math.abs(test.yPos1 - current.yPos) < this.epsilon)) {

                        // Found a link
                        if ((Math.abs(test.xPos2 - end.xPos) < this.epsilon)
                                && (Math.abs(test.yPos2 - end.yPos) < this.epsilon)) {

                            // Edge closed on the starting vertex - finished
                            finished = true;

                            break outer;
                        }
```

```
                    if (isLeft
                            == isCcw(current.xPos, current.yPos, prior.xPos, prior.yPos,
                                test.xPos2, test.yPos2)) {

                        dx2 = test.xPos2 - test.xPos1;
                        dy2 = test.yPos2 - test.yPos1;
                        angle = Math.abs(Math.acos(
                                    ((dx1 * dx2) + (dy1 * dy2))
                                    / (Math.sqrt((dx1 * dx1) + (dy1 * dy1)))
                                    / (Math.sqrt((dx2 * dx2) + (dy2 * dy2)))));

                        if (angle > maxAngle) {
                            found = true;
                            maxAngle = angle;
                            best = test;
                            bestX = test.xPos2;
                            bestY = test.yPos2;
                        }
                    }
                } else if ((Math.abs(test.xPos2 - current.xPos) < this.epsilon)
                        && (Math.abs(test.yPos2 - current.yPos) < this.epsilon)) {

                    // Found a link
                    if ((Math.abs(test.xPos1 - end.xPos) < this.epsilon)
                            && (Math.abs(test.yPos1 - end.yPos) < this.epsilon)) {

                        // Edge closed on the starting vertex - finished
                        finished = true;

                        break outer;
                    }

                    if (isLeft
                            == isCcw(current.xPos, current.yPos, prior.xPos, prior.yPos,
                                test.xPos1, test.yPos1)) {

                        dx2 = test.xPos1 - test.xPos2;
                        dy2 = test.yPos1 - test.yPos2;
                        angle = Math.abs(Math.acos(
                                    ((dx1 * dx2) + (dy1 * dy2))
                                    / (Math.sqrt((dx1 * dx1) + (dy1 * dy1)))
                                    / (Math.sqrt((dx2 * dx2) + (dy2 * dy2)))));

                        if (angle > maxAngle) {
                            found = true;
                            maxAngle = angle;
                            best = test;
                            bestX = test.xPos1;
                            bestY = test.yPos1;
                        }
                    }
                }
            }

            if (found) {
                prior = current;
                current = new Vertex(bestX, bestY); // NOPMD SRB
                list.add(0, current);
                terminal = best;
            } else {
                finished = true;

                break;
            }
        }
    }

    if (finished) {
        Vertex vert1 = null;
        Vertex vert2 = null;

        len = list.size();

        if (len > 2) {

            poly = new Path2D.Double();

            for (int i = 1; i < len; i++) {
                vert1 = list.get(i - 1);
                vert2 = list.get(i);
                line = new Line2D.Double(vert1.xPos, vert1.yPos, vert2.xPos, vert2.yPos); // NOPMD SRB
                poly.append(line, true);
            }

            for (Vertex v : vertices) {

                if (poly.contains(v.xPos, v.yPos)) {
```

```
                        map.put(v, poly);
                    }
                }
            }
        }
    }

    /**
     * Computes twice the area of the oriented triangle <code>(xPos1, yPos1), (xPos2, yPos2), (x3,
     * y3)</code> (the area is positive if the triangle is oriented counterclockwise).
     *
     * @param   xPos1  the first point X coordinate
     * @param   yPos1  the first point Y coordinate
     * @param   xPos2  the second point X coordinate
     * @param   yPos2  the second point Y coordinate
     * @param   xPos3  the third point X coordinate
     * @param   yPos3  the third point Y coordinate
     * @return  twice the oriented area
     */
    public static final double triArea(final double xPos1, final double yPos1, final double xPos2,
        final double yPos2, final double xPos3, final double yPos3) {

        return ((xPos2 - xPos1) * (yPos3 - yPos1)) - ((yPos2 - yPos1) * (xPos3 - xPos1));
    }

    /**
     * Tests whether the points of a triangle are in counterclockwise order.
     *
     * @param   xPos1  the first point X coordinate
     * @param   yPos1  the first point Y coordinate
     * @param   xPos2  the second point X coordinate
     * @param   yPos2  the second point Y coordinate
     * @param   xPos3  the third point X coordinate
     * @param   yPos3  the third point Y coordinate
     * @return  <code>true</code> if triangle <code>(xPos1, yPos1), (xPos2, yPos2), (x3, y3)</code>
     *          is in counterclockwise order
     */
    public static final boolean isCcw(final double xPos1, final double yPos1, final double xPos2,
        final double yPos2, final double xPos3, final double yPos3) {

        return triArea(xPos1, yPos1, xPos2, yPos2, xPos3, yPos3) > 0;
    }

    /**
     * Tests whether two points are at the same location; two points are considered to be at the
     * same location when their X and Y coordinates both differ by less than <code>epsilon</code>.
     *
     * @param   xPos1  the first point X coordinate
     * @param   yPos1  the first point Y coordinate
     * @param   xPos2  the second point X coordinate
     * @param   yPos2  the second point Y coordinate
     * @return  <code>true</code> if the points are at the same location; <code>false</code> if not
     */
    private boolean isSame(final double xPos1, final double yPos1, final double xPos2,
        final double yPos2) {

        return (Math.abs(xPos2 - xPos1) < this.epsilon)
            && (Math.abs(yPos2 - yPos1) < this.epsilon);
    }
}
```

E.5 Filter Tree Management and Video Microscopy Analysis

E.5.1 Filter Tree Infrastructure (com.srbenoit.filter)

This package provides management and execution of trees of filters, which take some generalized input data and produce output data that can be used by downstream filters in the tree.

```
package com.srbenoit.filter;

import java.io.File;
import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * The base class for a filter. Filters take one or more inputs (each in a specified data format)
 * and produce outputs (again, in specified data formats). Filters draw their input data from a
 * <code>Pipe</code>, and may add their output data back to that <code>Pipe</code> for downstream
 * filters to draw from.
 */
public abstract class AbstractFilter extends LoggedObject {

    /** version number for serialization */
    private static final long serialVersionUID = -5289023315832548170L;

    /** a zero-length class array to use when finding constructors */
    private static final Class<?>[] CLASS_0;

    /** a zero-length array of objects to facilitate array construction */
    private static final Object[] OBJECT_0;

    /** a zero-length array of Strings to facilitate array construction */
    private static final String[] STRING_0;

    /** the file extension for TXT files */
    public final static String TXT_EXT = ".txt";

    /** the filename of the report file indicating step completion */
    public final static String REPORT = "report" + TXT_EXT;

    /** the human-friendly name of the filter */
    private final String name;

    /** the XML tag of the filter */
    private final String tag;

    /** a set of properties that can configure the filter */
    private final Properties properties;

    /** the current state of the filter */
    private FilterState state;

    /** the selection state of the filter */
    private boolean selected;

    /** the provisional state of the filter */
    private boolean provisional;

    /** the list of formats of required input data */
    protected final List<FilterInput> inputs;

    /** the list of the formats of output data */
    protected final List<FilterOutput> outputs;

    /** the renderer for the filter */
    private FilterRenderer renderer;

    static {
        CLASS_0 = new Class<?>[0];
        OBJECT_0 = new Object[0];
        STRING_0 = new String[0];
    }

    /**
     * Constructs a new <code>AbstractFilter</code>.
     *
     * @param  filterName  the name of the filter
     * @param  filterTag   the XML tag of the filter
     */
    public AbstractFilter(final String filterName, final String filterTag) {

        super();

        this.name = filterName;
        this.tag = filterTag;

        this.properties = new Properties();
        this.inputs = new ArrayList<FilterInput>(5);
        this.outputs = new ArrayList<FilterOutput>(5);
        this.state = FilterState.INERT;
        this.provisional = false;
```

```
        this.selected = false;
    }

    /**
     * Builds the renderer (must be called after inputs and outputs are configured).
     */
    protected void makeRenderer() {

        this.renderer = new FilterRenderer(this);
    }

    /**
     * Gets the renderer that will draw the font.
     *
     * @return  the renderer
     */
    public FilterRenderer getRenderer() {

        return this.renderer;
    }

    /**
     * Gets the no-argument constructor for a filter class.
     *
     * @param   clazz  the filter class
     * @return  the no-argument constructor for instances of that filter
     */
    public static Constructor<? extends AbstractFilter> getNoArgConstructor(
        final Class<? extends AbstractFilter> clazz) {

        Constructor<? extends AbstractFilter> constr;

        try {
            constr = clazz.getConstructor(CLASS_0);
        } catch (Exception e) {
            LOG.log(Level.WARNING,
                "Exception while getting filter constructor for '" + clazz.getName() + "'", e);
            constr = null;
        }

        return constr;
    }

    /**
     * Gets a new instance of a filter based on its class.
     *
     * @param   clazz  the filter class
     * @return  the new instance
     */
    public static AbstractFilter getInstance(final Class<? extends AbstractFilter> clazz) {

        Constructor<? extends AbstractFilter> constr;
        AbstractFilter instance;

        try {
            constr = clazz.getConstructor(CLASS_0);
            instance = constr.newInstance(OBJECT_0);
        } catch (Exception e) {
            LOG.log(Level.WARNING, "Exception while getting instance of '" + clazz.getName() + "'",
                e);
            instance = null;
        }

        return instance;
    }

    /**
     * Gets the filter name.
     *
     * @return  the filter name
     */
    public String getName() {

        return this.name;
    }

    /**
     * Gets the XML tag for the filter.
     *
     * @return  the XML tag
     */
    public String getTag() {

        return this.tag;
    }

    /**
```

```
     * Sets the value of a property.
     *
     * @param  key    the property key
     * @param  value  the value to set
     */
    public void setProperty(final String key, final String value) {

        this.properties.setProperty(key, value);
    }

    /**
     * Gets the value of a property.
     *
     * @param   key  the property key
     * @return  the value, or <code>null</code> if the value has not been set
     */
    public String getProperty(final String key) {

        return this.properties.getProperty(key);
    }

    /**
     * Gets the keys for all properties in the filter.
     *
     * @return  the list of keys
     */
    public String[] getPropertyKeys() {

        return this.properties.keySet().toArray(STRING_0);
    }

    /**
     * Gets the list of keys for all properties supported by the filter
     *
     * @return  the list of keys
     */
    public String[] getSupportedPropertyKeys() {

        return new String[0];
    }

    /**
     * Gets the number of input data formats required by this filter.
     *
     * @return  the number of input data objects
     */
    public int getNumInputs() {

        return this.inputs.size();
    }

    /**
     * Gets the number of output data formats generated by this filter (assuming only the required
     * inputs are provided - non-required inputs may be passed through to the output, increasing
     * their number).
     *
     * @return  the number of output data objects
     */
    public int getNumOutputs() {

        return this.outputs.size();
    }

    /**
     * Gets an input data format required by this filter.
     *
     * @param   index  the index of the format to retrieve
     * @return  the input format
     */
    public FilterInput getInputFormat(final int index) {

        return this.inputs.get(index);
    }

    /**
     * Gets an output data format generated by this filter.
     *
     * @param   index  the index of the format to retrieve
     * @return  the output format
     */
    public FilterOutput getOutputFormat(final int index) {

        return this.outputs.get(index);
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
```

```
     *
     * @return  the duplicated object
     */
    public abstract AbstractFilter duplicate();

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      the pipe from which to draw input data items and to which to add output
     *                    data items
     * @throws  FilterException  if the filter cannot complete
     */
    public abstract void filter(FilterTreeExecutor executor, Pipe pipe) throws FilterException;

    /**
     * Validates that the pipe contains all required input data items.
     *
     * @param   pipe  the pipe containing the input data items to validate
     * @throws  FilterException  if the inputs are invalid
     */
    protected void validateInputs(final Pipe pipe) throws FilterException {

        AbstractPipeItem item;

        if (pipe == null) {
            throw new FilterException("Pipe passed to validateInputs must not be null");
        } else {

            // Scan our required inputs testing for presence if each in the pipe
            for (FilterInput input : this.inputs) {
                item = pipe.get(input.getKey());

                if (item == null) {
                    throw new FilterException("Input data did not contain a '"
                        + input.type.getSimpleName() + "' input named '" + input.getKey() + "'");
                }

                if (!item.getClass().equals(input.type)) {
                    throw new FilterException("Input data did not contain a '"
                        + input.type.getSimpleName() + "' input named '" + input.getKey() + "'");
                }
            }
        }
    }

    /**
     * Tests whether another object is equal to this one. To be equal, the other object must be an
     * <code>AbstractObject</code>, must have the same tag and name and the superclass, inputs,
     * outputs, and children list objects must all be equal.
     *
     * @param   obj  the object to test
     * @return  <code>true</code> if the object is equal to this object
     */
    @Override public boolean equals(final Object obj) {

        AbstractFilter filter;
        boolean equal;

        if (obj instanceof AbstractFilter) {
            filter = (AbstractFilter) obj;

            equal = super.equals(obj) && this.name.equals(filter.getName())
                && this.tag.equals(filter.getTag()) && this.inputs.equals(filter.inputs)
                && this.outputs.equals(filter.outputs);
        } else {
            equal = false;
        }

        return equal;
    }

    /**
     * Gets the hash code for this object.
     *
     * @return  the hash code
     */
    @Override public int hashCode() {

        return super.hashCode() + this.name.hashCode() + this.tag.hashCode()
            + this.inputs.hashCode() + this.outputs.hashCode();
    }

    /**
     * Tests whether a completion report exists in the target directory.
     *
     * @param   dir  the directory in which to look for the report
```

```
     * @return  <code>true</code> if the report was present; <code>false</code> if not
     */
    protected boolean testForCompletionReport(final File dir) {

        return new File(dir, REPORT).exists();
    }

    /**
     * Sets the state of the filter.
     *
     * @param  newState  the new state
     */
    public void setState(final FilterState newState) {

        this.state = newState;
    }

    /**
     * Gets the state of the filter.
     *
     * @return  the state
     */
    public FilterState getState() {

        return this.state;
    }

    /**
     * Sets the selection state of the filter.
     *
     * @param  isSelected  <code>true</code> if the filter is selected; <code>false</code> if not
     */
    public void setSelected(final boolean isSelected) {

        this.selected = isSelected;
    }

    /**
     * Gets the selection state of the filter.
     *
     * @return  <code>true</code> if the filter is selected; <code>false</code> if not
     */
    public boolean isSelected() {

        return this.selected;
    }

    /**
     * Sets the provisional state of the filter.
     *
     * @param  isProvisional  <code>true</code> if the filter is provisional; <code>false</code> if
     *                        not
     */
    public void setProvisional(final boolean isProvisional) {

        this.provisional = isProvisional;
    }

    /**
     * Gets the provisional state of the filter.
     *
     * @return  <code>true</code> if the filter is provisional; <code>false</code> if not
     */
    public boolean isProvisional() {

        return this.provisional;
    }
}

package com.srbenoit.filter;

import java.io.File;
import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.log.LoggedObject;

/**
 * A item that can flow into and out of a pipe.
 */
public abstract class AbstractPipeItem extends LoggedObject {

    /** a zero-length class array to use when finding constructors */
    private static final Class<?>[] CLASS_3;

    /** a zero-length array of objects to facilitate array construction */
```

```
private static final Object[] OBJECT_3;

/** the pipe this item belongs to */
private final transient Pipe pipe;

/** a unique key for the item (unique within the pipe) */
private final transient String key;

/** the label for the item (a human friendly name) */
private final transient String label;

/** the files where this pipe item is persisted */
private final List<PipeItemFileInfo> files;

static {
    CLASS_3 = new Class<?>[] { String.class, String.class, Pipe.class };
    OBJECT_3 = new Object[3];
}

/**
 * Gets the constructor for a pipe item class that takes two <code>String</code> arguments and
 * a <code>Pipe</code> argument.
 *
 * @param   clazz  the filter class
 * @return  the constructor for instances of that filter
 */
public static Constructor<? extends AbstractPipeItem> getConstructor(
    final Class<? extends AbstractPipeItem> clazz) {

    Constructor<? extends AbstractPipeItem> constr;

    try {
        constr = clazz.getConstructor(CLASS_3);
    } catch (Exception e) {
        LOG.log(Level.WARNING, "Unable to get constructor for " + clazz.getName(), e);
        constr = null;
    }

    return constr;
}

/**
 * Gets a new instance of a pipe item based on its class and string key and label.
 *
 * @param   clazz     the filter class
 * @param   theKey    the string key
 * @param   theLabel  the string label
 * @param   thePipe   the pipe in which the item is installed
 * @return  the new instance
 */
public static AbstractPipeItem getInstance(final Class<? extends AbstractPipeItem> clazz,
    final String theKey, final String theLabel, final Pipe thePipe) {

    Constructor<? extends AbstractPipeItem> constr;
    AbstractPipeItem instance;

    try {
        constr = clazz.getConstructor(CLASS_3);

        synchronized (OBJECT_3) {
            OBJECT_3[0] = theKey;
            OBJECT_3[1] = theLabel;
            OBJECT_3[2] = thePipe;
            instance = constr.newInstance(OBJECT_3);
        }
    } catch (Exception e) {
        LOG.log(Level.WARNING, "Unable to create instance of " + clazz.getName(), e);
        instance = null;
    }

    return instance;
}

/**
 * Constructs a new <code>AbstractPipeItem</code>.
 *
 * @param  theKey    the unique key for the pipe item
 * @param  theLabel  the label for the item (a human friendly name)
 * @param  thePipe   the pipe in which this item is installed
 */
public AbstractPipeItem(final String theKey, final String theLabel, final Pipe thePipe) {

    super();

    if (theKey == null) {
        throw new IllegalArgumentException("Key may not be null");
    }
```

```
        if (theLabel == null) {
            throw new IllegalArgumentException("Label may not be null");
        }

        if (thePipe == null) {
            throw new IllegalArgumentException("Pipe may not be null");
        }

        this.key = theKey;
        this.label = theLabel;
        this.pipe = thePipe;

        this.files = new ArrayList<PipeItemFileInfo>(10);
    }

    /**
     * Gets the unique key of the item. This will translate into the directory name in which the
     * item's files are stored, so it must be a valid directory name and should be meaningful in
     * the sense that the directory name should indicate its contents. For example, an item
     * containing a series of raw images might have a key "raw_images" and a label "Raw image files
     * (TIF format)".
     *
     * @return  the key
     */
    public String getKey() {

        return this.key;
    }

    /**
     * Gets the human-friendly name of the item. Think of this as a variable name, as opposed to
     * its data type (see <code>typeName</code>).
     *
     * @return  the label
     */
    public String getLabel() {

        return this.label;
    }

    /**
     * Gets the pipe this item is installed in.
     *
     * @return  the pipe
     */
    public Pipe getPipe() {

        return this.pipe;
    }

    /**
     * Gets a human-friendly name for the data type. For example, a list of sets of images
     * representing a time series of z-planes might return "Multi-plane image sequence".
     *
     * @return  the name of the data type this item represents
     */
    public abstract String typeName();

    /**
     * Resets the pipe item to a virgin (empty) state.
     */
    public abstract void reset();

    /**
     * Saves the item to a filesystem.
     *
     * @param   executor  the executor that is saving the pipe
     * @param   startPct  the starting progress percentage for the save operation
     * @param   endPct    the ending progress percentage for the save operation
     * @return  <code>true</code> on successful save; <code>false</code> on failure
     */
    public abstract boolean save(final FilterTreeExecutor executor, final int startPct,
        final int endPct);

    /**
     * Loads the item from the filesystem.
     *
     * @return  <code>true</code> if the load succeeded; <code>false</code> if not
     */
    public abstract boolean load();

    /**
     * Gets a subdirectory where this pipe item's data can be stored.
     *
     * @return  the subdirectory
     */
```

```
    public File getSubdir() {

        return new File(this.pipe.getDir(), this.key);
    }

    /**
     * Gets the list of the files where this pipe item's data is stored
     *
     * @return  the list of files (if dirty, this list may be incomplete)
     */
    public PipeItemFileInfo[] getFiles() {

        return this.files.toArray(new PipeItemFileInfo[0]);
    }

    /**
     * Gets a single file information object.
     *
     * @param   index  the index of the object
     * @return  the <code>PipeItemFileInfo</code>
     */
    public PipeItemFileInfo getFile(final int index) {

        return this.files.get(index);
    }

    /**
     * Adds a file information object.
     *
     * @param  info  the <code>PipeItemFileInfo</code> to add
     */
    public void addFile(final PipeItemFileInfo info) {

        this.files.add(info);
    }

    /**
     * Removes a file information object.
     *
     * @param  info  the <code>PipeItemFileInfo</code> to remove
     */
    public void removeFile(final PipeItemFileInfo info) {

        this.files.remove(info);
    }

    /**
     * Removes all but the first file in the files list.
     */
    public void removeAllFilesButFirst() {

        while (this.files.size() > 1) {
            this.files.remove(1);
        }
    }
}

package com.srbenoit.filter;

import java.io.File;
import javax.swing.JFileChooser;
import javax.swing.SwingUtilities;

/**
 * A class that displays a file chooser dialog in the AWT event thread.
 */
public class DirectoryAsker implements Runnable {

    /** the dialog title */
    private transient String title;

    /** the selected directory */
    private transient File dir;

    /**
     * Shows the dialog, waits for the user response, and returns the selection.
     *
     * @param   dialogTitle  the dialog title
     * @return  the selected directory, or <code>null</code> if the user canceled
     */
    public File askDir(final String dialogTitle) {

        this.title = dialogTitle;
        this.dir = null;

        try {
            SwingUtilities.invokeAndWait(this);
```

```
        } catch (Exception e) {
            this.dir = null;
        }

        return this.dir;
    }

    /**
     * Constructs the user interface in the background and gets the user's selection.
     */
    public void run() {

        JFileChooser chooser;
        int result;

        chooser = new JFileChooser();
        chooser.setDialogTitle(this.title);
        chooser.setMultiSelectionEnabled(false);
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

        result = chooser.showOpenDialog(null);

        if (result == JFileChooser.APPROVE_OPTION) {
            this.dir = chooser.getSelectedFile();
        } else {
            this.dir = null;
        }
    }
}
```

```
package com.srbenoit.filter;

/**
 * An exception thrown by filters if they cannot complete.
 */
public class FilterException extends Exception {

    /** version number for serialization */
    private static final long serialVersionUID = -7198057168424376761L;

    /**
     * Constructs a new <code>FilterException</code> with <code>null</code> as its detail message.
     */
    public FilterException() {

        super();
    }

    /**
     * Constructs a new <code>FilterException</code> with the specified detail message. The cause
     * is not initialized, and may subsequently be initialized by a call to {@link #initCause}.
     *
     * @param  message  the detail message
     */
    public FilterException(final String message) {

        super(message);
    }

    /**
     * Constructs a new <code>FilterException</code> with the specified detail message and cause.
     * the detail message
     *
     * @param  message  the detail message
     * @param  cause    the cause
     */
    public FilterException(final String message, final Throwable cause) {

        super(message, cause);
    }

    /**
     * Constructs a new <code>FilterException</code> with the specified cause and a detail message
     * of <code>(cause==null ? null : cause.toString())</code> (which typically contains the class
     * and detail message of <code>cause</code> ).
     *
     * @param  cause  the cause
     */
    public FilterException(final Throwable cause) {

        super(cause);
    }
}
```

```
package com.srbenoit.filter;

import java.awt.image.BufferedImage;
```

```
import com.srbenoit.filter.items.ByteArrayPipeItem;
import com.srbenoit.filter.items.ImageArrayPipeItem;
import com.srbenoit.filter.items.PointSetArrayPipeItem;
import com.srbenoit.filter.items.StringPipeItem;
import com.srbenoit.filter.items.TimeSeriesPipeItem;
import com.srbenoit.filter.items.TrajectoryListPipeItem;
import com.srbenoit.util.ResourceLoader;

/**
 * A specification for an input required by a filter.
 */
public class FilterInput {

    /** the data type of the input */
    public final Class<? extends AbstractPipeItem> type;

    /** a description of the input */
    public final String description;

    /** the key of the pipe item the filter will actually take its input from */
    private String actualKey;

    /** the icon that represents the output */
    private final BufferedImage icon;

    /**
     * Constructs a new <code>FilterInput</code>.
     *
     * @param  clazz  the data type of the input
     * @param  desc   a description of the input
     */
    public FilterInput(final Class<? extends AbstractPipeItem> clazz, final String desc) {

        BufferedImage img;

        if (clazz == null) {
            throw new IllegalArgumentException("input data type cannot be null");
        }

        if (desc == null) {
            throw new IllegalArgumentException("input data description cannot be null");
        }

        this.type = clazz;
        this.description = desc;

        this.actualKey = null;

        if (clazz.equals(ByteArrayPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/binary.png");
        } else if (clazz.equals(ImageArrayPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/images.png");
        } else if (clazz.equals(PointSetArrayPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/point_set.png");
        } else if (clazz.equals(StringPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/text.png");
        } else if (clazz.equals(TimeSeriesPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/time_series.png");
        } else if (clazz.equals(TrajectoryListPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/trajectories.png");
        } else {
            img = null;
        }

        if (img == null) {
            this.icon = new BufferedImage(24, 16, BufferedImage.TYPE_INT_RGB);
        } else {
            this.icon = img;
        }
    }

    /**
     * Sets the key of the pipe item from which the input will be taken.
     *
     * @param  key  the key
     */
    public void setKey(final String key) {

        this.actualKey = key;
    }

    /**
     * Gets the key of the pipe item from which the input will be taken.
     *
     * @return  the key
     */
    public String getKey() {
```

```
        return this.actualKey;
    }

    /**
     * Tests whether another object is equal to this <code>FilterInput</code>. In order to be
     * considered equal, the object must be a <code>FilterInput</code> and must have the same
     * values for all fields.
     *
     * @param   obj  the object to test
     * @return  <code>true</code> if the object is equal to this <code>FilterInput</code>; <code>
     *          false</code> otherwise
     */
    @Override public boolean equals(final Object obj) {

        FilterInput other;
        boolean equal;

        if (obj instanceof FilterInput) {
            other = (FilterInput) obj;
            equal = other.type.equals(this.type) && other.description.equals(this.description)
                && ((this.actualKey == null) ? (other.getKey() == null)
                                             : this.actualKey.equals(other.getKey()));
        } else {
            equal = false;
        }

        return equal;
    }

    /*
     * Gets the hash code for the object.
     *
     * @return the hash code
     */
    @Override public int hashCode() {

        return this.type.hashCode() + description.hashCode()
            + ((this.actualKey == null) ? 0 : this.actualKey.hashCode());
    }

    /**
     * Gets a 24x16 icon representing the filter input data type
     *
     * @return  the icon
     */
    public BufferedImage getIcon() {

        return this.icon;
    }
}

package com.srbenoit.filter;

import java.awt.image.BufferedImage;
import com.srbenoit.filter.items.ByteArrayPipeItem;
import com.srbenoit.filter.items.ImageArrayPipeItem;
import com.srbenoit.filter.items.PointSetArrayPipeItem;
import com.srbenoit.filter.items.StringPipeItem;
import com.srbenoit.filter.items.TimeSeriesPipeItem;
import com.srbenoit.filter.items.TrajectoryListPipeItem;
import com.srbenoit.util.ResourceLoader;

/**
 * A specification for an output generated by a filter.
 */
public class FilterOutput {

    /** the data type of the output */
    public final Class<? extends AbstractPipeItem> type;

    /** a description of the output */
    public final String description;

    /** the default key of the pipe item the filter will add to the pipe */
    public final String defaultKey;

    /** the actual key of the pipe item the filter will add to the pipe */
    private String actualKey;

    /** the icon that represents the output */
    private final BufferedImage icon;

    /**
     * Constructs a new <code>FilterOutput</code>.
     *
     * @param  clazz   the data type of the output
```

```
     * @param  desc    a description of the output
     * @param  defKey  the default key of the pipe item the filter will add to the pipe
     */
    public FilterOutput(final Class<? extends AbstractPipeItem> clazz, final String desc,
        final String defKey) {

        BufferedImage img;

        if (clazz == null) {
            throw new IllegalArgumentException("output data type cannot be null");
        }

        if (desc == null) {
            throw new IllegalArgumentException("output data description cannot be null");
        }

        if (defKey == null) {
            throw new IllegalArgumentException("output default key cannot be null");
        }

        this.type = clazz;
        this.description = desc;
        this.defaultKey = defKey;
        this.actualKey = defKey;

        if (clazz.equals(ByteArrayPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/binary.png");
        } else if (clazz.equals(ImageArrayPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/images.png");
        } else if (clazz.equals(PointSetArrayPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/point_set.png");
        } else if (clazz.equals(StringPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/text.png");
        } else if (clazz.equals(TimeSeriesPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/time_series.png");
        } else if (clazz.equals(TrajectoryListPipeItem.class)) {
            img = ResourceLoader.loadImage(FilterOutput.class, "icons/trajectories.png");
        } else {
            img = null;
        }

        if (img == null) {
            this.icon = new BufferedImage(24, 16, BufferedImage.TYPE_INT_RGB);
        } else {
            this.icon = img;
        }
    }

    /**
     * Sets the key of the pipe item from which the input will be taken.
     *
     * @param  key  the key
     */
    public void setKey(final String key) {

        this.actualKey = key;
    }

    /**
     * Gets the key of the pipe item from which the input will be taken.
     *
     * @return  the key
     */
    public String getKey() {

        return this.actualKey;
    }

    /**
     * Tests whether another object is equal to this <code>FilterOutput</code>. In order to be
     * considered equal, the object must be a <code>FilterOutput</code> and must have the same
     * values for all fields.
     *
     * @param   obj  the object to test
     * @return  <code>true</code> if the object is equal to this <code>FilterOutput</code>; <code>
     *          false</code> otherwise
     */
    @Override public boolean equals(final Object obj) {

        FilterOutput other;
        boolean equal;

        if (obj instanceof FilterOutput) {
            other = (FilterOutput) obj;
            equal = other.type.equals(this.type) && other.description.equals(this.description)
                && other.defaultKey.equals(this.defaultKey)
                && ((this.actualKey == null) ? (other.getKey() == null)
```

```
                                                : this.actualKey.equals(other.getKey()));
        } else {
            equal = false;
        }

        return equal;
    }

    /*
     * Gets the hash code for the object.
     *
     * @return the hash code
     */
    @Override public int hashCode() {

        return this.type.hashCode() + description.hashCode() + defaultKey.hashCode()
            + ((this.actualKey == null) ? 0 : this.actualKey.hashCode());
    }

    /**
     * Gets a 24x16 icon representing the filter output data type
     *
     * @return  the icon
     */
    public BufferedImage getIcon() {

        return this.icon;
    }
}

package com.srbenoit.filter;

import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Rectangle;
import com.srbenoit.ui.UIUtilities;

/**
 * A renderer that draws a graphical representation of a filter.
 */
public class FilterRenderer {

    /** insets between filter box and inner content */
    public final static int INSETS = 4;

    /** length of stub */
    public final static int STUB = 2;

    /** the shadow color */
    private static final Color SHADOW;

    /** the shadow color */
    private static final Color HIGHLIGHT;

    /** the shadow color */
    private static final Color BACKGROUND;

    /** the shadow color */
    private static final Color SELECTED;

    /** the shadow color */
    private static final Color RUNNING;

    /** the shadow color */
    private static final Color COMPLETED;

    /** the font to use to render the filter label */
    private final Font font;

    /** the filter to render */
    private final AbstractFilter filter;

    /** the bounding rectangle for the filter */
    private final Rectangle bounds;

    /** the X coordinate (relative to left edge) of each input */
    private final int[] inputX;

    /** the Y coordinate (relative to top edge) of each output */
    private final int[] outputY;

    static {
        BACKGROUND = new Color(200, 200, 180);
        HIGHLIGHT = new Color(255, 255, 220);
        SHADOW = new Color(150, 150, 100);
```

```
        SELECTED = new Color(220, 220, 200);
        RUNNING = new Color(200, 200, 255);
        COMPLETED = new Color(200, 255, 180);
    }

    /**
     * Constructs a new <code>FilterRenderer</code>.
     *
     * @param  theFilter  the filter to render
     */
    public FilterRenderer(final AbstractFilter theFilter) {

        Graphics grx;
        FontMetrics met;
        int width;
        int boxHeight;
        int perOut;
        int outHeight;
        int height;

        this.filter = theFilter;
        this.font = new Font("SansSerif", Font.PLAIN, 12);

        this.inputX = new int[theFilter.getNumInputs()];
        this.outputY = new int[theFilter.getNumOutputs()];

        grx = UIUtilities.getGraphics();
        grx.setFont(this.font);
        met = grx.getFontMetrics();

        width = 2 + INSETS + met.stringWidth(theFilter.getName()) + INSETS + 2;

        if (theFilter.getNumInputs() > 0) {

            for (int i = 0; i < theFilter.getNumInputs(); i++) {
                this.inputX[i] = width + 12 + (28 * i);
            }

            width += 28 * theFilter.getNumInputs();
        }

        if (theFilter.getNumOutputs() > 0) {
            width += 28;
        }

        boxHeight = 2 + INSETS + met.getAscent() + met.getDescent() + INSETS + 2;

        perOut = ((met.getHeight() > 16) ? met.getHeight() : 16) + 6;
        outHeight = (perOut * theFilter.getNumOutputs()) + 6;

        height = (boxHeight > outHeight) ? boxHeight : outHeight;

        for (int i = 0; i < theFilter.getNumOutputs(); i++) {
            this.outputY[i] = (perOut * (i + 1)) - 4;
        }

        if (theFilter.getNumInputs() > 0) {
            height += STUB;

            for (int i = 0; i < theFilter.getNumOutputs(); i++) {
                this.outputY[i] += STUB;
            }
        }

        this.bounds = new Rectangle(0, 0, width, height);
    }

    /**
     * Sets the location of the filter's rendered box.
     *
     * @param  xPos  the X coordinate of the left edge
     * @param  yPos  the Y coordinate of the top edge
     */
    public void setLocation(final int xPos, final int yPos) {

        this.bounds.x = xPos;
        this.bounds.y = yPos;
    }

    /**
     * Gets a copy of the bounding rectangle for the filter.
     *
     * @return  the copy of the bounding rectangle (clients are free to alter this rectangle, but
     *          changes will not affect the bounds of the filter)
     */
    public Rectangle getBounds() {
```

```
    return new Rectangle(this.bounds);
}

/**
 * Gets the X coordinate, relative to the left edge of the bounding rectangle, of a particular
 * input's stub.
 *
 * @param   index  the index of the input
 * @return  the X coordinate
 */
public int getInputX(final int index) {

    return this.inputX[index];
}

/**
 * Gets the Y coordinate, relative to the top edge of the bounding rectangle, of a particular
 * output's stub.
 *
 * @param   index  the index of the output
 * @return  the Y coordinate
 */
public int getOutputY(final int index) {

    return this.outputY[index];
}

/**
 * Draws the filter.
 *
 * @param  grx  the <code>Graphics</code> on which to draw
 */
public void draw(final Graphics grx) {

    int left;
    int right;
    int top;
    int bottom;
    int xPos;
    int yPos;
    FontMetrics met;

    grx.setFont(this.font);
    met = grx.getFontMetrics();

    left = this.bounds.x;
    right = this.bounds.x + this.bounds.width - 1;

    if (this.filter.getNumOutputs() > 0) {
        right -= 28;
    }

    top = this.bounds.y + STUB;
    bottom = this.bounds.y + this.bounds.height - 1;

    if (filter.isSelected()) {
        grx.setColor(SELECTED);
    } else {

        switch (filter.getState()) {

        case RUNNING:
            grx.setColor(RUNNING);
            break;

        case COMPLETED:
            grx.setColor(COMPLETED);
            break;

        default:
            grx.setColor(BACKGROUND);
            break;
        }
    }

    grx.fillRect(left, top, right - left, bottom - top);

    grx.setColor(SHADOW);
    grx.drawLine(left + 2, bottom - 1, right - 1, bottom - 1);
    grx.drawLine(right - 1, top + 2, right - 1, bottom - 1);

    grx.setColor(HIGHLIGHT);
    grx.drawLine(left + 1, top + 1, right - 2, top + 1);
    grx.drawLine(left + 1, top + 1, left + 1, bottom - 2);

    if (filter.isProvisional()) {
```

```
            grx.setColor(Color.GRAY);
        } else {
            grx.setColor(Color.BLACK);
        }

        grx.drawRect(left, top, right - left, bottom - top);

        grx.drawString(this.filter.getName(), left + INSETS, top + INSETS + met.getAscent());

        // Draw the inputs
        yPos = top + INSETS;

        for (int i = 0; i < this.filter.getNumInputs(); i++) {
            xPos = bounds.x + this.getInputX(i);
            grx.drawImage(this.filter.getInputFormat(i).getIcon(), xPos - 12, yPos, null);
            grx.drawLine(xPos, this.bounds.y, xPos, top);
        }

        // Draw the outputs
        xPos = right + 4;
        yPos = this.bounds.y;

        if (this.filter.getNumInputs() > 0) {
            yPos += STUB;
        }

        for (int i = 0; i < this.filter.getNumOutputs(); i++) {
            yPos = bounds.y + this.getOutputY(i);
            grx.drawImage(this.filter.getOutputFormat(i).getIcon(), xPos, yPos - 18, null);
            grx.drawLine(right, yPos, this.bounds.x + this.bounds.width, yPos);
        }
    }
}

package com.srbenoit.filter;

/**
 * A list of possible filter states.
 */
public enum FilterState {

    /** inert state (not yet run) */
    INERT,

    /** running state */
    RUNNING,

    /** completed state */
    COMPLETED;
}

package com.srbenoit.filter;

import java.util.ArrayList;
import java.util.List;

/**
 * A filter tree, consisting of an ordered list of filters, each of which has its actual input and
 * output tags set, as well as any parameter values.
 */
public class FilterTree {

    /** the filters in the tree */
    private final List<AbstractFilter> filters;

    /**
     * Constructs a new <code>FilterTree</code>.
     */
    public FilterTree() {

        this.filters = new ArrayList<AbstractFilter>(20);
    }

    /**
     * Gets the number of filters in the tree.
     *
     * @return  the number of filters
     */
    public int getNumFilters() {

        return this.filters.size();
    }

    /**
     * Gets a particular filter.
     *
     * @param   index  the index of the filter to retrieve
```

```
     * @return  the filter
     */
    public AbstractFilter getFilter(final int index) {

        return this.filters.get(index);
    }

    /**
     * Adds a particular filter.
     *
     * @param  filter  the filter to add
     */
    public void addFilter(final AbstractFilter filter) {

        this.filters.add(filter);
    }

    /**
     * Adds a particular filter at a specified position in the list, shifting the filter in that
     * position and all subsequent filters down one position.
     *
     * @param  index   the position at which to insert the filter
     * @param  filter  the filter to add
     */
    public void addFilter(final int index, final AbstractFilter filter) {

        this.filters.add(index, filter);
    }

    /**
     * Removes a particular filter.
     *
     * @param   index  the index of the filter to remove
     * @return  the removed filter
     */
    public AbstractFilter removeFilter(final int index) {

        return this.filters.remove(index);
    }
}

package com.srbenoit.filter;

import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.SwingWorker;
import com.srbenoit.log.LogMgr;

/**
 * A subclass of <code>SwingWorker</code> that executes the filters in a filter tree and allows
 * filters to interact with a GUI as they proceed.
 */
public class FilterTreeExecutor extends SwingWorker<Object, Object> {

    /** the working directory for filters in the tree */
    private transient File dir;

    /** the filter tree to execute */
    private final transient FilterTree tree;

    /** the panel that renders the filter tree */
    private final transient FilterTreePanel panel;

    /** a list of listeners registered to receive progress notifications */
    private final transient List<FilterTreeExecutorListener> listeners;

    /** a log to which to write diagnostic messages */
    protected static final Logger LOG;

    static {
        LOG = LogMgr.getLogger();
    }

    /**
     * Constructs a new <code>FilterTreeExecutor</code>.
     *
     * @param  theTree   the filter tree to execute
     * @param  thePanel  the panel that renders the filter tree
     */
    public FilterTreeExecutor(final FilterTree theTree, final FilterTreePanel thePanel) {

        super();

        this.tree = theTree;
```

```
        this.panel = thePanel;

        this.dir = null;
        this.listeners = new ArrayList<FilterTreeExecutorListener>(2);
    }

    /**
     * Adds a listener that will be notified of progress updates in the execution of the filter
     * tree.
     *
     * @param  listener  the listener to add
     */
    public void addListener(final FilterTreeExecutorListener listener) {

        synchronized (this.listeners) {
            this.listeners.add(listener);
        }

        addPropertyChangeListener(listener);
    }

    /**
     * Removes a listener that was previously registered with <code>addListener</code>.
     *
     * @param  listener  the listener to remove
     */
    public void removeListener(final FilterTreeExecutorListener listener) {

        removePropertyChangeListener(listener);

        synchronized (this.listeners) {
            this.listeners.remove(listener);
        }
    }

    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * <p>This method is executed only once, and is executed in a background thread.
     *
     * @return  the computed result
     */
    @Override protected Object doInBackground() {

        Pipe pipe;
        List<AbstractFilter> todo;
        AbstractFilter filter;
        int count;
        boolean done;
        boolean valid;
        boolean found;
        boolean ready;
        FilterInput inp;
        FilterOutput out;

        // Get the base directory
        this.dir = new DirectoryAsker().askDir("Source Directory");

        if (this.dir != null) {
            pipe = new Pipe(this.dir);
            pipe.load(); // Load whatever has been done already

            // Build the to-do list of filters
            count = this.tree.getNumFilters();
            todo = new ArrayList<AbstractFilter>(count);

            valid = true;

            for (int i = 0; i < count; i++) {

                if (isCancelled()) {
                    valid = false;

                    break;
                }

                filter = this.tree.getFilter(i);

                // See if the filter's outputs already exist
                if (filter.getNumOutputs() == 0) {
                    done = false;
                } else {
                    done = true;

                    for (int outIdx = 0; outIdx < filter.getNumOutputs(); outIdx++) {
                        out = filter.getOutputFormat(outIdx);
```

```
                    if (!pipe.hasItem(out.getKey(), out.type)) {
                        done = false;

                        if (pipe.get(out.getKey()) != null) { // NOPMD SRB
                            valid = false;
                        }
                    }
                }
            }

            if (done) {
                filter.setState(FilterState.COMPLETED);
            } else {
                todo.add(filter);
            }
        }

        if (valid) {

            // Run the filters in the to-do list, in order to the extent possible
            try {

                while (!todo.isEmpty()) {

                    LOG.log(Level.INFO, "{0} filters remain to execute", todo.size());

                    // Find the first filter for which we have all needed inputs
                    found = false;

                    for (AbstractFilter test : todo) {

                        ready = true;

                        for (int inIdx = 0; inIdx < test.getNumInputs(); inIdx++) {
                            inp = test.getInputFormat(inIdx);

                            if (!pipe.hasItem(inp.getKey(), inp.type)) {
                                ready = false;

                                break;
                            }
                        }

                        if (ready) {
                            LOG.log(Level.INFO, "Running {0}", test.getName());
                            found = true;
                            publish(test, Integer.valueOf(count - todo.size()),
                                Integer.valueOf(count));
                            firePropertyChange("filter", null, test.getName());
                            test.setState(FilterState.RUNNING);
                            this.panel.repaint();
                            test.filter(this, pipe);
                            LOG.log(Level.INFO, "{0} completed", test.getName());
                            test.setState(FilterState.COMPLETED);
                            todo.remove(test);
                            this.panel.repaint();

                            break;
                        }
                    }

                    if (!found) {
                        firePropertyChange("error", null, "No filters are ready to execute");

                        break;
                    }

                    if (isCancelled()) {
                        LOG.log(Level.INFO, "Filter tree execution was canceled");

                        break;
                    }
                }
            } catch (Exception ex) {
                LOG.log(Level.SEVERE, "Exception while executing filter", ex);
                firePropertyChange("error", null, "Exception while executing filter");
            }
        }
    }

    LOG.info("doInBackground is terminating");

    return null;
}

/**
 * Called after the <code>doInBackground</code> method completes.
```

```
     *
     * <p>Called on the AWT event thread after the <code>doInBackground</code> method completes.
     */
    @Override protected void done() {

        synchronized (this.listeners) {

            for (FilterTreeExecutorListener list : this.listeners) {
                list.done();
            }
        }
    }

    /**
     * Provides notification of a progress update.
     *
     * <p>Called on the AWT event thread each time <code>publish</code> is called from within
     * <code>doInBackground</code>.
     *
     * @param  updateList  A list that should contain some multiple of three objects, where each
     *                     group of three is (in the following order) an <code>
     *                     AbstractFilter</code> (the filter being executed), an <code>
     *                     Integer</code> (the 0-based index of the current filter), and an <code>
     *                     Integer</code> (the number of filters in the tree).
     */
    @Override protected void process(final List<Object> updateList) {

        int size;
        Object obj0;
        Object obj1;
        Object obj2;

        size = updateList.size();

        for (int i = 0; i < size; i += 3) {
            obj0 = updateList.get(i);
            obj1 = updateList.get(i + 1);
            obj2 = updateList.get(i + 2);

            if ((obj0 instanceof AbstractFilter) && (obj1 instanceof Integer)
                    && (obj2 instanceof Integer)) {

                synchronized (this.listeners) {

                    for (FilterTreeExecutorListener list : this.listeners) {
                        list.process((AbstractFilter) obj0, ((Integer) obj1).intValue(),
                            ((Integer) obj2).intValue());
                    }
                }
            }
        }

    }

    /**
     * Provides an update on the progress of the filter to the user while it is executing. This
     * update is in the form of a state string, like "Processing image", and a progress value (from
     * 0 to 100).
     *
     * @param  percent  an integer that indicates percent complete (clipped to the range 0-100)
     */
    public void indicateProgress(final int percent) {

        int actual;

        if (percent < 0) {
            actual = 0;
        } else if (percent > 100) {
            actual = 100;
        } else {
            actual = percent;
        }

        setProgress(actual);

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // No action
        }
    }
}

package com.srbenoit.filter;

import java.beans.PropertyChangeListener;
```

```
/**
 * A listener for objects that want to be notified of the progress of a filter tree's execution.
 */
public interface FilterTreeExecutorListener extends PropertyChangeListener {

    /**
     * Called on the AWT event dispatcher thread after the filter tree has been completely
     * executed.
     */
    void done();

    /**
     * Called on the AWT event dispatcher thread to provide notification of a progress update.
     *
     * @param  filter  the filter being executed
     * @param  index   the 0-based index of the filter being executed
     * @param  total   the total number of filters in the tree
     */
    void process(AbstractFilter filter, int index, int total);
}

package com.srbenoit.filter;

import java.io.UnsupportedEncodingException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.net.URLEncoder;
import java.text.ParseException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.xml.ElementBase;
import com.srbenoit.xml.EmptyElement;
import com.srbenoit.xml.Node;
import com.srbenoit.xml.NonemptyElement;
import com.srbenoit.xml.XmlParser;

/**
 * A class that can generate a serialized representation of a tree of filters (XML) and can
 * reconstruct the filter tree from a list of filters and a serialized representation.
 */
public class FilterTreeIO extends LoggedObject {

    /** the character encoding to use for URL encode/decode */
    private static final String UTF8 = "UTF-8";

    /** the line end string */
    private static final String CRLF;

    /** a zero-length class array to use when finding constructors */
    private static final Class<?>[] CLASS_0;

    /** a zero-length object array to use when instantiating objects */
    private static final Object[] OBJECT_0;

    static {
        String crlf;

        crlf = System.getProperty("line.separator");
        CRLF = (crlf == null) ? "\n" : crlf;

        CLASS_0 = new Class<?>[0];
        OBJECT_0 = new Object[0];
    }

    /**
     * Constructs a new <code>FilterTreeIO</code>.
     */
    public FilterTreeIO() {

        // No action
    }

    /**
     * Generates the serialized representation of a filter tree.
     *
     * @param   tree  the filter tree
     * @return  the serialized representation
     */
    public String serialize(final FilterTree tree) {

        int count;
        StringBuilder str;

        str = new StringBuilder(256);
        count = tree.getNumFilters();
```

```
        str.append("<filter_tree>");
        str.append(CRLF);

        for (int i = 0; i < count; i++) {
            appendFilter(tree.getFilter(i), str);
        }

        str.append("</filter_tree>");
        str.append(CRLF);

        return str.toString();
    }

    /**
     * Recursively appends a filter and all of its descendants to a <code>StringBuilder</code>. The
     * output format is a simple XML string describing the tree structure of filters.
     *
     * <p>Any settings in the filter are stored as attributes in the XML string.
     *
     * @param  filter  the filter to append
     * @param  str     the <code>StringBuilder</code> to which to append
     */
    private void appendFilter(final AbstractFilter filter, final StringBuilder str) {

        String[] keys;
        String tag;
        String key;
        String value;
        int index;

        keys = filter.getPropertyKeys();

        try {
            tag = URLEncoder.encode(filter.getTag(), UTF8);

            str.append(" <");
            str.append(tag);

            for (index = 0; index < filter.getNumInputs(); index++) {
                str.append(" in-");
                str.append(Integer.toString(index));
                str.append("='");
                str.append(filter.getInputFormat(index).getKey());
                str.append('\'');
            }

            for (index = 0; index < filter.getNumOutputs(); index++) {
                str.append(" out-");
                str.append(Integer.toString(index));
                str.append("='");
                str.append(filter.getOutputFormat(index).getKey());
                str.append('\'');
            }

            for (index = 0; index < keys.length; index++) {

                try {
                    key = URLEncoder.encode(keys[index], UTF8);
                    value = URLEncoder.encode(filter.getProperty(keys[index]), UTF8);

                    // Only after successfully encoding both do we append
                    str.append(' ');
                    str.append(key);
                    str.append('=');
                    str.append('\'');
                    str.append(value);
                    str.append('\'');
                } catch (UnsupportedEncodingException e) {

                    // Any encoding error, we append nothing and warn (keep the
                    // output valid XML in this case)
                    LOG.warning("URLEncoder indicated 'UTF-8' is not supported");
                }
            }

            str.append("/>");
            str.append(CRLF);
        } catch (UnsupportedEncodingException e) {

            // If we can't encode the tag, do not append anything, but warn
            LOG.warning("URLEncoder indicated 'UTF-8' is not supported");
        }
    }

    /**
     * Given an XML representation of a filter tree, and a list of filter classes from which to
```

```
     * construct the tree, parses an XML stream and rebuilds the filter tree if possible.
     *
     * @param    xml         the XML representation
     * @param    classes     the set of filter classes from which to draw when rebuilding the tree
     * @return   the reconstructed tree
     * @throws   NoSuchMethodException       if any of the filter classes does not have a no-argument
     *                                       constructor
     * @throws   SecurityException           if this class does not have permission to access the
     *                                       no-argument constructor on any of the filters
     * @throws   IllegalAccessException      if the no-argument constructor on any of the filters is
     *                                       not accessible
     * @throws   InstantiationException      if any of the given filter classes are abstract classes
     * @throws   InvocationTargetException   if the no-argument constructor on any filter throws an
     *                                       exception
     * @throws   ParseException              if the XML could not be parsed
     */
    public FilterTree deserialize(final String xml,
        final List<Class<? extends AbstractFilter>> classes) throws SecurityException,
        NoSuchMethodException, IllegalAccessException, InstantiationException,
        InvocationTargetException, ParseException {

        Map<String, Constructor<? extends AbstractFilter>> constructors;
        Constructor<? extends AbstractFilter> constr;
        AbstractFilter instance;
        List<Node> nodes;
        NonemptyElement toplevel;
        FilterTree tree;

        // Identify the no-argument constructors on all the filters and build
        // up a map from filter tag to its constructor
        constructors = new HashMap<String, Constructor<? extends AbstractFilter>>(10);

        for (Class<? extends AbstractFilter> clazz : classes) {
            constr = clazz.getConstructor(CLASS_0);
            instance = constr.newInstance(OBJECT_0);
            constructors.put(instance.getTag(), constr);
        }

        // Parse the XML
        nodes = new XmlParser().parse(xml, true);

        // Identify the one and only top-level node
        toplevel = null;
        tree = new FilterTree();

        if (nodes.isEmpty()) {
            throw new ParseException("Input XML contained no elements", 0);
        }

        for (Node node : nodes) {

            if (node instanceof NonemptyElement) {

                if (toplevel != null) {
                    throw new ParseException("Must have only one top-level node", 0);
                }

                toplevel = (NonemptyElement) node;

                if ("filter_tree".equals(toplevel.tagName)) {

                    for (Node child : toplevel.children) {

                        if (child instanceof ElementBase) {
                            buildFilter((ElementBase) child, constructors, tree);
                        }
                    }
                } else {
                    throw new ParseException("Must have one top-level 'filter-tree' node", 0);
                }
            } else {
                throw new ParseException("Must have one top-level non-empty filter tree node", 0);
            }
        }

        // Build the filter tree
        return tree;
    }

    /**
     * Recursively builds a filter from an element. If the element is nonempty, the filter will
     * have child filters for each element child of the supplied element.
     *
     * @param    element       the element with the filter definition
     * @param    constructors  a map from filter name to no-argument constructor
     * @param    tree          the filter tree to which to add the constructed filter
     * @return   the constructed filter
```

```
     * @throws  IllegalAccessException      if the no-argument constructor on any of the filters is
     *                                      not accessible
     * @throws  InstantiationException      if any of the given filter classes are abstract classes
     * @throws  InvocationTargetException   if the no-argument constructor on any filter throws an
     *                                      exception
     * @throws  ParseException              if the element's tag name does not match any configured
     *                                      filter names
     */
    private void buildFilter(final ElementBase element,
        final Map<String, Constructor<? extends AbstractFilter>> constructors,
        final FilterTree tree) throws IllegalAccessException, InstantiationException,
        InvocationTargetException, ParseException {

        EmptyElement empty;
        Constructor<? extends AbstractFilter> constr;
        AbstractFilter filter;
        int index;
        String attrib;
        String value;
        String[] supported;

        constr = constructors.get(element.tagName);

        if (element instanceof EmptyElement) {
            empty = (EmptyElement) element;

            if (constr == null) {
                throw new ParseException("Unrecognized filter name: '" + element.tagName + "'",
                    empty.tagSpan.nameStart);
            }

            filter = constr.newInstance(OBJECT_0);

            for (index = 0; index < filter.getNumInputs(); index++) {
                attrib = "in-" + Integer.toString(index);
                value = empty.get(attrib);
                filter.getInputFormat(index).setKey(value);
            }

            for (index = 0; index < filter.getNumOutputs(); index++) {
                attrib = "out-" + Integer.toString(index);
                value = empty.get(attrib);
                filter.getOutputFormat(index).setKey(value);
            }

            supported = filter.getSupportedPropertyKeys();

            for (index = 0; index < supported.length; index++) {
                value = empty.get(supported[index]);

                if (value != null) {
                    filter.setProperty(supported[index], value);
                }
            }

            tree.addFilter(filter);
        }
    }
}

package com.srbenoit.filter;

import java.awt.AlphaComposite;
import java.awt.Color;
import java.awt.Composite;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.RenderingHints;
import java.awt.event.InputEvent;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.log.LoggedPanel;
import com.srbenoit.ui.UIUtilities;

/**
```

```
 * A panel on which a user can drag around filters so they connect into a graph.
 */
public class FilterTreePanel extends LoggedPanel implements KeyListener, MouseListener,
    MouseMotionListener {

    /** a version number for serialization */
    private static final long serialVersionUID = -8366059288608371787L;

    /** spacing between the filters in the list */
    private final static Object[] OBJECT_0;

    /** alpha composite object for rendering drag */
    private static final AlphaComposite ALPHA;

    /** distance between filters in the list and edge of panel */
    private final static int LIST_BORDER = 7;

    /** spacing between the filters in the list */
    private final static int LIST_SPACING = 7;

    /** the list of filters to show */
    private final List<AbstractFilter> listFilters;

    /** object on which to synchronize access to member variables */
    private final transient Object synch;

    /** the filter tree */
    private FilterTree tree;

    /** the filter on which a drag was started */
    private transient AbstractFilter dragFilter;

    /** flag to indicate we're dragging a filter around */
    private transient boolean isDragging;

    /** the point where dragging from the list was started */
    private transient Point dragStart;

    /** the width of the list */
    private final transient int listWidth;

    /** the width of the list */
    private transient boolean dirty;

    static {
        OBJECT_0 = new Object[0];
        ALPHA = AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.7f);
    }

    /**
     * Constructs a new <code>FilterGraphPanel</code>. Call this method from within the AWT event
     * thread.
     *
     * @param  theFilters  the list of filters to show
     * @param  filterTree  the filter tree this panel will render
     */
    @SuppressWarnings("LeakingThisInConstructor")
    public FilterTreePanel(final List<Class<? extends AbstractFilter>> theFilters,
        final FilterTree filterTree) {

        super(null);

        Font font;
        Graphics2D grx;
        Constructor<? extends AbstractFilter> cons;
        AbstractFilter filter;
        FilterRenderer renderer;
        Rectangle bounds;
        int maxWidth;
        int yPos;
        int xCenter;

        if (theFilters == null) {
            throw new IllegalArgumentException("Filter class list may not be null");
        }

        if (filterTree == null) {
            throw new IllegalArgumentException("Filter tree may not be null");
        }

        this.synch = new Object();
        this.listFilters = new ArrayList<AbstractFilter>(theFilters.size());
        this.tree = filterTree;

        setFocusable(true);

        // Determine the width of each filter's box
```

```
        font = new Font("Dialog", Font.PLAIN, 12);
        setFont(font);
        grx = UIUtilities.getGraphics();
        grx.setFont(font);
        maxWidth = 0;

        for (Class<? extends AbstractFilter> clazz : theFilters) {

            cons = AbstractFilter.getNoArgConstructor(clazz);

            if (cons == null) {
                throw new IllegalArgumentException("Unable to get no-argument constructor for "
                    + clazz.getName());
            }

            try {
                filter = cons.newInstance(OBJECT_0);
            } catch (Exception e) {
                throw new IllegalArgumentException("Unable to instantiate " + clazz.getName(), e);
            }

            renderer = filter.getRenderer();
            bounds = renderer.getBounds();

            if (bounds.width > maxWidth) {
                maxWidth = bounds.width;
            }

            this.listFilters.add(filter);
        }

        // Now, given the max width, lay out the boxes for each filter
        xCenter = LIST_BORDER + (maxWidth / 2);
        yPos = LIST_BORDER;

        for (AbstractFilter filt : this.listFilters) {
            renderer = filt.getRenderer();
            bounds = renderer.getBounds();
            renderer.setLocation(xCenter - (bounds.width / 2), yPos);
            yPos += bounds.height + LIST_SPACING;
        }

        yPos += LIST_BORDER;

        this.listWidth = LIST_BORDER + maxWidth + LIST_BORDER;
        setPreferredSize(new Dimension(1000, 500));

        addMouseListener(this);
        addMouseMotionListener(this);
        addKeyListener(this);
    }

    /**
     * Tests whether the panel contains unsaved changes to the active filter.
     *
     * @return  <code>true</code> if there are unsaved changes; <code>false</code> if not
     */
    public boolean isDirty() {

        return this.dirty;
    }

    /**
     * Sets the flag indicating whether the panel contains unsaved changes to the active filter.
     *
     * @param  isDirty  <code>true</code> if there are unsaved changes; <code>false</code> if not
     */
    public void setDirty(final boolean isDirty) {

        this.dirty = isDirty;
    }

    /**
     * Sets the filter tree that this panel will render.
     *
     * @param  filterTree  the filter tree this panel will render
     */
    public void setTree(final FilterTree filterTree) {

        if (filterTree == null) {
            throw new IllegalArgumentException("Filter tree may not be null");
        }

        synchronized (this.synch) {
            this.tree = filterTree;
        }
```

```
        repaint();
    }

    /**
     * Deletes any filters in the current filter tree and clears the dirty flag.
     */
    public void clear() {

        while (this.tree.getNumFilters() > 0) {
            this.tree.removeFilter(0);
        }

        this.dirty = false;
        repaint();
    }

    /**
     * Gets the tree of filters this panel is displaying.
     *
     * @return  the root of the filter tree
     */
    public FilterTree getTree() {

        return this.tree;
    }

    /**
     * Paints the panel.
     *
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    @Override public void paintComponent(final Graphics grx) {

        Composite orig;
        Rectangle bounds;
        Point mouse;
        boolean added;

        super.paintComponent(grx);

        if (grx instanceof Graphics2D) {
            ((Graphics2D) grx).setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_GASP);
        }

        paintList(grx);

        // If a drag is in progress, draw the drag result
        if (this.isDragging && (this.dragFilter != null)) {
            mouse = getMousePosition();

            removeProvisionals();

            if ((mouse != null) && (mouse.x > this.listWidth)) {

                // Test for a viable insertion of the dragged node in the tree
                added = checkFilterInsertion(mouse.y);

                // If the node could not be added, render it as a translucent
                // overlay as the user drags it around
                if (!added) {

                    if (grx instanceof Graphics2D) { // NOPMD SRB
                        orig = ((Graphics2D) grx).getComposite();
                        ((Graphics2D) grx).setComposite(ALPHA);
                    } else {
                        orig = null;
                    }

                    bounds = this.dragFilter.getRenderer().getBounds();
                    this.dragFilter.getRenderer().setLocation(mouse.x - (bounds.width / 2),
                        mouse.y - (bounds.height / 2));
                    this.dragFilter.getRenderer().draw(grx);

                    if (grx instanceof Graphics2D) { // NOPMD SRB
                        ((Graphics2D) grx).setComposite(orig);
                    }
                }
            }
        }

        paintTree(grx);
    }

    /**
     * Paints the list portion of the panel.
     *
```

```
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    private void paintList(final Graphics grx) {

        int size;
        Composite orig;

        size = this.listFilters.size();

        grx.setColor(Color.lightGray);
        grx.fillRect(0, 0, this.listWidth, getHeight());
        grx.setColor(Color.darkGray);
        grx.drawRect(2, 2, this.listWidth - 4, getHeight() - 4);
        grx.setColor(Color.white);
        grx.drawRect(1, 1, this.listWidth - 4, getHeight() - 4);

        for (int i = 0; i < size; i++) {
            this.listFilters.get(i).getRenderer().draw(grx);
        }

        // If we're dragging a list filter around, render it transsluscent
        if (this.isDragging) {

            if (grx instanceof Graphics2D) {
                orig = ((Graphics2D) grx).getComposite();
                ((Graphics2D) grx).setComposite(ALPHA);
            } else {
                orig = null;
            }

            this.dragFilter.getRenderer().draw(grx);

            if (grx instanceof Graphics2D) {
                ((Graphics2D) grx).setComposite(orig);
            }
        }
    }

    /**
     * Removes any provisional filters from the filter tree.
     */
    private void removeProvisionals() {

        int index;
        AbstractFilter filter;

        index = 0;

        while (index < this.tree.getNumFilters()) {
            filter = this.tree.getFilter(index);

            if (filter.isProvisional()) {
                this.tree.removeFilter(index);
                this.dirty = true;
            } else {
                index++;
            }
        }
    }

    /**
     * Recursively clears the provisional status on any provisional filters in the filter tree.
     */
    private void commitProvisionals() {

        int index;
        AbstractFilter filter;

        index = 0;

        while (index < this.tree.getNumFilters()) {
            filter = this.tree.getFilter(index);

            if (filter.isProvisional()) {
                filter.setProvisional(false);
                this.dirty = true;
            } else {
                index++;
            }
        }
    }

    /**
     * Tests whether the drag location puts the filter being dragged at a point where it could join
     * the tree. In order for a filter to join a tree, there must be a node above the filter in the
     * tree whose output matches each of this node's inputs. If this node has no inputs, the filter
     * can be inserted.
```

```
     *
     * <p>If we find a match, the filer being dragged is provisionally added to the tree (and any
     * other provisional filters are removed from the tree).
     *
     * @param   yPos  the Y position of the mouse
     * @return  <code>true</code> if the filter was provisionally added to the tree; <code>
     *          false</code> if not
     */
    private boolean checkFilterInsertion(final int yPos) {

        AbstractFilter test;
        AbstractFilter filter;
        Rectangle bounds;
        boolean isOK;

        test = this.dragFilter;

        if (test == null) {
            isOK = false;
        } else if (test.getNumInputs() == 0) {

            if (this.tree.getNumFilters() == 0) {
                test.setProvisional(true);
                this.tree.addFilter(test);
                this.dirty = true;
                isOK = true;
            } else {
                isOK = false;

                for (int j = this.tree.getNumFilters() - 1; j >= 0; j--) {
                    filter = this.tree.getFilter(j);
                    bounds = filter.getRenderer().getBounds();

                    if ((bounds.getY() + bounds.getHeight()) < yPos) {

                        test.setProvisional(true);
                        this.tree.addFilter(j + 1, test);
                        this.dirty = true;
                        isOK = true;

                        break;
                    }
                }
            }
        } else {
            isOK = testForCompatible(test, yPos);

            if (isOK) {

                for (int j = this.tree.getNumFilters() - 1; j >= 0; j--) {
                    filter = this.tree.getFilter(j);
                    bounds = filter.getRenderer().getBounds();

                    if ((bounds.getY() + bounds.getHeight()) < yPos) {

                        test.setProvisional(true);
                        this.tree.addFilter(j + 1, test);
                        this.dirty = true;

                        break;
                    }
                }
            }
        }

        return isOK;
    }

    /**
     * Walks the filter tree for all nodes whose Y value is less than a given Y value, checking
     * whether all required inputs for a filter are present.
     *
     * @param  test  the filter to be tested
     * @param  yPos  the Y position of the mouse (nodes with Y values higher than this will not be
     *               considered)
     */
    private boolean testForCompatible(final AbstractFilter test, final int yPos) {

        AbstractFilter filter;
        boolean found;
        boolean valid;
        Rectangle bounds;
        FilterInput input;
        FilterOutput output;

        valid = true;
```

```
        for (int inIdx = 0; inIdx < test.getNumInputs(); inIdx++) {
            input = test.getInputFormat(inIdx);

            found = false;

outer:
            for (int j = this.tree.getNumFilters() - 1; j >= 0; j--) {
                filter = this.tree.getFilter(j);
                bounds = filter.getRenderer().getBounds();

                if ((bounds.getY() + bounds.getHeight()) < yPos) {

                    for (int outIdx = 0; outIdx < filter.getNumOutputs(); outIdx++) {
                        output = filter.getOutputFormat(outIdx);

                        if (output.type.equals(input.type)) {
                            input.setKey(output.getKey());
                            found = true;

                            break outer;
                        }
                    }
                }
            }

            if (!found) {
                valid = false;

                break;
            }
        }

        return valid;
    }

    /**
     * Draws the filter tree in the window.
     *
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    private void paintTree(final Graphics grx) {

        FontMetrics met;
        int unit;
        int yPos;
        int xPos;
        int xPix;
        int yPix;
        int rightEdge;
        int minX;
        AbstractFilter filter;
        FilterRenderer renderer;
        Rectangle bounds;
        FilterInput inFmt;
        FilterOutput outFmt;
        AbstractFilter testing;
        int intersectX;
        int intersectY;

        met = grx.getFontMetrics();
        unit = met.getHeight() / 6;

        xPos = this.listWidth + (2 * unit);
        yPos = 2 * unit;
        rightEdge = 0;

        for (int i = 0; i < this.tree.getNumFilters(); i++) {
            filter = this.tree.getFilter(i);
            renderer = filter.getRenderer();
            bounds = renderer.getBounds();

            // Determine the minimum X coordinate where the new filter can be drawn
            minX = rightEdge - bounds.width;

            if (filter.getNumInputs() > 0) {
                minX += bounds.width - renderer.getInputX(0) + 4;
            }

            if (xPos < minX) {
                xPos = minX;
            }

            rightEdge = xPos + bounds.width;

            renderer.setLocation(xPos, yPos);
            renderer.draw(grx);
            bounds = renderer.getBounds();
```

```
            // Continue the output lines across the diagram
            for (int out = 0; out < filter.getNumOutputs(); out++) {
                xPix = bounds.x + bounds.width;
                yPix = bounds.y + renderer.getOutputY(out);
                grx.setColor(Color.BLACK);
                grx.drawLine(xPix, yPix, getWidth(), yPix);
                grx.setColor(Color.GRAY);
                grx.drawString(filter.getOutputFormat(out).description, xPix + 5,
                    yPix - 2 - met.getDescent());
            }

            // Draw lines linking inputs to prior outputs, including drag handles
            grx.setColor(Color.BLACK);

            for (int in = 0; in < filter.getNumInputs(); in++) {
                inFmt = filter.getInputFormat(in);
                intersectX = bounds.x + renderer.getInputX(in);

outer:
                for (int ii = i - 1; ii >= 0; ii--) {
                    testing = this.tree.getFilter(ii);

                    for (int out = 0; out < testing.getNumOutputs(); out++) {
                        outFmt = testing.getOutputFormat(out);

                        if ((outFmt.type.equals(inFmt.type))
                                && (outFmt.getKey().equals(inFmt.getKey()))) {
                            intersectY = testing.getRenderer().getBounds().y
                                + testing.getRenderer().getOutputY(out);
                            grx.drawLine(intersectX, intersectY, intersectX, yPos);
                            grx.fillOval(intersectX - 3, intersectY - 3, 6, 6);

                            break outer;
                        }
                    }
                }
            }

            yPos += bounds.height + (2 * unit);
        }
    }

    /**
     * Recursively places all filters in the tree in a not selected state.
     *
     * @param  tree  the base of the tree to recurse through
     */
    private void clearSelections() {

        for (int i = 0; i < this.tree.getNumFilters(); i++) {
            this.tree.getFilter(i).setSelected(false);
        }
    }

    /**
     * Handles mouse drag events.
     *
     * @param  evt  the mouse event
     */
    public void mouseDragged(final MouseEvent evt) {

        Point where;
        Rectangle bounds;

        if (isEnabled()) {
            where = evt.getPoint();

            if (isDragging) {
                bounds = this.dragFilter.getRenderer().getBounds();

                if (this.dragFilter.isProvisional()) {

                    if (where.x < this.listWidth) {
                        this.dragFilter.getRenderer().setLocation(where.x - (bounds.width / 2),
                            where.y - (bounds.height / 2));
                    }
                } else {
                    this.dragFilter.getRenderer().setLocation(where.x - (bounds.width / 2),
                        where.y - (bounds.height / 2));
                }
            } else {

                if ((this.dragFilter != null) && (where.distance(this.dragStart) > 30)) {
                    bounds = this.dragFilter.getRenderer().getBounds();
                    this.isDragging = true;
```

```
                if (where.x < this.listWidth) {

                    // When dragging from the list, we duplicate so the list is unchanged
                    this.dragFilter = this.dragFilter.duplicate();
                    this.dragFilter.getRenderer().setLocation(where.x - (bounds.width / 2),
                        where.y - (bounds.height / 2));
                } else {

                    // When dragging from the tree, delete the filter from the tree
                    for (int i = 0; i < this.tree.getNumFilters(); i++) {

                        if (this.tree.getFilter(i) == this.dragFilter) {
                            this.tree.removeFilter(i);
                            this.dirty = true;

                            break;
                        }
                    }
                }
            }
        }

        repaint();
    }
}

/**
 * Handles mouse move events.
 *
 * @param  evt  the mouse event
 */
public void mouseMoved(final MouseEvent evt) {

    // No action
}

/**
 * Handles mouse click events. We recognize double-clicks and bring up the properties pages for
 * any selected filters.
 *
 * @param  evt  the mouse event
 */
public void mouseClicked(final MouseEvent evt) {

    AbstractFilter filter;

    if (isEnabled()) {
        requestFocus();

        if (evt.getClickCount() == 2) {

            // See if the double-click occurred on a filter
            for (int i = 0; i < this.tree.getNumFilters(); i++) {

                filter = this.tree.getFilter(i);

                if (filter.getRenderer().getBounds().contains(evt.getPoint())) {

                    // TODO: Properties pages
                    LOG.log(Level.INFO, "Properties pages for {0}", filter.getName());

                    break;
                }
            }
        }
    }
}

/**
 * Handles mouse press events.
 *
 * @param  evt  the mouse event
 */
public void mousePressed(final MouseEvent evt) {

    Point where;
    AbstractFilter filter;

    if (isEnabled()) {

        requestFocus();

        where = evt.getPoint();

        if (where.x > this.listWidth) {

            // Shift key can do multiple selections
```

```
            if ((evt.getModifiers() & InputEvent.SHIFT_MASK) == 0) {
                clearSelections();
            }

            // See if the mouse was pressed inside a filter
            for (int i = 0; i < this.tree.getNumFilters(); i++) {

                filter = this.tree.getFilter(i);

                if (filter.getRenderer().getBounds().contains(where)) {

                    filter.setSelected(true);

                    this.dragFilter = filter;
                    this.dragStart = where;
                    this.isDragging = false;

                    break;
                }
            }

            repaint();
        } else {

            // See if the mouse press was in a filter
            for (AbstractFilter listFilter : this.listFilters) {

                if (listFilter.getRenderer().getBounds().contains(where)) {
                    this.dragFilter = listFilter;
                    this.dragStart = where;
                    this.isDragging = false;

                    break;
                }
            }
        }
    }
}

/**
 * Handles mouse release events.
 *
 * @param  evt  the mouse event
 */
public void mouseReleased(final MouseEvent evt) {

    if (isEnabled()) {
        this.dragStart = null;

        if (this.dragFilter != null) {
            commitProvisionals();
            this.dragFilter = null;
        }

        this.isDragging = false;
        repaint();
    }
}

/**
 * Handles mouse enter events.
 *
 * @param  evt  the mouse event
 */
public void mouseEntered(final MouseEvent evt) {

    // No action
}

/**
 * Handles mouse exit events.
 *
 * @param  evt  the mouse event
 */
public void mouseExited(final MouseEvent evt) {

    // No action
}

/**
 * Handles key typed events
 *
 * @param  evt  the key event
 */
public void keyTyped(final KeyEvent evt) {

    // No action
```

```
    }

    /**
     * Handles key pressed events
     *
     * @param  evt  the key event
     */
    public void keyPressed(final KeyEvent evt) {

        int index = 0;

        if ((isEnabled() && (evt.getKeyCode() == KeyEvent.VK_DELETE))
                || (evt.getKeyCode() == KeyEvent.VK_BACK_SPACE)) {

            while (index < this.tree.getNumFilters()) {

                if (this.tree.getFilter(index).isSelected()) {
                    this.tree.removeFilter(index);
                    this.dirty = true;
                } else {
                    index++;
                }
            }

            repaint();
        }
    }

    /**
     * Handles key released events
     *
     * @param  evt  the key event
     */
    public void keyReleased(final KeyEvent evt) {

        // No action
    }
}
```

```
package com.srbenoit.filter;

import java.io.File;
import javax.swing.filechooser.FileFilter;

/**
 * A file filter to restrict access to ".ftree" files.
 */
public class GraphFileFilter extends FileFilter {

    /**
     * Tests whether or not the specified abstract pathname should be included in a pathname list.
     *
     * @param   pathname  the abstract pathname to be tested
     * @return  <code>true</code> if and only if <code>pathname</code> should be included
     */
    @Override public boolean accept(final File pathname) {

        return pathname.getName().toLowerCase().endsWith(".ftree");
    }

    /**
     * The description of this filter.
     *
     * @return  the description
     */
    @Override public String getDescription() {

        return "*.ftree (Filter Trees)";
    }
}
```

```
package com.srbenoit.filter;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.text.ParseException;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.logging.Level;
import java.util.zip.CRC32;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.xml.ElementBase;
import com.srbenoit.xml.EmptyElement;
import com.srbenoit.xml.Node;
```

```
import com.srbenoit.xml.NonemptyElement;
import com.srbenoit.xml.XmlParser;

/**
 * A pipe from which filters can draw input data and to which they can add output data. Pipes
 * manage the storage of their items to a filesystem and reloading them when the pipe is created to
 * allow filters to avoid regenerating existing data.
 *
 * <p>Pipes contain a map from a <code>String</code> item key to an item that must be a subclass of
 * <code>AbstractPipeItem</code>. A pipe may contain any number of items. Pipes begin in an empty
 * state (no items). At any time, the pipe may be saved to the filesystem, and a saved pipe can be
 * reloaded.
 */
public class Pipe extends LoggedObject {

    /** version number for serialization */
    private static final long serialVersionUID = 6523308344875295475L;

    /** an object to compute file CRCs */
    private static final transient CRC32 CRC;

    /** line termination characters */
    public static final String CRLF;

    /** the directory where the pipe's data should be stored (or null) */
    private final transient File dir;

    /** the items in the pipe, each with unique key */
    private final Map<String, AbstractPipeItem> items;

    static {
        String crlf;

        crlf = System.getProperty("line.separator");
        CRLF = (crlf == null) ? "\n" : crlf;
        CRC = new CRC32();
    }

    /**
     * Constructs a new <code>Pipe</code>.
     *
     * @param  pipeDir  the directory where the pipe's data should be stored (if this is <code>
     *                  null</code>, the pipe cannot be persisted)
     */
    public Pipe(final File pipeDir) {

        super();

        if (pipeDir.exists()) {

            if (!pipeDir.isDirectory()) {
                throw new IllegalArgumentException("Invalid directory: "
                    + pipeDir.getAbsolutePath());
            }
        } else {

            if (!pipeDir.mkdirs()) {
                throw new IllegalArgumentException("Invalid directory: "
                    + pipeDir.getAbsolutePath());
            }
        }

        this.items = new TreeMap<String, AbstractPipeItem>();
        this.dir = pipeDir;
    }

    /**
     * Gets the directory where items in the pipe should be persisted to the filesystem.
     *
     * @return  the directory
     */
    public File getDir() {

        return this.dir;
    }

    /**
     * Adds an item to the pipe (usually the output of a filter).
     *
     * @param  item  the item to add
     */
    public void add(final AbstractPipeItem item) {

        this.items.put(item.getKey(), item);
    }

    /**
```

```
     * Tests whether the pipe contains a data item of a particular class and with a particular key.
     *
     * @param   key     the key to test for
     * @param   clazz   the class that the object with the given key must have
     * @return  <code>true</code> if the pipe has an object of the required class under the given
     *          key
     */
    public boolean hasItem(final String key, final Class<? extends AbstractPipeItem> clazz) {

        AbstractPipeItem item;
        boolean hasItem;

        item = get(key);

        if (item == null) {
            hasItem = false;
        } else {
            hasItem = item.getClass().equals(clazz);
        }

        return hasItem;
    }

    /**
     * Gets an item from the pipe.
     *
     * @param   key  the key of the item to get
     * @return  the item
     */
    public AbstractPipeItem get(final String key) {

        return this.items.get(key);
    }

    /**
     * Tests whether there is a complete persisted data set that can be loaded to fill the pipe
     * from the filesystem.
     *
     * @return  <code>true</code> if there is a complete persisted data set on the filesystem;
     *          <code>false</code> otherwise
     */
    public boolean isPersisted() {

        File file;
        List<Node> nodes;
        Node node;
        NonemptyElement pipe;
        boolean result;

        file = pipeInfoFile();
        nodes = parsePipeFile(file);

        // Root of tree should be 'pipe' node...
        if (nodes == null) {
            LOG.log(Level.WARNING, "Unable to parse pipe file: {0}",
                    pipeInfoFile().getAbsolutePath());
            result = false;
        } else if (nodes.isEmpty()) {
            LOG.log(Level.WARNING, "Missing ''pipe'' node in pipe file: {0}",
                    pipeInfoFile().getAbsolutePath());
            result = false;
        } else if (nodes.size() > 1) {
            LOG.log(Level.WARNING, "Multiple top-level nodes in pipe file: {0}",
                    pipeInfoFile().getAbsolutePath());
            result = false;
        } else {
            node = nodes.get(0);

            if (node instanceof NonemptyElement) {
                pipe = (NonemptyElement) node;

                if ("pipe".equals(pipe.tagName)) {

                    // validate list of persisted items
                    result = true;

                    for (int i = 0; result && (i < pipe.children.size()); i++) {
                        result = validateItem(pipe.children.get(i));
                    }

                } else {
                    LOG.log(Level.WARNING, "Missing top-level ''pipe'' node in pipe file: {0}",
                            pipeInfoFile().getAbsolutePath());
                    result = false;
                }
            } else {
                LOG.log(Level.WARNING, "Invalid top-level node in pipe file: {0}",
```

```
                    pipeInfoFile().getAbsolutePath());
                result = false;
            }
        }

        return result;
    }

    /**
     * Gets the file where the pipe information is stored.
     *
     * @return  the file
     */
    private File pipeInfoFile() {

        return new File(this.dir, "pipe_info.xml");
    }

    /**
     * Reads the pipe information file and parses the XML into a node tree.
     *
     * @return  the parsed node tree, or <code>null</code> if parsing failed
     */
    private List<Node> parsePipeFile(final File file) {

        byte[] content;
        List<Node> nodes;

        if (file.exists()) {
            content = readFile(file);

            if (content == null) {
                nodes = null;
            } else {

                try {
                    nodes = new XmlParser().parse(new String(content), true);
                } catch (ParseException e) {
                    LOG.log(Level.WARNING, "Unable to parse pipe file", e);
                    nodes = null;
                }
            }
        } else {
            LOG.log(Level.WARNING, "Pipe file {0} not found", file.getAbsolutePath());
            nodes = null;
        }

        return nodes;
    }

    /**
     * validates a single pipe items information and tests whether the item's data files have the
     * correct size and SHA-1 hash.
     *
     * @param   itemNode  the item's node in the parsed pipe XML
     * @return  <code>true</code> if the item is valid and all indicated files are present and have
     *          the correct size and hash
     */
    private boolean validateItem(final Node itemNode) {

        NonemptyElement element;
        String key;
        String clsName;
        String label;
        String type;
        Class<?> clazz;
        boolean result;

        if (itemNode instanceof NonemptyElement) {

            element = (NonemptyElement) itemNode;

            if ("item".equals(element.tagName)) {
                key = element.get("key");
                clsName = element.get("class");
                label = element.get("label");
                type = element.get("type");

                // Make sure all required parameters are present
                if ((key == null) || (clsName == null) || (label == null) || (type == null)) {
                    LOG.log(Level.WARNING,
                            "Missing required attribute in item element in pipe file {0}",
                            pipeInfoFile().getAbsolutePath());
                    result = false;
                } else {

                    // Make sure the class name is a valid class
```

```
                try {
                    clazz = Class.forName(clsName);

                    // Make sure the class extends AbstractPipeItem
                    result = false;

                    while (clazz != null) {

                        if (clazz.equals(AbstractPipeItem.class)) {
                            result = true;

                            break;
                        }

                        clazz = clazz.getSuperclass();
                    }

                    if (!result) {
                        LOG.log(Level.WARNING,
                            "Class ''{0}'' in item element not a subclass of AbstractPipeItem in pipe file {1}",
                            new Object[] { clsName, pipeInfoFile().getAbsolutePath() });
                    }
                } catch (ClassNotFoundException e) {
                    LOG.log(Level.WARNING,
                        "Invalid class ''{0}'' in item element in pipe file {1}",
                        new Object[] { clsName, pipeInfoFile().getAbsolutePath() });
                    result = false;
                }
            }

            if (result) {

                // Class is valid, so verify files are valid
                for (Node child : element.children) {

                    if (!validateItemFile(child)) {
                        result = false;

                        break;
                    }
                }
            }
        } else {
            LOG.log(Level.WARNING,
                "Invalid element ''{0}'' (expected ''item'') in pipe file {1}",
                new Object[] { element.tagName, pipeInfoFile().getAbsolutePath() });
            result = false;
        }
    } else {
        LOG.log(Level.WARNING, "Invalid node in pipe file {0}",
            pipeInfoFile().getAbsolutePath());
        result = false;
    }

    return result;
}

/**
 * validates a single file within a pipe item and tests whether it has the correct size and
 * SHA-1 hash.
 *
 * @param   fileNode  the file's node in the parsed pipe item XML
 * @return  <code>true</code> if the file is present and has the correct size and hash
 */
private boolean validateItemFile(final Node fileNode) {

    EmptyElement element;
    String name;
    String size;
    String crc;
    File test;
    boolean result;

    if (fileNode instanceof EmptyElement) {

        element = (EmptyElement) fileNode;

        if ("file".equals(element.tagName)) {

            name = element.get("name");
            size = element.get("size");
            crc = element.get("crc");

            test = new File(this.dir, name);

            if (test.exists()) {
                result = checkFile(test, size, crc);
```

```
            } else {
                LOG.log(Level.WARNING, "File ''{0}'' missing", test.getAbsolutePath());
                result = false;
            }
        } else {
            LOG.log(Level.WARNING,
                "Invalid element ''{0}'' (expected ''file'') in pipe file {1}",
                new Object[] { element.tagName, pipeInfoFile().getAbsolutePath() });
            result = false;
        }
    } else {
        LOG.log(Level.WARNING, "Invalid file node in pipe file {0}",
            pipeInfoFile().getAbsolutePath());
        result = false;
    }

    return result;
}

/**
 * Checks the size and hash of a file against expected values.
 *
 * @param   file  the file to test
 * @param   size  the expected file size (string representation)
 * @param   crc   the expected file CRC
 * @return  <code>true</code> if the file has the correct size and hash
 */
private boolean checkFile(final File file, final String size, final String crc) {

    String test;
    boolean result;

    test = Long.toString(file.length());

    if (test.equals(size)) {
        test = crcFile(file);

        result = test.equals(crc);

        if (!result) {
            LOG.log(Level.WARNING, "File ''{0}'' CRC mismatch (got {1}, expected {2})",
                new Object[] { file.getAbsolutePath(), test, crc });
        }
    } else {
        LOG.log(Level.WARNING, "File ''{0}'' size mismatch (got {1}, expected {2})",
            new Object[] { file.getAbsolutePath(), test, size });
        result = false;
    }

    return result;
}

/**
 * Writes out a filled pipe to the filesystem.
 *
 * @param   executor  the filter tree executor that is running filters and saving the pipe from
 *                    time to time
 * @return  <code>true</code> if saving succeeded; <code>false</code> otherwise
 */
public boolean save(final FilterTreeExecutor executor) {

    int total;
    int count;
    int finished;
    PipeItemFileInfo[] files;
    StringBuilder str;
    AbstractPipeItem item;
    boolean success;

    // Count the total number of item files that need persisting
    total = 0;
    finished = 0;

    for (String key : this.items.keySet()) {
        item = this.items.get(key);

        files = item.getFiles();

        for (int i = 0; i < files.length; i++) {

            if (!files[i].isPersisted()) {
                total++;
            }
        }
    }

    str = new StringBuilder(500);
```

```
        str.append("<pipe>");
        str.append(CRLF);

        success = true;

        for (String key : this.items.keySet()) {
            item = this.items.get(key);

            str.append("  <item key='");
            str.append(ElementBase.encode(key));
            str.append("' class='");
            str.append(ElementBase.encode(item.getClass().getName()));
            str.append("' label='");
            str.append(ElementBase.encode(item.getLabel()));
            str.append("' type='");
            str.append(ElementBase.encode(item.typeName()));
            str.append("'>");
            str.append(CRLF);

            count = 0;
            files = item.getFiles();

            for (int i = 0; i < files.length; i++) {

                if (!files[i].isPersisted()) {
                    count++;
                }
            }

            success = item.save(executor, 80 + (20 * finished / total),
                    80 + (20 * (count + finished) / total));

            if (!success) {
                break;
            }

            finished += count;

            for (PipeItemFileInfo test : item.getFiles()) {
                str.append("    <file name='");
                str.append(ElementBase.encode(test.getFile().getName()));
                str.append("' size='");
                str.append(test.getSize());
                str.append("' crc='");
                str.append(test.getCrc());
                str.append("'/>");
                str.append(CRLF);
            }

            str.append("  </item>");
            str.append(CRLF);
        }

        str.append("</pipe>");
        str.append(CRLF);

        if (success) {
            success = writeFile(pipeInfoFile(), str.toString().getBytes());
        }

        return success;
    }

    /**
     * Loads the pipe from the persisted data set. If loading does not succeed, the pipe will be
     * returned to an empty state (any data in the pipe before loading was attempted will be lost).
     *
     * @return  <code>true</code> if loading succeeded and the pipe is now filled; <code>
     *          false</code> otherwise
     */
    public boolean load() {

        File file;
        List<Node> nodes;
        Node node;
        NonemptyElement pipe;
        NonemptyElement item;
        boolean result;

        file = pipeInfoFile();
        LOG.log(Level.INFO, "Attempting to load pipe from: {0}", file.getAbsolutePath());

        nodes = parsePipeFile(file);

        // Root of tree should be 'pipe' node...
        if (nodes == null) {
```

```
            LOG.log(Level.WARNING, "Unable to parse pipe file: {0}",
                pipeInfoFile().getAbsolutePath());
            result = false;
        } else if (nodes.isEmpty()) {
            LOG.log(Level.WARNING, "Missing ''pipe'' node in pipe file: {0}",
                pipeInfoFile().getAbsolutePath());
            result = false;
        } else if (nodes.size() > 1) {
            LOG.log(Level.WARNING, "Multiple top-level nodes in pipe file: {0}",
                pipeInfoFile().getAbsolutePath());
            result = false;
        } else {
            node = nodes.get(0);

            if (node instanceof NonemptyElement) {
                pipe = (NonemptyElement) node;

                if ("pipe".equals(pipe.tagName)) {

                    // Load all persisted items
                    result = true;

                    for (Node child : pipe.children) {

                        if (child instanceof NonemptyElement) {
                            item = (NonemptyElement) child;

                            if ("item".equals(item.tagName)) {
                                result = loadItem(item);

                                if (!result) {
                                    break;
                                }
                            }
                        }
                    }
                } else {
                    LOG.log(Level.WARNING, "Missing top-level ''pipe'' node in pipe file: {0}",
                        pipeInfoFile().getAbsolutePath());
                    result = false;
                }
            } else {
                LOG.log(Level.WARNING, "Invalid top-level node in pipe file: {0}",
                    pipeInfoFile().getAbsolutePath());
                result = false;
            }
        }

        return result;
    }

    /**
     * Loads a single item in the persisted pipe.
     *
     * @param   item  the node describing the item to load
     * @return  the loaded item
     */
    @SuppressWarnings("unchecked")
    private boolean loadItem(final NonemptyElement item) {

        String key;
        String clsName;
        String label;
        Class<?> cls;
        Class<? extends AbstractPipeItem> clazz;
        AbstractPipeItem obj;
        boolean result;

        key = item.get("key");

        if (key == null) {
            LOG.warning("Missing 'key' tag in <info> element");
            result = false;
        } else {
            clsName = item.get("class");

            if (clsName == null) {
                LOG.warning("Missing 'class' tag in <info> element");
                result = false;
            } else {
                label = item.get("label");

                if (label == null) {
                    LOG.warning("Missing 'label' tag in <info> element");
                    result = false;
                } else {
```

```
                    try {
                        cls = Class.forName(clsName);

                        if (AbstractPipeItem.class.isAssignableFrom(cls)) {
                            clazz = (Class<? extends AbstractPipeItem>) cls;
                            obj = AbstractPipeItem.getInstance(clazz, key, label, this);
                            result = obj.load();

                            if (result) {
                                this.items.put(obj.getKey(), obj);
                            }
                        } else {
                            LOG.log(Level.WARNING,
                                "Class ({0}) specified in ''class'' tag in <info> element is not a subclass of Abst
                                clsName);
                            result = false;
                        }
                    } catch (ClassNotFoundException e) {
                        LOG.log(Level.WARNING,
                            "Invalid class ({0}) specified in ''class'' tag in <info> element",
                            clsName);
                        result = false;
                    }
                }
            }
        }

        return result;
    }

    /**
     * A utility method to write a file.
     *
     * @param   file  the <code>File</code> to which to write
     * @param   data  the data to write to the file
     * @return  <code>true</code> if the file was written successfully; <code>false</code> if not
     */
    public static boolean writeFile(final File file, final byte[] data) {

        FileOutputStream out;
        boolean result;

        try {

            if (!file.getParentFile().exists()) {
                file.getParentFile().mkdirs();
            }

            out = new FileOutputStream(file);

            try {
                out.write(data);
                result = true;
            } finally {
                out.close();
            }
        } catch (IOException e) {
            LOG.log(Level.WARNING, "Exception while writing file '" + file.getAbsolutePath() + "'",
                e);
            result = false;
        }

        return result;
    }

    /**
     * A utility method to read a file.
     *
     * @param   file  the <code>File</code> from which to read
     * @return  the data read from the file, or <code>null</code> on any error
     */
    public static byte[] readFile(final File file) {

        int len;
        int total;
        int count;
        FileInputStream input;
        byte[] result;

        try {
            len = (int) file.length();
            result = new byte[len];
            input = new FileInputStream(file);

            try {
                total = input.read(result);
```

```
            if (total == -1) {
                LOG.log(Level.WARNING, "Premature end of file on file ''{0}''",
                    file.getAbsolutePath());
                result = null;
            } else {

                while (total < len) {
                    count = input.read(result, total, len - total);

                    if (count == -1) {
                        LOG.log(Level.WARNING, "Premature end of file on file ''{0}''",
                            file.getAbsolutePath());
                        result = null;

                        break;
                    }
                }
            }
        } finally {
            input.close();
        }
    } catch (IOException e) {
        LOG.log(Level.WARNING, "Exception while reading file '" + file.getAbsolutePath() + "'",
            e);
        result = null;
    }

    return result;
}

/**
 * Computes the CRC-32 of a file.
 *
 * @param   file  the file whose CRC is to be computed
 * @return  the file hash, or <code>null</code> if the file did not exist or could not be read
 */
public static String crcFile(final File file) {

    FileInputStream input;
    int length;
    int total;
    int count;
    byte[] buffer;
    String crc;

    if (CRC == null) {
        LOG.warning("No CRC computation engine");
        crc = null;
    } else {

        buffer = new byte[1024];
        length = (int) file.length();

        try {
            input = new FileInputStream(file);

            total = input.read(buffer);

            if (total < 1) {
                LOG.warning("CRC: failed to read from file");
                crc = null;
            } else {

                synchronized (CRC) {
                    CRC.reset();
                    CRC.update(buffer, 0, total);

                    while (total < length) {
                        count = input.read(buffer);

                        if (count < 1) {
                            break;
                        }

                        CRC.update(buffer, 0, count);
                        total += count;
                    }
                }

                if (total == length) {
                    crc = Long.toString(CRC.getValue());
                } else {
                    LOG.warning("CRC: failed to read all of file");
                    crc = null;
                }

                input.close();
```

```
                }
            } catch (IOException e) {
                LOG.log(Level.WARNING, "Exception while computing CRC", e);
                crc = null;
            }
        }

        return crc;
    }
}

package com.srbenoit.filter;

import java.io.File;

/**
 * Information on a file used to persist part of a pipe item.
 */
public class PipeItemFileInfo {

    /** the file */
    private File file;

    /** true if persisted; false if not */
    private boolean persisted;

    /** the file size */
    private long size;

    /** the file CRC */
    private String crc;

    /**
     * Constructs a new <code>PipeItemFileInfo</code> representing an unpersisted file.
     *
     * @param  theFile  the file
     */
    public PipeItemFileInfo(final File theFile) {

        this.file = theFile;
        this.persisted = false;
        this.size = 0;
        this.crc = null;
    }

    /**
     * Gets the file.
     *
     * @return  the file
     */
    public File getFile() {

        return this.file;
    }

    /**
     * Sets the file.
     *
     * @param  newFile  the new file
     */
    public void setFile(final File newFile) {

        this.file = newFile;
    }

    /**
     * Tests whether the file has been persisted.
     *
     * @return  <code>true</code> if the file is persisted; <code>false</code> if not
     */
    public boolean isPersisted() {

        return this.persisted;
    }

    /**
     * Gets the size of the persisted file.
     *
     * @return  the file size
     */
    public long getSize() {

        return this.size;
    }

    /**
     * Gets the CRC of the persisted file, expressed as a <code>String</code> (this is to
```

```
     * facilitate its use in XML files, where comparisons can be done without parsing the numerical
     * value from the XML).
     *
     * @return  the file CRC
     */
    public String getCrc() {

        return this.crc;
    }

    /**
     * Called after the file is actually persisted, this computes the size and CRC and sets the
     * persisted flag.
     */
    public void wasPersisted() {

        if (this.file.exists()) {
            this.size = this.file.length();
            this.crc = Pipe.crcFile(this.file);
            this.persisted = true;
        }
    }

    /**
     * Indicates that the file is not persisted.
     */
    public void notPersisted() {

        this.size = 0;
        this.crc = null;
        this.persisted = false;
    }
}
```

E.5.2 Filter Tree Infrastructure (com.srbenoit.filter.filters)

This package contains example filters to demonstrate how filters are constructed.

```
package com.srbenoit.filter.filters;

import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ByteArrayPipeItem;
import com.srbenoit.filter.items.StringPipeItem;
import org.bouncycastle.util.encoders.Base64;

/**
 * A filter that converts a Base-64 representation of some data back into the raw byte array.
 */
public class FromBase64 extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = -374358648313375169L;

    /**
     * Constructs a new <code>FromBase64</code>.
     */
    public FromBase64() {

        super("FromBase64", FromBase64.class.getName());

        this.inputs.add(new FilterInput(StringPipeItem.class, "Base-64 encoded string"));
        this.outputs.add(new FilterOutput(ByteArrayPipeItem.class, "Recovered binary data",
                "binary"));
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new FromBase64();
    }
```

```
    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        StringPipeItem in0;
        ByteArrayPipeItem out0;

        validateInputs(pipe);

        in0 = (StringPipeItem) pipe.get("Base64");

        out0 = new ByteArrayPipeItem("decoded", "Binary", pipe);
        out0.setData(Base64.decode(in0.getData()));

        pipe.add(out0);
    }
}

package com.srbenoit.filter.filters;

import java.util.Random;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.StringPipeItem;

/**
 * A filter that generates quotations.
 */
public class Quotations extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = -5381326527364928071L;

    /** The quotes this filter can generate */
    public static final String[] QUOTES = {
            "A fanatic is one who can't change his mind and won't change the subject.",
            "A love for tradition has never weakened a nation, indeed it has strengthened nations in
their hour of peril.",
            "All great things are simple, and many can be expressed in single words: freedom, justice,
honor, duty, mercy, hope.",
            "Although prepared for martyrdom, I preferred that it be postponed.",
            "An appeaser is one who feeds a crocodile, hoping it will eat him last.",
            "Broadly speaking, the short words are the best, and the old words best of all.",
            "Every day you may make progress. Every step may be fruitful. Yet there will stretch
out before you an ever-lengthening, ever-ascending, ever-improving path. You know you will never get
to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the
climb.",
            "From now on, ending a sentence with a preposition is something up with which I will not
put.",
            "He has all the virtues I dislike and none of the vices I admire.",
            "History will be kind to me for I intend to write it.",
            "However beautiful the strategy, you should occasionally look at the results.",
            "I cannot pretend to feel impartial about colours. I rejoice with the brilliant ones and
am genuinely sorry for the poor browns.",
            "I have always felt that a politician is to be judged by the animosities he excites
among his opponents.",
            "I like pigs. Dogs look up to us. Cats look down on us. Pigs treat us as equals.",
            "It has been said that democracy is the worst form of government except all the others
that have been tried.",
            "It is a mistake to try to look too far ahead. The chain of destiny can only be grasped
one link at a time.",
            "It's not enough that we do our best; sometimes we have to do what's required.",
            "Men occasionally stumble over the truth, but most of them pick themselves up and hurry
off as if nothing ever happened.",
            "Never hold discussions with the monkey when the organ grinder is in the room.",
            "Never, never, never believe any war will be smooth and easy, or that anyone who embarks
on the strange voyage can measure the tides and hurricanes he will encounter. The statesman who yields
to war fever must realize that once the signal is given, he is no longer the master of policy but the
slave of unforeseeable and uncontrollable events.",
            "One ought never to turn one's back on a threatened danger and try to run away from it. If
you do that, you will double the danger. But if you meet it promptly and without flinching, you will
reduce the danger by half.",
            "Personally I'm always ready to learn, although I do not always like being taught.",
            "Success is the ability to go from one failure to another with no loss of enthusiasm.",
            "The price of greatness is responsibility.",
            "The reserve of modern assertions is sometimes pushed to extremes, in which the fear of
being contradicted leads the writer to strip himself of almost all sense and meaning.",
            "There are a terrible lot of lies going around the world, and the worst of it is half
```

```
of them are true.",
            "To build may have to be the slow and laborious task of years. To destroy can be the
thoughtless act of a single day.",
            "We make a living by what we get, we make a life by what we give.",
            "When I am abroad, I always make it a rule never to criticize or attack the government
of my own country. I make up for lost time when I come home.",
            "When the eagles are silent, the parrots begin to jabber."
        };

    /** a random number generator */
    private final transient Random rnd;

    /**
     * Constructs a new <code>Quotations</code>.
     */
    public Quotations() {

        super("Quotations", Quotations.class.getName());

        this.outputs.add(new FilterOutput(StringPipeItem.class, "A Random Quotation",
                "quotation"));

        this.rnd = new Random(System.currentTimeMillis());
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new Quotations();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        StringPipeItem out0;

        validateInputs(pipe);

        out0 = new StringPipeItem("quotation", "Quotation", pipe);
        out0.setData(QUOTES[this.rnd.nextInt(QUOTES.length)]);
        pipe.add(out0);
    }
}

package com.srbenoit.filter.filters;

import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ByteArrayPipeItem;
import com.srbenoit.filter.items.StringPipeItem;
import org.bouncycastle.util.encoders.Base64;

/**
 * A filter that converts a byte array into a Base-64 representation of the data.
 */
public class ToBase64 extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = 1418293103288603213L;

    /**
     * Constructs a new <code>ToBase64</code>.
     */
    public ToBase64() {

        super("ToBase64", ToBase64.class.getName());

        this.inputs.add(new FilterInput(ByteArrayPipeItem.class, "Binary data"));
        this.outputs.add(new FilterOutput(StringPipeItem.class, "Base-64 encoded string",
                "base-64"));
    }
```

```
    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new ToBase64();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ByteArrayPipeItem in0;
        StringPipeItem out0;

        validateInputs(pipe);

        in0 = (ByteArrayPipeItem) pipe.get("Binary");

        out0 = new StringPipeItem("encoded", "Base64", pipe);
        out0.setData(new String(Base64.encode(in0.getData())));

        pipe.add(out0);
    }
}
```

E.5.3 Filter Tree Infrastructure (com.srbenoit.filter.items)

This package provides the input and output data classes that filtrers can produce, which are referred to as "Pipe Items". This package relies on the Bouncy Castle cryptographic libraries. To obtain these, download "bcprov-jdk16-146.jar" from the BouncyCastle latest releases page.

```
package com.srbenoit.filter.items;

import java.io.File;
import com.srbenoit.filter.AbstractPipeItem;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.PipeItemFileInfo;

/**
 * A simple pipe item that contains a non-null byte array.
 */
public class ByteArrayPipeItem extends AbstractPipeItem {

    /** the byte array data */
    private byte[] data = null;

    /**
     * Constructs a new <code>ByteArrayPipeItem</code>.
     *
     * @param  theKey    the unique key for the item
     * @param  theLabel  the label for the item (a human friendly name)
     * @param  thePipe   the pipe in which this item is installed
     */
    public ByteArrayPipeItem(final String theKey, final String theLabel, final Pipe thePipe) {

        super(theKey, theLabel, thePipe);

        PipeItemFileInfo info;

        info = new PipeItemFileInfo(makeFile(getSubdir()));
        addFile(info);
```

```
}

/**
 * Sets this item's data.
 *
 * @param  newData  the new data
 */
public void setData(final byte[] newData) {

    if (newData == null) {
        throw new IllegalArgumentException("Data may not be null");
    }

    this.data = newData.clone();
    getFile(0).notPersisted();
}

/**
 * Gets this item's data.
 *
 * @return  the data
 */
public byte[] getData() {

    return this.data.clone();
}

/**
 * Gets the name of this type.
 *
 * @return  the type name
 */
@Override public String typeName() {

    return "byte[]";
}

/**
 * Resets the pipe item to a virgin (empty) state.
 */
@Override public void reset() {

    this.data = null;
    getFile(0).notPersisted();
}

/**
 * Saves the item to a filesystem.
 *
 * @param   executor  the executor that is saving the pipe
 * @param   startPct  the starting progress percentage for the save operation
 * @param   endPct    the ending progress percentage for the save operation
 * @return  <code>true</code> on successful save; <code>false</code> on failure
 */
@Override public boolean save(final FilterTreeExecutor executor, final int startPct,
    final int endPct) {

    PipeItemFileInfo info;
    boolean result;

    executor.indicateProgress((startPct + endPct) / 2);

    if (this.data == null) {
        LOG.warning("Byte array pipe item was marked dirty but had no data");
        result = true;
    } else {
        info = getFile(0);

        if (Pipe.writeFile(info.getFile(), this.data)) {
            info.wasPersisted();
            result = true;
        } else {
            result = false;
        }
    }

    return result;
}

/**
 * Loads the items from the filesystem.
 *
 * @return  <code>true</code> if the load succeeded; <code>false</code> if not
 */
@Override public boolean load() {

    PipeItemFileInfo info;
```

```
        info = getFile(0);

        if (info.getFile().exists()) {
            this.data = Pipe.readFile(info.getFile());

            if (this.data == null) {
                info.notPersisted();
            } else {
                info.wasPersisted();
            }
        } else {
            info.notPersisted();
        }

        return this.data != null;
    }

    /**
     * Creates a file that represents the object data.
     *
     * @param   key   the key associated with the item
     * @param   pipe  the pipe in which this item is being loaded/saved
     * @return  the file
     */
    private File makeFile(final File dir) {

        return new File(dir, getKey() + "_ByteArray.dat");
    }
}

package com.srbenoit.filter.items;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.text.ParseException;
import java.util.List;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import com.srbenoit.filter.AbstractPipeItem;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.PipeItemFileInfo;
import com.srbenoit.xml.ElementBase;
import com.srbenoit.xml.EmptyElement;
import com.srbenoit.xml.Node;
import com.srbenoit.xml.XmlParser;

/**
 * An array of images that are backed by files. This class includes methods to create such an image
 * and retrieve the <code>BufferedImage</code>, to load it from its backing file (if present), and
 * write it to its backing file.
 *
 * <p>We maintain an array of file information objects in parallel with the image objects. While
 * the image may or may not be loaded at any given time, the presence of a file information object
 * with persisted = true indicates that there is a file with the image data and we can load the
 * file.
 */
public class ImageArrayPipeItem extends AbstractPipeItem {

    /** format string for JPEG files */
    public static final String FORMAT_JPEG = "JPEG";

    /** format string for PNG files */
    public static final String FORMAT_PNG = "PNG";

    /** format string for TIFF files */
    public static final String FORMAT_TIFF = "TIFF";

    /** the image (either created by this class, or loaded from the file) */
    private transient BufferedImage[][] images;

    /** the file info for each image file */
    private transient PipeItemFileInfo[][] fileInfo;

    /** the label for the X axis (typically "x" or "t") */
    private transient String xLabel;

    /** the label for the Y axis (typically "y" or "z") */
    private transient String yLabel;

    /** the file suffix for the image file format */
    private transient String type;

    /**
     * Constructs a new empty <code>ImageArrayPipeItem</code>.
```

```
     *
     * @param  theKey     the unique key for the item
     * @param  theLabel   the label for the item (a human friendly name)
     * @param  thePipe    the pipe in which this item is installed
     */
    public ImageArrayPipeItem(final String theKey, final String theLabel, final Pipe thePipe) {

        super(theKey, theLabel, thePipe);

        File file;

        this.images = new BufferedImage[0][0];
        this.fileInfo = new PipeItemFileInfo[0][0];
        this.xLabel = "";
        this.yLabel = "";
        this.type = "";

        ImageIO.scanForPlugins();

        file = makeFile(getSubdir());
        addFile(new PipeItemFileInfo(file));
    }

    /**
     * Constructs a new <code>ImageArrayPipeItem</code> with a newly allocated <code>
     * BufferedImage</code> of a given size and type. The new (blank) image is not written to the
     * file.
     *
     * @param  theKey       the unique key for the item
     * @param  theLabel     the label for the item (a human friendly name)
     * @param  thePipe      the pipe in which this item is installed
     * @param  theXLabel    the label for the X axis (typically "x" or "t")
     * @param  theYLabel    the label for the Y axis (typically "y" or "z")
     * @param  numX         the number of images in the X (or time) direction
     * @param  numY         the number if images in the Y (or plane) direction
     * @param  fileSuffix   the file suffix for images written to disk (determines format)
     */
    public ImageArrayPipeItem(final String theKey, final String theLabel, final Pipe thePipe,
        final String theXLabel, final String theYLabel, final int numX, final int numY,
        final String fileSuffix) {

        super(theKey, theLabel, thePipe);

        File file;

        if (numX < 1) {
            throw new IllegalArgumentException("Number of time points be at least 1");
        }

        if (numY < 1) {
            throw new IllegalArgumentException("Number of planes be at least 1");
        }

        if (fileSuffix == null) {
            throw new IllegalArgumentException("File suffix may not be null");
        }

        this.images = new BufferedImage[numX][numY];
        this.fileInfo = new PipeItemFileInfo[numX][numY];
        this.xLabel = theXLabel;
        this.yLabel = theYLabel;

        setType(fileSuffix);

        file = makeFile(getSubdir());
        addFile(new PipeItemFileInfo(file));
    }

    /**
     * Sets the file type to which images should be saved.
     *
     * @param   type  the file type (extension)
     * @throws  IllegalArgumentException  if the type is not valid
     */
    private void setType(final String type) throws IllegalArgumentException {

        boolean found;
        String[] suffixes;
        StringBuilder str;

        ImageIO.scanForPlugins();
        found = false;
        suffixes = ImageIO.getReaderFileSuffixes();

        for (String test : suffixes) {

            if (test.equals(type)) {
```

```
                found = true;

                break;
            }
        }

        if (!found) {
            str = new StringBuilder(80);
            str.append("Invalid image file suffix '");
            str.append(type);
            str.append("' (supported suffixes are");

            for (String test : suffixes) {
                str.append(' ');
                str.append(test);
            }

            throw new IllegalArgumentException(str.toString());
        }

        this.type = type;
    }

    /**
     * Sets the image at a particular X and Y position.
     *
     * @param  xPos  the X position
     * @param  yPos  the Y position
     * @param  img   the image
     */
    public void setImage(final int xPos, final int yPos, final BufferedImage img) {

        File sub;
        File file;

        this.images[xPos][yPos] = img;

        if (this.fileInfo[xPos][yPos] != null) {
            removeFile(this.fileInfo[xPos][yPos]);
        }

        sub = new File(getPipe().getDir(), getKey());
        file = makeImageFile(sub, xPos, yPos);
        this.fileInfo[xPos][yPos] = new PipeItemFileInfo(file);
        addFile(this.fileInfo[xPos][yPos]);
    }

    /**
     * Sets the file suffix for files written by this pipe.
     *
     * @param  fileSuffix  the file suffix for images written to disk (determines format)
     */
    public void setFileSuffix(final String fileSuffix) {

        File sub;

        if (fileSuffix == null) {
            throw new IllegalArgumentException("File suffix may not be null");
        }

        if (!this.type.equals(fileSuffix)) {
            this.type = fileSuffix;
            getFile(0).notPersisted();

            // We need to rebuild all file info objects and mark everything as not persisted
            sub = getSubdir();

            for (int x = 0; x < this.fileInfo.length; x++) {

                for (int y = 0; y < this.fileInfo[x].length; y++) {

                    if (this.fileInfo[x][y] != null) {
                        this.fileInfo[x][y].setFile(makeImageFile(sub, x, y));
                        this.fileInfo[x][y].notPersisted();
                    }
                }
            }
        }
    }

    /**
     * Gets the size of the image array along the X direction.
     *
     * @return  the array X dimension
     */
    public int getXSize() {
```

```
    return this.images.length;
}

/**
 * Gets the size of the image array along the Y direction.
 *
 * @return  the array Y dimension
 */
public int getYSize() {

    return this.images[0].length;
}

/**
 * Gets the label for the X dimension.
 *
 * @return  the label
 */
public String getXLabel() {

    return this.xLabel;
}

/**
 * Gets the label for the Y dimension.
 *
 * @return  the label
 */
public String getYLabel() {

    return this.yLabel;
}

/**
 * Gets the image at a particular X and Y position. If the image is cached but has not been
 * loaded, this loads the image then returns it.
 *
 * @param   xPos  the X position
 * @param   yPos  the Y position
 * @return  the image, or <code>null</code> if no image has been set
 */
public BufferedImage getImage(final int xPos, final int yPos) {

    File sub;
    File file;

    sub = getSubdir();

    if (this.images[xPos][yPos] == null) {
        file = makeImageFile(sub, xPos, yPos);

        if (file.exists()) {
            this.images[xPos][yPos] = loadImage(file);
        }
    }

    return this.images[xPos][yPos];
}

/**
 * Gets the set of images at a particular X position. If any image in the set is cached but has
 * not been loaded, this loads the image before returning the array.
 *
 * @param   xPos  the X position
 * @return  the images, any of which may be <code>null</code>
 */
public BufferedImage[] getImages(final int xPos) {

    File sub;
    File file;

    sub = getSubdir();

    for (int y = 0; y < this.images[xPos].length; y++) {

        if (this.images[xPos][y] == null) {
            file = makeImageFile(sub, xPos, y);

            if (file.exists()) {
                this.images[xPos][y] = loadImage(file);
            }
        }
    }

    return this.images[xPos];
}
```

```
/**
 * Loads an image from the filesystem.
 *
 * @param   file  the file to load
 * @return  the loaded image
 */
private BufferedImage loadImage(final File file) {

    BufferedImage img;

    try {
        img = ImageIO.read(file);
    } catch (IOException e) {
        LOG.log(Level.WARNING, "Unable to load image " + file.getAbsolutePath(), e);
        img = null;
    }

    return img;
}

/**
 * Gets a human-friendly name for the data type. For example, a list of sets of images
 * representing a time series of z-planes might return "Multi-plane image sequence".
 *
 * @return  the name of the data type this item represents
 */
@Override public String typeName() {

    return "Image Array";
}

/**
 * Resets the pipe item to a virgin (empty) state.
 */
@Override public void reset() {

    for (int x = 0; x < this.images.length; x++) {

        for (int y = 0; y < this.images[x].length; y++) {
            this.images[x][y] = null;
            this.fileInfo[x][y] = null;
        }
    }

    removeAllFilesButFirst();

    getFile(0).notPersisted();
}

/**
 * Saves the item to a filesystem.
 *
 * @param   executor  the executor that is saving the pipe
 * @param   startPct  the starting progress percentage for the save operation
 * @param   endPct    the ending progress percentage for the save operation
 * @return  <code>true</code> if the save succeeded; <code>false</code> if not
 */
@Override public boolean save(final FilterTreeExecutor executor, final int startPct,
    final int endPct) {

    int total;
    int soFar;
    File sub;
    StringBuilder str;
    boolean result;
    PipeItemFileInfo info;

    sub = getSubdir();

    if (!sub.exists()) {
        sub.mkdir();
    }

    // Count the number of files we need to write
    total = 0;

    for (int x = 0; x < this.images.length; x++) {

        for (int y = 0; y < this.images[x].length; y++) {

            info = this.fileInfo[x][y];

            if ((info != null) && (!info.isPersisted())) {
                total++;
            }
        }
    }
```

```
        result = true;
        soFar = 0;

        for (int x = 0; x < this.images.length; x++) {

            for (int y = 0; y < this.images[x].length; y++) {

                info = this.fileInfo[x][y];

                if ((info != null) && (!info.isPersisted())) {

                    soFar++;
                    executor.indicateProgress(startPct + (soFar * (endPct - startPct) / total));
                    result = persistImage(sub, x, y);

                    if (result) {
                        info.wasPersisted();
                    } else {
                        break;
                    }
                }
            }
        }

        if (result) {
            executor.indicateProgress(endPct);

            info = getFile(0);
            str = new StringBuilder(500);
            str.append("<image-array x-len='");
            str.append(Integer.toString(this.images.length));
            str.append("' x-lbl='");
            str.append(ElementBase.encode(this.xLabel));
            str.append("' y-len='");
            str.append(Integer.toString(this.images[0].length));
            str.append("' y-lbl='");
            str.append(ElementBase.encode(this.yLabel));
            str.append("' type='");
            str.append(ElementBase.encode(this.type));
            str.append("'/>");

            result = Pipe.writeFile(info.getFile(), str.toString().getBytes());

            if (result) {
                info.wasPersisted();
            } else {
                info.notPersisted();
            }
        }

        return result;
    }

    /**
     * Writes a single image file to the filesystem.
     *
     * @param   sub   the directory to which to save the item's image files
     * @param   xPos  the X index of the image to write
     * @param   yPos  the Y index of the image to write
     * @return  <code>true</code> if all non-null images were written to the disk successfully;
     *          <code>false</code> if not
     */
    private boolean persistImage(final File sub, final int xPos, final int yPos) {

        File file;
        boolean result;

        file = makeImageFile(sub, xPos, yPos);

        try {
            ImageIO.write(this.images[xPos][yPos], this.type, file);
            result = true;
        } catch (IOException e) {
            LOG.log(Level.WARNING, "Exception writing image file", e);
            result = false;
        }

        return result;
    }

    /**
     * Loads the items from the filesystem
     *
     * @return  <code>true</code> if the load succeeded; <code>false</code> if not
     */
    @Override public boolean load() {
```

```
        File sub;
        File file;
        byte[] bytes;
        List<Node> nodes;
        EmptyElement empty;
        String xLen;
        String yLen;
        int width;
        int height;
        PipeItemFileInfo info;
        boolean result;

        reset();

        sub = getSubdir();
        info = getFile(0);

        bytes = Pipe.readFile(info.getFile());

        if (bytes == null) {
            LOG.log(Level.INFO, "Unable to read image array file: {0}",
                info.getFile().getAbsolutePath());
            info.notPersisted();
            result = false;
        } else {

            try {
                nodes = new XmlParser().parse(new String(bytes), true);

                if ((nodes.size() == 1) && (nodes.get(0) instanceof EmptyElement)) {
                    empty = (EmptyElement) nodes.get(0);
                    xLen = empty.get("x-len");
                    yLen = empty.get("y-len");
                    this.xLabel = empty.get("x-lbl");
                    this.yLabel = empty.get("y-lbl");

                    if ((this.xLabel == null) || (this.yLabel == null)) {
                        LOG.warning("Missing x-lbl or y-lbl in ImageArray xml file");
                        result = false;
                    } else {
                        setType(empty.get("type"));

                        try {
                            width = Integer.parseInt(xLen);
                            height = Integer.parseInt(yLen);

                            if ((width != this.images.length)
                                    || (height != this.images[0].length)) {
                                this.images = new BufferedImage[width][height];
                                this.fileInfo = new PipeItemFileInfo[width][height];
                            }

                            info.wasPersisted();

                            result = true;

                        } catch (NumberFormatException e) {
                            LOG.warning("Invalid length while loading ImageArray");
                            result = false;
                        }
                    }
                } else {
                    LOG.warning("Unable to parse ImageArray xml file");
                    result = false;
                }
            } catch (ParseException e) {
                LOG.warning("Unable to parse ImageArray xml file");
                result = false;
            }
        }

        if (result) {

outer:
            for (int x = 0; x < this.images.length; x++) {

                for (int y = 0; y < this.images[x].length; y++) {
                    file = makeImageFile(sub, x, y);

                    if (file.exists()) {
                        info = new PipeItemFileInfo(file); // NOPMD SRB
                        info.wasPersisted();
                        addFile(info);
                    }
                }
            }
```

```
        }

        return result;
    }

    /**
     * Creates a file that represents the object data.
     *
     * @param   key   the key associated with the item
     * @param   pipe  the pipe in which this item is being loaded/saved
     * @return  the file
     */
    private File makeFile(final File dir) {

        return new File(dir, getKey() + "_ImageArray.xml");
    }

    /**
     * Creates a file that will be used to store one image.
     *
     * @param   key      the key associated with the item
     * @param   pipe     the pipe in which this item is being loaded/saved
     * @param   xIndex   the X index of the image whose filename to construct
     * @param   yIndex   the Y index of the image whose filename to construct
     * @return  the file
     */
    private File makeImageFile(final File dir, final int xIndex, final int yIndex) {

        return new File(dir,
                getKey() + "_Image_" + this.xLabel + xIndex + this.yLabel + yIndex + "."
                + this.type);
    }
}

package com.srbenoit.filter.items;

/**
 * A point within an image that stores the point position, the velocity of the point, and the
 * velocity of the ambient surroundings of the point.
 */
public class ImagePoint {

    /** the X coordinate of the point position */
    private final int xPos;

    /** the Y coordinate of the point position */
    private final int yPos;

    /** the X component of the point velocity */
    private int xVel;

    /** the Y component of the point velocity */
    private int yVel;

    /** the X component of the ambient velocity near the point */
    private int xAmbientVel;

    /** the Y component of the ambient velocity near the point */
    private int yAmbientVel;

    /**
     * Constructs a new <code>ImagePoint</code> with zero velocities.
     *
     * @param  xCoord  the X coordinate of the point position
     * @param  yCoord  the Y coordinate of the point position
     */
    public ImagePoint(final int xCoord, final int yCoord) {

        this.xPos = xCoord;
        this.yPos = yCoord;
        this.xVel = 0;
        this.yVel = 0;
        this.xAmbientVel = 0;
        this.yAmbientVel = 0;
    }

    /**
     * Constructs a new <code>ImagePoint</code>.
     *
     * @param  xCoord             the X coordinate of the point position
     * @param  yCoord             the Y coordinate of the point position
     * @param  xVelocity          the X component of the point velocity
     * @param  yVelocity          the Y component of the point velocity
     * @param  xAmbientVelocity   the X component of the ambient velocity near the point
     * @param  yAmbientVelocity   the Y component of the ambient velocity near the point
     */
    public ImagePoint(final int xCoord, final int yCoord, final int xVelocity, final int yVelocity,
```

```
        final int xAmbientVelocity, final int yAmbientVelocity) {

        this.xPos = xCoord;
        this.yPos = yCoord;
        this.xVel = xVelocity;
        this.yVel = yVelocity;
        this.xAmbientVel = xAmbientVelocity;
        this.yAmbientVel = yAmbientVelocity;
    }

    /**
     * Gets the X coordinate of the point position.
     *
     * @return  the X coordinate
     */
    public int getXPos() {

        return this.xPos;
    }

    /**
     * Gets the Y coordinate of the point position.
     *
     * @return  the Y coordinate
     */
    public int getYPos() {

        return this.yPos;
    }

    /**
     * Gets the X component of the point velocity.
     *
     * @return  the X component
     */
    public int getXVel() {

        return this.xVel;
    }

    /**
     * Gets the Y component of the point velocity.
     *
     * @return  the Y component
     */
    public int getYVel() {

        return this.yVel;
    }

    /**
     * Gets the X component of the ambient velocity near the point.
     *
     * @return  the X component
     */
    public int getXAmbientVel() {

        return this.xAmbientVel;
    }

    /**
     * Gets the Y component of the ambient velocity near the point.
     *
     * @return  the Y component
     */
    public int getYAmbientVel() {

        return this.yAmbientVel;
    }

    /**
     * Sets the new velocity of the point
     *
     * @param  newXVel  the X component of the velocity
     * @param  newYVel  the Y component of the velocity
     */
    public void setVel(final int newXVel, final int newYVel) {

        this.xVel = newXVel;
        this.yVel = newYVel;
    }

    /**
     * Sets the new ambient velocity of the point
     *
     * @param  newXVel  the X component of the velocity
     * @param  newYVel  the Y component of the velocity
```

```
     */
    public void setAmbientVel(final int newXVel, final int newYVel) {

        this.xAmbientVel = newXVel;
        this.yAmbientVel = newYVel;
    }
}

package com.srbenoit.filter.items;

import java.io.File;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.filter.AbstractPipeItem;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.PipeItemFileInfo;
import com.srbenoit.xml.EmptyElement;
import com.srbenoit.xml.Node;
import com.srbenoit.xml.NonemptyElement;
import com.srbenoit.xml.XmlParser;

/**
 * An array of point sets. The intent is that an image array has a list of points associated with
 * each image, so we would have a point set array of the same dimensions as our image array.
 */
public class PointSetArrayPipeItem extends AbstractPipeItem {

    /** end of line characters */
    private static final String CRLF;

    /** the point sets - need list of list of lists to maintain type safety. */
    private transient List<List<List<ImagePoint>>> pointSets;

    static {
        String crlf;

        crlf = System.getProperty("line.separator");
        CRLF = (crlf == null) ? "\n" : crlf;
    }

    /**
     * Constructs a new empty <code>PointSetArrayPipeItem</code>.
     *
     * @param  theKey     the unique key for the item
     * @param  theLabel   the label for the item (a human friendly name)
     * @param  thePipe    the pipe in which this item is installed
     */
    public PointSetArrayPipeItem(final String theKey, final String theLabel, final Pipe thePipe) {

        super(theKey, theLabel, thePipe);

        PipeItemFileInfo info;

        buildPointLists(0, 0);

        info = new PipeItemFileInfo(makeFile(getSubdir()));
        addFile(info);
    }

    /**
     * Constructs a new <code>PointSetArrayPipeItem</code> of a given size and type.
     *
     * @param  theKey     the unique key for the item
     * @param  theLabel   the label for the item (a human friendly name)
     * @param  thePipe    the pipe in which this item is installed
     * @param  numX       the number of images in the X (or time) direction
     * @param  numY       the number if images in the Y (or plane) direction
     */
    public PointSetArrayPipeItem(final String theKey, final String theLabel, final Pipe thePipe,
        final int numX, final int numY) {

        super(theKey, theLabel, thePipe);

        PipeItemFileInfo info;

        if ((numX < 1) || (numY < 1)) {
            throw new IllegalArgumentException("Each array dimension must be at least 1");
        }

        buildPointLists(numX, numY);

        info = new PipeItemFileInfo(makeFile(getSubdir()));
        addFile(info);
    }
```

```
/**
 * Gets the size of the image array along the X direction.
 *
 * @return  the array X dimension
 */
public int getXSize() {

    return this.pointSets.size();
}

/**
 * Gets the size of the image array along the Y direction.
 *
 * @return  the array Y dimension
 */
public int getYSize() {

    return this.pointSets.get(0).size();
}

/**
 * Gets a human-friendly name for the data type. For example, a list of sets of images
 * representing a time series of z-planes might return "Multi-plane image sequence".
 *
 * @return  the name of the data type this item represents
 */
@Override public String typeName() {

    return "Point Set Array";
}

/**
 * Resets the pipe item to a virgin (empty) state.
 */
@Override public void reset() {

    for (List<List<ImagePoint>> list : this.pointSets) {

        for (List<ImagePoint> inner : list) {
            inner.clear();
        }
    }

    getFile(0).notPersisted();
}

/**
 * Gets the number of points in the point set at a particular X and Y position.
 *
 * @param   xPos  the X position
 * @param   yPos  the Y position
 * @return  the number of points in the point set
 */
public int getNumPoints(final int xPos, final int yPos) {

    return this.pointSets.get(xPos).get(yPos).size();
}

/**
 * Gets a point from the point set at a particular X and Y position.
 *
 * @param   xPos    the X position
 * @param   yPos    the Y position
 * @param   index   the point index to retrieve
 * @return  the point
 */
public ImagePoint getPoint(final int xPos, final int yPos, final int index) {

    return this.pointSets.get(xPos).get(yPos).get(index);
}

/**
 * Clears the point set at a particular X and Y position.
 *
 * @param  xPos  the X position
 * @param  yPos  the Y position
 */
public void clear(final int xPos, final int yPos) {

    this.pointSets.get(xPos).get(yPos).clear();
    getFile(0).notPersisted();
}

/**
 * Adds a point to the point set at a particular X and Y position.
 *
```

```
     * @param  xPos    the X position
     * @param  yPos    the Y position
     * @param  point   the point to add
     */
    public void addPoint(final int xPos, final int yPos, final ImagePoint point) {

        this.pointSets.get(xPos).get(yPos).add(point);
        getFile(0).notPersisted();
    }

    /**
     * Sets the point at a particular index in a point set at a particular X and Y position.
     *
     * @param  xPos    the X position
     * @param  yPos    the Y position
     * @param  index   the index of the point to set
     * @param  point   the point to add
     */
    public void setPoint(final int xPos, final int yPos, final int index, final ImagePoint point) {

        this.pointSets.get(xPos).get(yPos).set(index, point);
        getFile(0).notPersisted();
    }

    /**
     * Saves the item to a filesystem.
     *
     * @param   executor  the executor that is saving the pipe
     * @param   startPct  the starting progress percentage for the save operation
     * @param   endPct    the ending progress percentage for the save operation
     * @return  <code>true</code> if the save succeeded; <code>false</code> if not
     */
    @Override public boolean save(final FilterTreeExecutor executor, final int startPct,
        final int endPct) {

        PipeItemFileInfo info;
        StringBuilder str;
        boolean result;
        List<ImagePoint> points;

        executor.indicateProgress((startPct + endPct) / 2);

        info = getFile(0);

        str = new StringBuilder(500);
        str.append("<point-set-array x-len='");
        str.append(Integer.toString(getXSize()));
        str.append("' y-len='");
        str.append(Integer.toString(getYSize()));
        str.append("'>");
        str.append(CRLF);

        for (int x = 0; x < getXSize(); x++) {

            for (int y = 0; y < getYSize(); y++) {
                str.append(" <point-set x='");
                str.append(x);
                str.append("' y='");
                str.append(y);
                str.append("'>");
                str.append(CRLF);

                points = this.pointSets.get(x).get(y);

                for (ImagePoint point : points) {
                    str.append(" <point x='");
                    str.append(point.getXPos());
                    str.append("' y='");
                    str.append(point.getYPos());
                    str.append("' vx='");
                    str.append(point.getXVel());
                    str.append("' vy='");
                    str.append(point.getYVel());
                    str.append("' vxAmbient='");
                    str.append(point.getXAmbientVel());
                    str.append("' vyAmbient='");
                    str.append(point.getYAmbientVel());
                    str.append("'/>");
                    str.append(CRLF);
                }

                str.append("</point-set>");
                str.append(CRLF);
            }
        }

        str.append("</point-set-array>");
```

```
        str.append(CRLF);

        if (Pipe.writeFile(info.getFile(), str.toString().getBytes())) {
            info.wasPersisted();
            result = true;
        } else {
            info.notPersisted();
            result = false;
        }

        return result;
    }

    /**
     * Loads the items from the filesystem.
     *
     * @return  <code>true</code> if the load succeeded; <code>false</code> if not
     */
    @Override public boolean load() {

        PipeItemFileInfo info;
        byte[] bytes;
        List<Node> nodes;
        NonemptyElement nonempty;
        String xLen;
        String yLen;
        int numX;
        int numY;
        boolean result;

        info = getFile(0);

        reset();

        bytes = Pipe.readFile(info.getFile());

        if (bytes == null) {
            result = false;
            info.notPersisted();
        } else {

            try {
                nodes = new XmlParser().parse(new String(bytes), true);

                if ((nodes.size() == 1) && (nodes.get(0) instanceof NonemptyElement)) {
                    nonempty = (NonemptyElement) nodes.get(0);
                    xLen = nonempty.get("x-len");
                    yLen = nonempty.get("y-len");

                    try {
                        numX = Integer.parseInt(xLen);
                        numY = Integer.parseInt(yLen);

                        if (nonempty.children.size() == (numX * numY)) {
                            buildPointLists(numX, numY);
                            result = populatePointLists(nonempty.children);
                        } else {
                            LOG.warning(
                                "Invalid number of <point-set> entries while loading PointSetArray");
                            result = false;
                        }
                    } catch (NumberFormatException e) {
                        LOG.warning("Invalid length while loading PointSetArray");
                        result = false;
                    }
                } else {
                    LOG.warning("Unable to parse PointSetArray xml file");
                    result = false;
                }
            } catch (ParseException e) {
                LOG.warning("Unable to parse PointSetArray xml file");
                result = false;
            }

            if (result) {
                info.wasPersisted();
            } else {
                info.notPersisted();
            }
        }

        return result;
    }

    /**
     * Builds the list of lists of points that stores the data.
     *
```

```
     * @param  numX  the number of point sets along the X axis
     * @param  numY  the number of point sets along the Y axis
     */
    private void buildPointLists(final int numX, final int numY) {

        List<List<ImagePoint>> list;

        // First list is indexed by 'x'
        this.pointSets = new ArrayList<List<List<ImagePoint>>>(numX);

        for (int i = 0; i < numX; i++) {

            // Second list is indexed by 'y'
            list = new ArrayList<List<ImagePoint>>(numY); // NOPMD SRB
            this.pointSets.add(list);

            for (int j = 0; j < numY; j++) {

                // Create the interior point lists (empty)
                list.add(new ArrayList<ImagePoint>(20)); // NOPMD SRB
            }
        }
    }

    /**
     * Populates the points lists from a parsed XML node that should contain only "point-set"
     * elements.
     *
     * @param   nodes  the list of "point-set" nodes
     * @return  <code>true</code> if successful loading, <code>false</code> otherwise
     */
    private boolean populatePointLists(final List<Node> nodes) {

        NonemptyElement nonempty;
        String xStr;
        String yStr;
        int xPos;
        int yPos;
        List<ImagePoint> list;
        boolean result = true;

        for (Node node : nodes) {

            if (node instanceof NonemptyElement) {
                nonempty = (NonemptyElement) node;

                if ("point-set".equals(nonempty.tagName)) {
                    xStr = nonempty.get("x");
                    yStr = nonempty.get("y");

                    try {
                        xPos = Integer.parseInt(xStr);
                        yPos = Integer.parseInt(yStr);

                        list = this.pointSets.get(xPos).get(yPos);
                        result = populatePoints(list, nonempty);

                        if (!result) {
                            break;
                        }

                    } catch (NumberFormatException e) {
                        LOG.warning(
                                "Invalid x/y coordinate in <point-set> while loading PointSetArray");
                        result = false;

                        break;
                    }
                } else {
                    LOG.log(Level.WARNING, "Invalid element ''{0}'' while loading PointSetArray",
                            nonempty.tagName);
                    result = false;

                    break;
                }
            } else {
                LOG.warning("Invalid element while loading PointSetArray");
                result = false;

                break;
            }
        }

        return result;
    }

    /**
```

```
     * Populates a single point set from "point" elements in a parsed XML node.
     *
     * @param   points    the list to which to add the extracted points
     * @param   nonemtpy  the list containing the "point" elements
     * @return  <code>true</code> if successful loading, <code>false</code> otherwise
     */
    private boolean populatePoints(final List<ImagePoint> points, final NonemptyElement nonempty) {

        EmptyElement empty;
        String xStr;
        String yStr;
        String vxStr;
        String vyStr;
        String vxAmbStr;
        String vyAmbStr;
        int xPos;
        int yPos;
        int xVel;
        int yVel;
        int xVelAmb;
        int yVelAmb;
        boolean result = true;

        for (Node node : nonempty.children) {

            if (node instanceof EmptyElement) {
                empty = (EmptyElement) node;

                if ("point".equals(empty.tagName)) {

                    xStr = empty.get("x");
                    yStr = empty.get("y");
                    vxStr = empty.get("vx");
                    vyStr = empty.get("vy");
                    vxAmbStr = empty.get("vxAmbient");
                    vyAmbStr = empty.get("vyAmbient");

                    try {
                        xPos = Integer.parseInt(xStr);
                        yPos = Integer.parseInt(yStr);
                        xVel = Integer.parseInt(vxStr);
                        yVel = Integer.parseInt(vyStr);
                        xVelAmb = Integer.parseInt(vxAmbStr);
                        yVelAmb = Integer.parseInt(vyAmbStr);
                        points.add(new ImagePoint(xPos, yPos, xVel, yVel, xVelAmb, yVelAmb)); // NOPMD SRB
                    } catch (NumberFormatException e) {
                        LOG.warning(
                            "Invalid x/y coordinate in <point> while loading PointSetArray");
                        result = false;

                        break;
                    }
                } else {
                    LOG.log(Level.WARNING, "Invalid element ''{0}'' while loading PointSetArray",
                        empty.tagName);
                    result = false;

                    break;
                }
            } else {
                LOG.warning("Invalid element while loading PointSetArray");
                result = false;

                break;
            }
        }

        return result;
    }

    /**
     * Creates a file that represents the object data.
     *
     * @param   dir  the directory where the pipe's items are stored
     * @return  the file
     */
    private File makeFile(final File dir) {

        return new File(dir, getKey() + "_PointSetArray.xml");
    }
}

package com.srbenoit.filter.items;

import java.io.File;
import java.util.logging.Level;
import com.srbenoit.filter.AbstractPipeItem;
```

```
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.PipeItemFileInfo;

/**
 * A simple pipe item that contains a non-null string.
 */
public class StringPipeItem extends AbstractPipeItem {

    /** the string data */
    private String data = null;

    /**
     * Constructs a new <code>StringPipeItem</code>.
     *
     * @param  theKey     the unique key for the item
     * @param  theLabel   the label for the item (a human friendly name)
     * @param  thePipe    the pipe in which this item is installed
     */
    public StringPipeItem(final String theKey, final String theLabel, final Pipe thePipe) {

        super(theKey, theLabel, thePipe);

        PipeItemFileInfo info;

        info = new PipeItemFileInfo(makeFile(getSubdir()));
        addFile(info);
    }

    /**
     * Sets this item's data.
     *
     * @param  newData  the new data
     */
    public void setData(final String newData) {

        if (newData == null) {
            throw new IllegalArgumentException("Data may not be null");
        }

        this.data = newData;
        getFile(0).notPersisted();
    }

    /**
     * Gets this item's data.
     *
     * @return  the data
     */
    public String getData() {

        return this.data;
    }

    /**
     * Gets the name of this type.
     *
     * @return  the type name
     */
    @Override public String typeName() {

        return "String";
    }

    /**
     * Resets the pipe item to a virgin (empty) state.
     */
    @Override public void reset() {

        this.data = null;
        getFile(0).notPersisted();
    }

    /**
     * Saves the item to a filesystem.
     *
     * @param    executor  the executor that is saving the pipe
     * @param    startPct  the starting progress percentage for the save operation
     * @param    endPct    the ending progress percentage for the save operation
     * @return  <code>true</code> on successful save; <code>false</code> on failure
     */
    @Override public boolean save(final FilterTreeExecutor executor, final int startPct,
        final int endPct) {

        PipeItemFileInfo info;
        boolean result;
```

```
        executor.indicateProgress((startPct + endPct) / 2);

        info = getFile(0);

        if (this.data == null) {
            result = false;
            info.notPersisted();
        } else {

            if (Pipe.writeFile(info.getFile(), this.data.getBytes())) {
                info.wasPersisted();
                result = true;
            } else {
                info.notPersisted();
                result = false;
            }
        }

        return result;
    }

    /**
     * Loads the items from the filesystem.
     *
     * @return  <code>true</code> if the load succeeded; <code>false</code> if not
     */
    @Override public boolean load() {

        PipeItemFileInfo info;
        byte[] bytes;

        info = getFile(0);

        bytes = Pipe.readFile(info.getFile());

        if (bytes == null) {
            LOG.log(Level.INFO, "Unable to read string file: {0}",
                info.getFile().getAbsolutePath());
            this.data = null;
            info.notPersisted();
        } else {
            this.data = new String(bytes);
            info.wasPersisted();
        }

        return this.data != null;
    }

    /**
     * Creates a file that represents the object data.
     *
     * @param   dir  the directory where the item's files are stored
     * @return  the file
     */
    private File makeFile(final File dir) {

        return new File(dir, getKey() + "_String.txt");
    }
}

package com.srbenoit.filter.items;

import java.io.File;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.logging.Level;
import com.srbenoit.filter.AbstractPipeItem;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.PipeItemFileInfo;
import com.srbenoit.util.LocalTime;
import com.srbenoit.xml.ElementBase;
import com.srbenoit.xml.EmptyElement;
import com.srbenoit.xml.Node;
import com.srbenoit.xml.NonemptyElement;
import com.srbenoit.xml.XmlParser;

/**
 * An ordered series of points in time (specified as local date/times), with the ability to
 * designate named subsequences.
 */
public class TimeSeriesPipeItem extends AbstractPipeItem {

    /** version number for serialization */
```

```
private static final long serialVersionUID = -7300471956483249199L;

/** zero-length String array for use in list to array conversion */
private static final String[] STRING_0 = new String[0];

/** zero-length local time array for use in list to array conversion */
private static final LocalTime[] LOCALTIME_0 = new LocalTime[0];

/** zero-length local time array for use in list to array conversion */
private static final String VALUE_TAG = "value";

/** end of line characters */
private static final String CRLF;

/** the list of times */
private final transient List<LocalTime> times;

/** the list of named subsequences */
private final transient Map<String, List<LocalTime>> subsequences;

static {
    String crlf;

    crlf = System.getProperty("line.separator");
    CRLF = (crlf == null) ? "\n" : crlf;
}

/**
 * Constructs a new <code>TimeSeriesPipeItem</code>.
 *
 * @param  theKey     the unique key for the item
 * @param  theLabel   the label for the item (a human friendly name)
 * @param  thePipe    the pipe in which this item is installed
 */
public TimeSeriesPipeItem(final String theKey, final String theLabel, final Pipe thePipe) {

    super(theKey, theLabel, thePipe);

    PipeItemFileInfo info;

    this.times = new ArrayList<LocalTime>(50);
    this.subsequences = new TreeMap<String, List<LocalTime>>();

    info = new PipeItemFileInfo(makeFile(getSubdir()));
    addFile(info);
}

/**
 * Adds a time point to the end of the series.
 *
 * @param  time  the time point to add
 */
public void addTimePoint(final LocalTime time) {

    this.times.add(time);
    getFile(0).notPersisted();
}

/**
 * Adds a time point at a particular position in the series.
 *
 * @param  index  the position at which to add the time point
 * @param  time   the time point to add
 */
public void addTimePoint(final int index, final LocalTime time) {

    this.times.add(index, time);
    getFile(0).notPersisted();
}

/**
 * Replaces the time point at the specified position in this time series with the specified
 * element, and also replaces it in any named subsequences of which the value being replaced
 * was a part.
 *
 * @param   index    index of the time point to replace
 * @param   element  time point to be stored at the specified position
 * @return  the time point previously at the specified position
 */
public LocalTime setTimePoint(final int index, final LocalTime element) {

    LocalTime oldValue;
    List<LocalTime> list;

    oldValue = this.times.set(index, element);

    if (oldValue != null) {
```

```
        for (String key : this.subsequences.keySet()) {
            list = this.subsequences.get(key);

            if (list.remove(oldValue)) {
                list.add(element);
            }
        }
    }

    getFile(0).notPersisted();

    return oldValue;
}

/**
 * Removes the time point of the time series at the specified position in this list, and
 * removes that element from any named subsequences of which it is a part.
 *
 * @param   index  the index of the time point to be removed
 * @return  the time point that was removed from the list
 */
public LocalTime removeTimePoint(final int index) {

    LocalTime oldValue;

    oldValue = this.times.remove(index);

    if (oldValue != null) {

        for (String key : this.subsequences.keySet()) {
            this.subsequences.get(key).remove(oldValue);
        }

        getFile(0).notPersisted();
    }

    return oldValue;
}

/**
 * Removes the first occurrence of the specified time point from this time series, if it is
 * present, and also removes the time point from any named subsequence of which it is a part.
 *
 * @param   obj  time point to be removed from this list, if present
 * @return  <code>true</code> if this list contained the specified time point
 */
public boolean removeTimePoint(final LocalTime obj) {

    boolean found;

    found = this.times.remove(obj);

    if (found) {

        for (String key : this.subsequences.keySet()) {
            this.subsequences.get(key).remove(obj);
        }

        getFile(0).notPersisted();
    }

    return found;
}

/**
 * Gets the number of time points in the series
 *
 * @return  the number of time points
 */
public int getNumTimePoints() {

    return this.times.size();
}

/**
 * Get a particular time points from the series.
 *
 * @param   index  the index of the time point to get
 * @return  the time point
 */
public LocalTime getTimePoint(final int index) {

    return this.times.get(index);
}

/**
```

```
     * Gets the current ordered list of time points in the series.
     *
     * @return  the list of time points
     */
    public LocalTime[] getTimePoints() {

        return this.times.toArray(LOCALTIME_0);
    }

    /**
     * Creates a new named subsequence.
     *
     * @param  name  the name of the subsequence
     */
    public void createSubsequence(final String name) {

        if (!this.subsequences.containsKey(name)) {
            this.subsequences.put(name, new ArrayList<LocalTime>(50));
            getFile(0).notPersisted();
        }
    }

    /**
     * Returns the list of names of subsequences.
     *
     * @return  the list of names
     */
    public String[] getSubsequencNames() {

        return this.subsequences.keySet().toArray(STRING_0);
    }

    /**
     * Adds a time to a subsequence.
     *
     * @param  name  the name of the subsequence
     * @param  time  the time to add
     */
    public void addToSubsequence(final String name, final LocalTime time) {

        List<LocalTime> list;

        list = this.subsequences.get(name);

        if (list == null) {
            throw new IllegalArgumentException("No subsequence was found named '" + name + "'");
        }

        if (!list.contains(time)) {
            list.add(time);
            getFile(0).notPersisted();
        }
    }

    /**
     * Gets the elements of a named subsequence in the order in which they appear in the time
     * series.
     *
     * @param   name  the name of the subsequence
     * @return  the elements in the subsequence, or <code>null</code> if there is no subsequence
     *          with the specified name
     */
    public LocalTime[] getSubsequenceTimes(final String name) {

        List<LocalTime> list;
        LocalTime[] array;
        int index;

        list = this.subsequences.get(name);

        if (list == null) {
            array = null;
        } else {
            array = new LocalTime[list.size()];
            index = 0;

            for (LocalTime test : this.times) {

                if (list.contains(test)) {
                    array[index] = test;
                    index++;
                }
            }
        }

        return array;
    }
```

```
/**
 * Removes a time from a subsequence.
 *
 * @param  name  the name of the subsequence
 * @param  time  the time to remove
 */
public void removeFromSubsequence(final String name, final LocalTime time) {

    List<LocalTime> list;

    list = this.subsequences.get(name);

    if (list == null) {
        throw new IllegalArgumentException("No subsequence was found named '" + name + "'");
    }

    if (list.remove(time)) {
        getFile(0).notPersisted();
    }
}

/**
 * Gets a human-friendly name for the data type. For example, a list of sets of images
 * representing a time series of z-planes might return "Multi-plane image sequence".
 *
 * @return  the name of the data type this item represents
 */
@Override public String typeName() {

    return "Time Series";
}

/**
 * Resets the pipe item to a virgin (empty) state.
 */
@Override public void reset() {

    this.times.clear();

    for (String key : this.subsequences.keySet()) {
        this.subsequences.get(key).clear();
    }

    getFile(0).notPersisted();
}

/**
 * Saves the item to a filesystem.
 *
 * @param   executor  the executor that is saving the pipe
 * @param   startPct  the starting progress percentage for the save operation
 * @param   endPct    the ending progress percentage for the save operation
 * @return  <code>true</code> on successful save; <code>false</code> on failure
 */
@Override public boolean save(final FilterTreeExecutor executor, final int startPct,
    final int endPct) {

    StringBuilder str;
    List<LocalTime> seq;
    PipeItemFileInfo info;
    boolean result;

    executor.indicateProgress((startPct + endPct) / 2);

    if (this.times.isEmpty()) {
        result = false;
    } else {
        str = new StringBuilder(500);

        str.append("<time-series>");
        str.append(CRLF);

        // Write out the ordered list of time points
        for (LocalTime time : this.times) {
            str.append("  ");
            str.append("<time ");
            str.append(VALUE_TAG);
            str.append("='");
            str.append(time.toString());
            str.append("'/>");
            str.append(CRLF);
        }

        // Write out any subsequences
        for (String subseq : this.subsequences.keySet()) {
            seq = this.subsequences.get(subseq);
```

```
            str.append("  ");
            str.append("<subsequence name='");
            str.append(ElementBase.encode(subseq));
            str.append("'>");
            str.append(CRLF);

            for (LocalTime time : seq) {
                str.append("    ");
                str.append("<time ");
                str.append(VALUE_TAG);
                str.append("='");
                str.append(time.toString());
                str.append("</>");
                str.append(CRLF);
            }

            str.append("  ");
            str.append("</subsequence>");
        }

        str.append("</time-series>");
        str.append(CRLF);

        info = getFile(0);

        if (Pipe.writeFile(info.getFile(), str.toString().getBytes())) {
            info.wasPersisted();
            result = true;
        } else {
            info.notPersisted();
            result = false;
        }
    }

    return result;
}

/**
 * Loads the items from the filesystem
 *
 * @return  <code>true</code> if the load succeeded; <code>false</code> if not
 */
@Override public boolean load() {

    PipeItemFileInfo info;
    byte[] data;
    XmlParser parser;
    List<Node> parsed;
    NonemptyElement top;
    boolean result;

    info = getFile(0);

    reset();

    data = Pipe.readFile(info.getFile());

    if (data == null) {
        LOG.log(Level.INFO, "Unable to read string file: {0}",
            info.getFile().getAbsolutePath());
        info.notPersisted();
        result = false;
    } else {
        parser = new XmlParser();

        try {
            parsed = parser.parse(new String(data), true);

            // Should be one node at top level, "time-series"
            if ((parsed != null) && (parsed.size() == 1)
                    && (parsed.get(0) instanceof NonemptyElement)) {

                top = (NonemptyElement) parsed.get(0);

                if ("time-series".equals(top.tagName)) {
                    result = processTopLevel(top);
                } else {
                    LOG.warning("Missing <time-series> element");
                    result = false;
                }
            } else {
                LOG.warning("Missing <time-series> element");
                result = false;
            }

        } catch (ParseException e) {
            LOG.log(Level.WARNING, "Exception while parsing XML", e);
```

```
            result = false;
        }

        if (result) {
            info.wasPersisted();
        } else {
            info.notPersisted();
        }
    }

    return result;
}

/**
 * Processes the top-level "time-series" node by verifying that all contained nodes are either
 * <time> or <sequence> nodes, and parsing them as they are found.
 *
 * @param   top  the top-level node to parse
 * @return  <code>true</code> if the node is valid; <code>false</code> if not
 */
private boolean processTopLevel(final NonemptyElement top) {

    NonemptyElement nonempty;
    EmptyElement empty;
    boolean result;

    result = true;

    for (Node node : top.children) {

        if (node instanceof NonemptyElement) {
            nonempty = (NonemptyElement) node;

            if ("subsequence".equals(nonempty.tagName)) {
                result = processSubsequence(nonempty);
            } else {
                LOG.log(Level.WARNING, "Invalid element ''{0}'' within <time-series>",
                    nonempty.tagName);
                result = false;

                break;
            }
        } else if (node instanceof EmptyElement) {
            empty = (EmptyElement) node;

            if ("time".equals(empty.tagName)) {

                try {
                    addTimePoint(LocalTime.parse(empty.get(VALUE_TAG)));
                } catch (IllegalArgumentException e) {
                    LOG.log(Level.WARNING, "Invalid time point value ''{0}'' in <time>",
                        empty.get(VALUE_TAG));
                    result = false;

                    break;
                }
            } else {
                LOG.log(Level.WARNING, "Invalid element ''{0}'' within <time-series>",
                    empty.tagName);
                result = false;

                break;
            }
        }
    }

    return result;
}

/**
 * Processes a "subsequence" node by verifying that all contained nodes are <time> nodes, and
 * parsing them as they are found.
 *
 * @param   subseq  the subsequence node to parse
 * @return  <code>true</code> if the node is valid; <code>false</code> if not
 */
private boolean processSubsequence(final NonemptyElement subseq) {

    String name;
    EmptyElement empty;
    boolean result;

    name = subseq.get("name");

    if (name == null) {
        LOG.warning("Missing name on <subsequence>");
        result = false;
```

```
        } else {
            createSubsequence(name);
            result = true;

            for (Node node : subseq.children) {

                if (node instanceof NonemptyElement) {
                    LOG.log(Level.WARNING, "Invalid element ''{0}'' within <subsequence>",
                        ((NonemptyElement) node).tagName);
                    result = false;

                    break;
                } else if (node instanceof EmptyElement) {
                    empty = (EmptyElement) node;

                    if ("time".equals(empty.tagName)) {

                        try {
                            this.addToSubsequence(name, LocalTime.parse(empty.get(VALUE_TAG)));
                        } catch (IllegalArgumentException e) {
                            LOG.log(Level.WARNING, "Invalid time point value ''{0}'' in <time>",
                                empty.get(VALUE_TAG));
                            result = false;

                            break;
                        }
                    } else {
                        LOG.log(Level.WARNING, "Invalid element ''{0}'' within <subsequence>",
                            empty.tagName);
                        result = false;

                        break;
                    }
                }
            }
        }

        return result;
    }

    /**
     * Creates a file that represents the object data.
     *
     * @param   dir  the directory where the pipe's files are stored
     * @return  the file
     */
    private File makeFile(final File dir) {

        return new File(dir, getKey() + "-TimeSeries.xml");
    }
}
```

```
package com.srbenoit.filter.items;

import java.awt.Point;
import java.awt.Rectangle;
import java.util.ArrayList;
import java.util.List;

/**
 * A class that stores data relating to a trajectory.
 */
public class Trajectory {

    /** the time point of each point in the trajectory */
    public final List<Integer> timePoints;

    /** the planes of each point in the trajectory */
    public final List<Integer> planes;

    /** the position, velocity, and ambient velocity of the point */
    public final List<ImagePoint> points;

    /** the relative motion (first point is always (0,0)) */
    public final List<Point> relative;

    /**
     * Constructs a new <code>Trajectory</code> with no points.
     */
    public Trajectory() {

        this.timePoints = new ArrayList<Integer>(10);
        this.planes = new ArrayList<Integer>(10);
        this.points = new ArrayList<ImagePoint>(10);
        this.relative = new ArrayList<Point>(10);
    }
```

```
/**
 * Gets the number of points in the trajectory.
 *
 * @return  the number of points
 */
public int numPoints() {

    return this.points.size();
}

/**
 * Gets the time point for a given index.
 *
 * @param   index  the index of the point to retrieve
 * @return  the time point
 */
public int getTimePoint(final int index) {

    return this.timePoints.get(index).intValue();
}

/**
 * Gets the Z plane for a given index.
 *
 * @param   index  the index of the point to retrieve
 * @return  the Z plane
 */
public int getPlane(final int index) {

    return this.planes.get(index).intValue();
}

/**
 * Gets the absolute position of the trajectory point for given index.
 *
 * @param   index  the index of the point to retrieve
 * @return  the point
 */
public ImagePoint getPoint(final int index) {

    return this.points.get(index);
}

/**
 * Gets the relative position, compensated for ambient drift. The first point is always (0, 0).
 *
 * @param   index  the index of the point to retrieve
 * @return  the relative point
 */
public Point getRelative(final int index) {

    return this.relative.get(index);
}

/**
 * Gets the sum of the x position and x velocity of the last point in the trajectory.
 *
 * @return  the X coordinate
 */
public int getCurrentX() {

    ImagePoint last;

    last = this.points.get(this.points.size() - 1);

    return last.getXPos() + last.getXVel();
}

/**
 * Gets the sum of the y position and y velocity of the last point in the trajectory.
 *
 * @return  the Y coordinate
 */
public int getCurrentY() {

    ImagePoint last;

    last = this.points.get(this.points.size() - 1);

    return last.getYPos() + last.getYVel();
}

/**
 * Gets the plane where the last image point was recorded.
 *
 * @return  the plane of the last image point
 */
```

```
    public int getCurrentPlane() {

        return this.planes.get(this.planes.size() - 1);
    }

    /**
     * Adds a new point to the trajectory, adjusting the thumbnail size as needed.
     *
     * @param  timePoint  the time point when the trajectory passed through the point
     * @param  plane      the Z plane of the trajectory when it passed through the point
     * @param  point      the point to add to the trajectory
     */
    public void addPoint(final int timePoint, final int plane, final ImagePoint point) {

        int distX;
        int distY;
        Point prior;

        if (this.points.isEmpty()) {
            this.relative.add(new Point(0, 0));
        } else {
            prior = this.relative.get(this.points.size() - 1);
            distX = point.getXVel() - point.getXAmbientVel();
            distY = point.getYVel() - point.getYAmbientVel();
            this.relative.add(new Point(prior.x + distX, prior.y + distY));
        }

        this.timePoints.add(Integer.valueOf(timePoint));
        this.planes.add(Integer.valueOf(plane));
        this.points.add(point);
    }

    /**
     * Gets the extends of the point's relative motion over the trajectory.
     *
     * @return  the extents
     */
    public Rectangle extents() {

        Rectangle rect;

        rect = new Rectangle();

        for (Point pt : this.relative) {
            rect.add(pt);
        }

        return rect;
    }
}

package com.srbenoit.filter.items;

import java.io.File;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.filter.AbstractPipeItem;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.PipeItemFileInfo;
import com.srbenoit.xml.EmptyElement;
import com.srbenoit.xml.Node;
import com.srbenoit.xml.NonemptyElement;
import com.srbenoit.xml.XmlParser;

/**
 * An ordered list of trajectories.
 */
public class TrajectoryListPipeItem extends AbstractPipeItem {

    /** end of line characters */
    private static final String CRLF;

    /** the list of trajectories */
    private final transient List<Trajectory> trajectories;

    static {
        String crlf;

        crlf = System.getProperty("line.separator");
        CRLF = (crlf == null) ? "\n" : crlf;
    }

    /**
     * Constructs a new <code>TrajectoryListPipeItem</code>.
```

```
     *
     * @param  theKey     the unique key for the item
     * @param  theLabel   the label for the item (a human friendly name)
     * @param  thePipe    the pipe in which this item is installed
     */
    public TrajectoryListPipeItem(final String theKey, final String theLabel, final Pipe thePipe) {

        super(theKey, theLabel, thePipe);

        PipeItemFileInfo info;

        this.trajectories = new ArrayList<Trajectory>(10);

        info = new PipeItemFileInfo(makeFile(getSubdir()));
        addFile(info);
    }

    /**
     * Adds a trajectory to the end of the list.
     *
     * @param  traj  the trajectory to add
     */
    public void addTrajectory(final Trajectory traj) {

        this.trajectories.add(traj);
        getFile(0).notPersisted();
    }

    /**
     * Gets the number of trajectories in the series
     *
     * @return  the number of trajectories
     */
    public int getNumTrajectories() {

        return this.trajectories.size();
    }

    /**
     * Get a particular trajectory from the series.
     *
     * @param   index  the index of the trajectory to get
     * @return  the trajectory
     */
    public Trajectory getTrajectory(final int index) {

        return this.trajectories.get(index);
    }

    /**
     * Gets a human-friendly name for the data type. For example, a list of sets of images
     * representing a time series of z-planes might return "Multi-plane image sequence".
     *
     * @return  the name of the data type this item represents
     */
    @Override public String typeName() {

        return "Trajectory List";
    }

    /**
     * Resets the pipe item to a virgin (empty) state.
     */
    @Override public void reset() {

        this.trajectories.clear();
        getFile(0).notPersisted();
    }

    /**
     * Saves the item to a filesystem.
     *
     * @param   executor  the executor that is saving the pipe
     * @param   startPct  the starting progress percentage for the save operation
     * @param   endPct    the ending progress percentage for the save operation
     * @return  <code>true</code> on successful save; <code>false</code> on failure
     */
    @Override public boolean save(final FilterTreeExecutor executor, final int startPct,
        final int endPct) {

        PipeItemFileInfo info;
        StringBuilder str;
        ImagePoint point;
        boolean result;

        executor.indicateProgress((startPct + endPct) / 2);
```

```
    info = getFile(0);

    if (this.trajectories.isEmpty()) {
        info.notPersisted();
        result = false;
    } else {
        str = new StringBuilder(500);

        str.append("<trajectory-list>");
        str.append(CRLF);

        // Write out the ordered list of time points
        for (Trajectory traj : this.trajectories) {
            str.append("  <trajectory>");
            str.append(CRLF);

            for (int i = 0; i < traj.numPoints(); i++) {
                str.append("    <point_time='");
                str.append(traj.getTimePoint(i));
                str.append("' plane='");
                str.append(traj.getPlane(i));

                point = traj.getPoint(i);
                str.append("' x='");
                str.append(Integer.toString(point.getXPos()));
                str.append("' y='");
                str.append(Integer.toString(point.getYPos()));
                str.append("' x-vel='");
                str.append(Integer.toString(point.getXVel()));
                str.append("' y-vel='");
                str.append(Integer.toString(point.getYVel()));
                str.append("' x-ambient-vel='");
                str.append(Integer.toString(point.getXAmbientVel()));
                str.append("' y-ambient-vel='");
                str.append(Integer.toString(point.getYAmbientVel()));

                str.append("'/>");
                str.append(CRLF);
            }

            str.append("  </trajectory>");
            str.append(CRLF);
        }

        str.append("</trajectory-list>");
        str.append(CRLF);

        if (Pipe.writeFile(info.getFile(), str.toString().getBytes())) {
            info.wasPersisted();
            result = true;
        } else {
            info.notPersisted();
            result = false;
        }
    }

    return result;
}

/**
 * Loads the items from the filesystem.
 *
 * @return  <code>true</code> if the load succeeded; <code>false</code> if not
 */
@Override public boolean load() {

    PipeItemFileInfo info;
    byte[] data;
    XmlParser parser;
    List<Node> parsed;
    NonemptyElement top;
    boolean result;

    info = getFile(0);

    reset();

    data = Pipe.readFile(info.getFile());

    if (data == null) {
        info.notPersisted();
        result = false;
    } else {
        parser = new XmlParser();

        try {
            parsed = parser.parse(new String(data), true);
```

```
            // Should be one node at top level, "time-series"
            if ((parsed != null) && (parsed.size() == 1)
                    && (parsed.get(0) instanceof NonemptyElement)) {

                top = (NonemptyElement) parsed.get(0);

                if ("trajectory-list".equals(top.tagName)) {
                    result = processTopLevel(top);
                } else {
                    LOG.warning("Missing <trajectory-list> element");
                    result = false;
                }
            } else {
                LOG.warning("Missing <trajectory-list> element");
                result = false;
            }

        } catch (ParseException e) {
            LOG.log(Level.WARNING, "Exception while parsing XML", e);
            result = false;
        }

        if (result) {
            info.wasPersisted();
        } else {
            info.notPersisted();
        }
    }

    return result;
}

/**
 * Processes the top-level "trajectory-list" node by verifying that all contained nodes are
 * either <trajectory> nodes, and parsing them as they are found.
 *
 * @param   top  the top-level node to parse
 * @return  <code>true</code> if the node is valid; <code>false</code> if not
 */
private boolean processTopLevel(final NonemptyElement top) {

    NonemptyElement nonempty;
    boolean result;

    result = true;

    for (Node node : top.children) {

        if (node instanceof NonemptyElement) {
            nonempty = (NonemptyElement) node;

            if ("trajectory".equals(nonempty.tagName)) {
                result = processTrajectory(nonempty);
            } else {
                LOG.log(Level.WARNING, "Invalid element ''{0}'' within <trajectory-list>",
                    nonempty.tagName);
                result = false;

                break;
            }
        } else if (node instanceof EmptyElement) {
            LOG.log(Level.WARNING, "Invalid element ''{0}'' within <trajectory-list>",
                ((EmptyElement) node).tagName);
            result = false;

            break;
        }
    }

    return result;
}

/**
 * Processes a "trajectory" node by verifying that all contained nodes are <point> nodes, and
 * parsing them as they are found.
 *
 * @param   elem  the trajectory node to parse
 * @return  <code>true</code> if the node is valid; <code>false</code> if not
 */
private boolean processTrajectory(final NonemptyElement elem) {

    Trajectory traj;
    EmptyElement empty;
    ImagePoint point;
    boolean result;
```

```
            traj = new Trajectory();
            result = true;

            for (Node node : elem.children) {

                if (node instanceof NonemptyElement) {
                    LOG.log(Level.WARNING, "Invalid element ''{0}'' within <trajectory>",
                            ((NonemptyElement) node).tagName);
                    result = false;

                    break;
                } else if (node instanceof EmptyElement) {
                    empty = (EmptyElement) node;

                    if ("point".equals(empty.tagName)) {

                        try {
                            point = new ImagePoint(Integer.parseInt(empty.get("x")), // NOPMD SRB
                                    Integer.parseInt(empty.get("y")),
                                    Integer.parseInt(empty.get("x-vel")),
                                    Integer.parseInt(empty.get("y-vel")),
                                    Integer.parseInt(empty.get("x-ambient-vel")),
                                    Integer.parseInt(empty.get("y-ambient-vel")));
                            traj.addPoint(Integer.parseInt(empty.get("time")),
                                Integer.parseInt(empty.get("plane")), point);
                        } catch (NumberFormatException e) {
                            LOG.warning("Invalid value in <point>");
                            result = false;

                            break;
                        } catch (NullPointerException e) {
                            LOG.warning("Missing value in <point>");
                            result = false;

                            break;
                        }
                    } else {
                        LOG.log(Level.WARNING, "Invalid element ''{0}'' within <trajectory>",
                            empty.tagName);
                        result = false;

                        break;
                    }
                }
            }

            if (result) {
                this.trajectories.add(traj);
            }

            return result;
        }

        /**
         * Creates a file that represents the object data.
         *
         * @param   dir  the directory where the item's files are stored
         * @return  the file
         */
        private File makeFile(final File dir) {

            return new File(dir, getKey() + "_TimeSeries.xml");
        }
    }
```

E.5.4 Video Microscopy Analysis (com.srbenoit.microscopy/code¿)

This package provides a filter set to support analysis of sets of images gathered through video fluorescence microscopy. This package relies on the LOCI Bio-Formats libraries to read MetaMorph files. To obtain these, download "loci_tools.jar" from the LOCI web site downloads page.

```
package com.srbenoit.microscopy;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.beans.PropertyChangeEvent;
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JProgressBar;
import javax.swing.JSeparator;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterTree;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.FilterTreeExecutorListener;
import com.srbenoit.filter.FilterTreeIO;
import com.srbenoit.filter.FilterTreePanel;
import com.srbenoit.filter.GraphFileFilter;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.ui.UIUtilities;
import com.srbenoit.util.ResourceLoader;

/**
 * A class that supports creating a new filter tree (analysis), opening an existing filter tree,
 * editing the tree and saving changes, and executing an open tree.
 */
public class AnalysisApp extends LoggedObject implements ActionListener,
    FilterTreeExecutorListener, Runnable {

    /** a file filter for the graph tree file extension */
    private static final GraphFileFilter FILTER;

    /** the list of classes the user can choose from */
    private final transient List<Class<? extends AbstractFilter>> classes;

    /** the frame */
    private transient JFrame frame;

    /** the filter tree */
    private transient FilterTree tree;

    /** the panel */
    private transient FilterTreePanel panel;

    /** the file location of the currently open graph */
    private transient File file;

    /** the last directory opened */
    private transient File lastDir;

    /** class to serialize/deserialize filter trees */
    private final transient FilterTreeIO ser;

    /** the text above the progress bar */
    private transient JLabel progressText;

    /** the progress bar */
    private transient JProgressBar progress;

    /** the execute menu item (disabled while executing) */
    private transient JMenuItem execute;

    /** the abort menu item (enabled while executing) */
    private transient JMenuItem abort;

    /** the executor that is running the current process */
    private FilterTreeExecutor executor;

    /** icon for the play button in enabled state */
```

```
private ImageIcon playNormal;

/** icon for the play button in disabled state */
private ImageIcon playDisabled;

/** icon for the stop button in enabled state */
private ImageIcon stopNormal;

/** icon for the stop button in disabled state */
private ImageIcon stopDisabled;

/** play/execute button */
private JButton play;

/** stop/abort button */
private JButton stop;

static {
    FILTER = new GraphFileFilter();
}

/**
 * Constructs a new <code>AnalysisApp</code>.
 *
 * @param  filterClasses  the list of classes the user can choose from
 */
public AnalysisApp(final List<Class<? extends AbstractFilter>> filterClasses) {

    this.classes = filterClasses;
    this.file = null;
    this.lastDir = null;
    this.ser = new FilterTreeIO();

    this.tree = new FilterTree();
}

/**
 * Called in the AWT event dispatcher thread to construct the GUI.
 */
public void run() {

    AbstractFilter[] filters;
    Font font;
    JPanel content;
    JMenuBar bar;
    JMenu menu;
    JMenuItem item;
    JPanel prog;
    JPanel inner;
    BufferedImage img;

    buildFilterTree();

    this.frame = new JFrame("Generalized Video Microscopy Analysis");
    this.frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    content = new JPanel(new BorderLayout());
    this.frame.setContentPane(content);

    this.panel = new FilterTreePanel(classes, this.tree);
    this.panel.setBackground(Color.WHITE);

    filters = new AbstractFilter[this.classes.size()];

    for (int i = 0; i < filters.length; i++) {
        filters[i] = AbstractFilter.getInstance(this.classes.get(i));
    }

    content.add(this.panel, BorderLayout.CENTER);

    bar = new JMenuBar();
    this.frame.setJMenuBar(bar);
    menu = new JMenu("File");
    font = menu.getFont();
    font = font.deriveFont(12.0f);
    menu.setFont(font);
    bar.add(menu);
    item = new JMenuItem("New Process");
    item.setActionCommand("new");
    item.addActionListener(this);
    item.setFont(font);
    menu.add(item);
    item = new JMenuItem("Open Process...");
    item.setActionCommand("open");
    item.addActionListener(this);
    item.setFont(font);
    menu.add(item);
```

```
        item = new JMenuItem("Save Process");
        item.setActionCommand("save");
        item.addActionListener(this);
        item.setFont(font);
        menu.add(item);
        item = new JMenuItem("Save Process As...");
        item.setActionCommand("saveas");
        item.addActionListener(this);
        item.setFont(font);
        menu.add(item);
        menu.add(new JSeparator());
        item = new JMenuItem("Exit");
        item.setActionCommand("exit");
        item.addActionListener(this);
        item.setFont(font);
        menu.add(item);

        menu = new JMenu("Process");
        menu.setFont(font);
        bar.add(menu);
        item = new JMenuItem("Execute...");
        item.setActionCommand("exec");
        item.addActionListener(this);
        item.setFont(font);
        menu.add(item);
        this.execute = item;
        item = new JMenuItem("Abort");
        item.setActionCommand("abort");
        item.addActionListener(this);
        item.setFont(font);
        item.setEnabled(false);
        menu.add(item);
        this.abort = item;

        this.progressText = new JLabel(" ");
        this.progress = new JProgressBar(0, 100);

        prog = new JPanel(new BorderLayout());
        prog.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createEmptyBorder(1, 1, 1, 1), BorderFactory.createEtchedBorder()));

        inner = new JPanel(new BorderLayout(3, 3));
        inner.setBorder(BorderFactory.createEmptyBorder(2, 5, 5, 5));
        inner.add(this.progressText, BorderLayout.NORTH);
        inner.add(this.progress, BorderLayout.SOUTH);
        prog.add(inner, BorderLayout.CENTER);
        content.add(prog, BorderLayout.SOUTH);

        img = ResourceLoader.loadImage(AnalysisApp.class, "images/play-normal.png");
        this.playNormal = new ImageIcon(img);
        img = ResourceLoader.loadImage(AnalysisApp.class, "images/play-disabled.png");
        this.playDisabled = new ImageIcon(img);

        img = ResourceLoader.loadImage(AnalysisApp.class, "images/stop-normal.png");
        this.stopNormal = new ImageIcon(img);
        img = ResourceLoader.loadImage(AnalysisApp.class, "images/stop-disabled.png");
        this.stopDisabled = new ImageIcon(img);

        this.play = new JButton(this.playNormal);
        this.play.setActionCommand("exec");
        this.play.addActionListener(this);

        this.stop = new JButton(this.stopDisabled);
        this.stop.setActionCommand("abort");
        this.stop.addActionListener(this);
        this.stop.setEnabled(false);

        inner = new JPanel(new BorderLayout(3, 3));
        inner.add(this.play, BorderLayout.WEST);
        inner.add(this.stop, BorderLayout.EAST);
        prog.add(inner, BorderLayout.WEST);

        this.frame.pack();
        UIUtilities.positionFrame(this.frame, 0.1, 0.1);
        this.frame.setVisible(true);
    }

    /**
     * Builds the filter tree that the program will begin with. This should be stored in a file
     * system, but for now is hard coded.
     */
    private void buildFilterTree() {

        MetaMorphReaderFilter metamorph;
        IntensityAutoLevelerFilter leveler;
        ZPlaneMergerFilter merger;
        SmootherFilter smoother;
```

```
    MotionCompensationFilter stabilizer;
    MaximaFinderFilter maxima;
    TrajectoryFilter trajectory;
    DiffusionAnalysisFilter analysis;
    QuicktimeBuilderFilter quicktime1;
    QuicktimeBuilderFilter quicktime2;

    metamorph = new MetaMorphReaderFilter();
    leveler = new IntensityAutoLevelerFilter();
    merger = new ZPlaneMergerFilter();
    smoother = new SmootherFilter();
    stabilizer = new MotionCompensationFilter();
    quicktime1 = new QuicktimeBuilderFilter();
    maxima = new MaximaFinderFilter();
    trajectory = new TrajectoryFilter();
    analysis = new DiffusionAnalysisFilter();
    quicktime2 = new QuicktimeBuilderFilter();

    leveler.getInputFormat(0).setKey(metamorph.getOutputFormat(2).getKey());
    merger.getInputFormat(0).setKey(leveler.getOutputFormat(0).getKey());
    smoother.getInputFormat(0).setKey(merger.getOutputFormat(0).getKey());
    stabilizer.getInputFormat(0).setKey(smoother.getOutputFormat(0).getKey());
    quicktime1.getInputFormat(0).setKey(stabilizer.getOutputFormat(0).getKey());
    maxima.getInputFormat(0).setKey(stabilizer.getOutputFormat(0).getKey());
    trajectory.getInputFormat(0).setKey(stabilizer.getOutputFormat(0).getKey());
    trajectory.getInputFormat(1).setKey(maxima.getOutputFormat(1).getKey());
    analysis.getInputFormat(0).setKey(stabilizer.getOutputFormat(0).getKey());
    analysis.getInputFormat(1).setKey(trajectory.getOutputFormat(0).getKey());
    quicktime2.getInputFormat(0).setKey(trajectory.getOutputFormat(1).getKey());

    this.tree.addFilter(metamorph);
    this.tree.addFilter(leveler);
    this.tree.addFilter(merger);
    this.tree.addFilter(smoother);
    this.tree.addFilter(stabilizer);
    this.tree.addFilter(quicktime1);
    this.tree.addFilter(maxima);
    this.tree.addFilter(trajectory);
    this.tree.addFilter(analysis);
    this.tree.addFilter(quicktime2);
}

/**
 * Handles action events generated by the menu items.
 *
 * @param  evt  the action event
 */
public void actionPerformed(final ActionEvent evt) {

    String cmd;
    int option;

    cmd = evt.getActionCommand();

    if ("new".equals(cmd)) {

        if (this.panel.isDirty()) {
            option = promptForSave("New Filter Graph");

            if (option == JOptionPane.YES_OPTION) {
                doSave();
                this.panel.clear();
            } else if (option == JOptionPane.NO_OPTION) {
                this.panel.clear();
            }
        } else {
            this.panel.clear();
        }
    } else if ("open".equals(cmd)) {

        if (this.panel.isDirty()) {
            option = promptForSave("Open Filter Graph");

            if (option == JOptionPane.YES_OPTION) {
                doSave();
                doOpen();
            } else if (option == JOptionPane.NO_OPTION) {
                doOpen();
            }
        } else {
            doOpen();
        }
    } else if ("save".equals(cmd)) {

        if (this.panel.isDirty()) {
            doSave();
        }
```

```
    } else if ("saveas".equals(cmd)) {

        doSaveAs();
    } else if ("exec".equals(cmd)) {
        doExecute();
    } else if ("abort".equals(cmd)) {
        doAbort();
    } else if ("exit".equals(cmd)) {

        if (this.panel.isDirty()) {
            option = promptForSave("Exit");

            if (option == JOptionPane.YES_OPTION) {
                doSave();
                doExit();
            } else if (option == JOptionPane.NO_OPTION) {
                doExit();
            }
        } else {
            doExit();
        }
    }
}

/**
 * Prompts the user for a file to load, then attempts to parse that file into a filter tree. If
 * successful, the panel is configured with the filter tree.
 */
private void doOpen() {

    JFileChooser chooser;
    int result;
    File selected;
    String content;
    String[] msg;

    chooser = new JFileChooser();

    if (this.lastDir != null) {
        chooser.setCurrentDirectory(this.lastDir);
    }

    chooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    chooser.setAcceptAllFileFilterUsed(true);
    chooser.setFileFilter(FILTER);
    result = chooser.showOpenDialog(this.panel);

    if (result == JFileChooser.APPROVE_OPTION) {
        selected = chooser.getSelectedFile();
        this.lastDir = selected.getParentFile();

        // Try to load the file
        content = ResourceLoader.loadFile(selected);

        try {
            this.tree = this.ser.deserialize(content, this.classes);
            this.panel.setTree(this.tree);
            this.file = selected;
            this.panel.repaint();
        } catch (Exception e) {
            msg = new String[] { "Unable to load this filter tree", e.getMessage() };
            JOptionPane.showMessageDialog(this.panel, msg, "Open Filter Tree",
                JOptionPane.ERROR_MESSAGE);
        }
    }
}

/**
 * If there is a file loaded, saves the current filter tree to that file. If there is no file
 * loaded (the current filter tree has not been saved yet), this method acts as if <code>
 * doSaveAs</code> had been called.
 */
private void doSave() {

    String content;
    FileOutputStream out;
    String[] msg;

    if (this.file == null) {
        doSaveAs();
    } else {
        content = this.ser.serialize(this.tree);

        try {
            out = new FileOutputStream(this.file);
            out.write(content.getBytes());
            out.close();
```

```
            this.panel.setDirty(false);
        } catch (Exception e) {
            msg = new String[] { "Unable to save this filter tree", e.getMessage() };
            JOptionPane.showMessageDialog(this.panel, msg, "Save Filter Tree",
                JOptionPane.ERROR_MESSAGE);
        }
    }
}

/**
 * Prompts the user for a file in which to save the current filter graph.
 */
private void doSaveAs() {

    JFileChooser chooser;
    int result;
    File selected;
    String content;
    FileOutputStream out;
    String[] msg;
    boolean approved;

    chooser = new JFileChooser();

    if (this.lastDir != null) {
        chooser.setCurrentDirectory(this.lastDir);
    }

    chooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    chooser.setAcceptAllFileFilterUsed(true);
    chooser.setFileFilter(FILTER);
    result = chooser.showSaveDialog(this.panel);

    if (result == JFileChooser.APPROVE_OPTION) {
        selected = chooser.getSelectedFile();
        this.lastDir = selected.getParentFile();

        // Add the extension ".ftree" if no extension was given
        if (!selected.getName().contains(".")) {
            selected = new File(this.lastDir, selected.getName() + ".ftree");
        }

        if (selected.exists()) {
            approved = (JOptionPane.showConfirmDialog(this.panel, "Overwrite existing file?",
                    "Save Filte Tree", JOptionPane.OK_CANCEL_OPTION)
                == JOptionPane.OK_OPTION);
        } else {
            approved = true;
        }

        if (approved) {
            content = this.ser.serialize(this.tree);

            try {
                out = new FileOutputStream(selected);
                out.write(content.getBytes());
                out.close();
                this.panel.setDirty(false);
                this.file = selected;
            } catch (Exception e) {
                msg = new String[] { "Unable to save this filter tree", e.getMessage() };
                JOptionPane.showMessageDialog(this.panel, msg, "Save Filter Tree",
                    JOptionPane.ERROR_MESSAGE);
            }
        }
    }
}

/**
 * Exits the application (takes no action to preserve unsaved data).
 */
private void doExit() {

    this.frame.setVisible(false);
    this.frame.dispose();
}

/**
 * Executes the current process.
 */
private void doExecute() {

    this.panel.setEnabled(false);

    this.execute.setEnabled(false);
    this.play.setEnabled(false);
    this.play.setIcon(this.playDisabled);
```

```
        this.abort.setEnabled(true);
        this.stop.setEnabled(true);
        this.stop.setIcon(this.stopNormal);

        this.progressText.setText("Initializing");

        this.executor = new FilterTreeExecutor(this.tree, this.panel);
        this.executor.addListener(this);
        this.executor.execute();
    }

    /**
     * Aborts the current process.
     */
    private void doAbort() {

        if (this.executor != null) {
            this.executor.cancel(false);
        }
    }

    /**
     * Called on the AWT event dispatcher thread after the filter tree has been completely
     * executed.
     */
    public void done() {

        if (this.executor != null) {

            if (this.executor.isCancelled()) {
                this.progressText.setText("Canceled");
            } else {
                this.progressText.setText("Finished");
            }

            this.progress.setValue(0);
            this.executor = null;
        }

        this.execute.setEnabled(true);
        this.play.setEnabled(true);
        this.play.setIcon(this.playNormal);

        this.abort.setEnabled(false);
        this.stop.setEnabled(false);
        this.stop.setIcon(this.stopDisabled);

        this.panel.setEnabled(true);
    }

    /**
     * Called on the AWT event dispatcher thread to provide notification of a progress update.
     *
     * @param  filter  the filter being executed
     * @param  index   the 0-based index of the filter being executed
     * @param  total   the total number of filters in the tree
     */
    public void process(final AbstractFilter filter, final int index, final int total) {

        // No action
    }

    /**
     * This method gets called when a bound property is changed.
     *
     * @param  evt  A PropertyChangeEvent object describing the event source and the property that
     *              has changed.
     */

    public void propertyChange(final PropertyChangeEvent evt) {

        String name;
        Object value;

        name = evt.getPropertyName();

        if ("progress".equals(name)) {
            value = evt.getNewValue();

            if (value instanceof Integer) {
                this.progress.setValue(((Integer) value).intValue());
            } else {
                LOG.log(Level.WARNING, "Invalid object type in progress update: {0}",
                    evt.getNewValue().getClass().getName());
            }
        } else if ("error".equals(name)) {
```

```
            JOptionPane.showMessageDialog(null, evt.getNewValue(), "Filter Error",
                    JOptionPane.WARNING_MESSAGE);
        } else if ("filter".equals(name)) {
            this.progressText.setText(evt.getNewValue().toString());
        }
    }

    /**
     * Displays a prompt to the user asking whether the current filter graph should be saved before
     * its data is flushed.
     *
     * @param   title  the title of the prompt dialog
     * @return  the user's choice - one of <code>JOptionPane.YES_OPTION</code>, <code>
     *          JOptionPane.NO_OPTION</code>, r <code>JOptionPane.CANCEL_OPTION</code>
     */
    private int promptForSave(final String title) {

        return JOptionPane.showConfirmDialog(this.panel, "Save the current filter graph?", title,
                JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.WARNING_MESSAGE);
    }

    /**
     * Main method to display the frame.
     *
     * @param  args  command-line arguments (ignored)
     */
    public static void main(final String... args) {

        List<Class<? extends AbstractFilter>> classes;

        classes = new ArrayList<Class<? extends AbstractFilter>>(10);

        classes.add(MetaMorphReaderFilter.class);
        classes.add(IntensityAutoLevelerFilter.class);
        classes.add(ZPlaneMergerFilter.class);
        classes.add(SmootherFilter.class);
        classes.add(MotionCompensationFilter.class);
        classes.add(MaximaFinderFilter.class);
        classes.add(TrajectoryFilter.class);
        classes.add(DiffusionAnalysisFilter.class);
        classes.add(QuicktimeBuilderFilter.class);

        SwingUtilities.invokeLater(new AnalysisApp(classes));
    }
}
```

```
package com.srbenoit.microscopy;

/**
 * An enumeration of the data formats supported by the analysis package.
 */
public enum DataFormat {

    /** a raw sequence of multi-layer TIF files generated by MetaMorph */
    METAMORPH_TIF_SEQUENCE,

    /** a sequence of single-image TIF files */
    TIF_SEQUENCE,

    /** a sequence of single-image PNG files */
    PNG_SEQUENCE,

    /** a QuickTime video */
    MOV_VIDEO,

    /** a cell position spreadsheet (comma-separated values) */
    CELL_POS_CSV,

    /** a cell and tissue vector spreadsheet (comma-separated values) */
    TISSUE_POS_VEC_CSV,

    /** cell trajectory spreadsheet (comma-separated values) */
    TRAJECTORIES_CSV,

    /** graphs illustrating the trajectory analyses */
    TRAJECTORY_GRAPHS;
}
```

```
package com.srbenoit.microscopy;

import java.awt.AlphaComposite;
import java.awt.Color;
import java.awt.Composite;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.geom.Path2D;
import java.awt.image.BufferedImage;
```

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import com.srbenoit.color.Gradient;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;
import com.srbenoit.filter.items.ImagePoint;
import com.srbenoit.filter.items.Trajectory;
import com.srbenoit.filter.items.TrajectoryListPipeItem;
import com.srbenoit.math.delaunay.GraphEdge;
import com.srbenoit.math.delaunay.Vertex;
import com.srbenoit.math.delaunay.Voronoi;
import com.srbenoit.math.grapher.Grapher;
import com.srbenoit.math.grapher.PointListGraph;
import com.srbenoit.math.optimizers.FailedToConvergeException;
import com.srbenoit.math.optimizers.Optimizable;
import com.srbenoit.math.optimizers.Optimizer;
import com.srbenoit.math.optimizers.OutputValue;

/**
 * A filter that performs a diffusion analysis on a set of trajectories.
 */
public class DiffusionAnalysisFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = 9105886485480586656L;

    /** PNG file format */
    private static final String PNG = "PNG";

    /** the set of trajectories loaded */
    private final transient List<float[][]> paths;

    /** site data corresponding to each trajectory */
    private final transient List<double[]> siteData;

    /** an offscreen image in which to draw the large graph */
    private final transient Grapher large;

    /** an offscreen image in which to draw the small graph */
    private final transient Grapher small;

    /** a gradient over hue with 100 steps */
    private final Gradient gradient;

    /**
     * Constructs a new <code>DiffusionAnalysisFilter</code>.
     */
    public DiffusionAnalysisFilter() {

        super("Diffusion Analysis", DiffusionAnalysisFilter.class.getName());

        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Motion-compensated images"));
        this.inputs.add(new FilterInput(TrajectoryListPipeItem.class, "Trajectories to analyze"));
        makeRenderer();

        this.paths = new ArrayList<float[][]>(50);
        this.siteData = new ArrayList<double[]>(100);
        this.large = new Grapher(800, 600);
        this.small = new Grapher(400, 300);

        this.gradient = new Gradient(360, 1);
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new DiffusionAnalysisFilter();
    }

    /**
```

```
     * Performs the filter operation.
     *
     * @param    executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param    pipe      a pipe containing the input data items
     * @throws   FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ImageArrayPipeItem images;
        TrajectoryListPipeItem trajectories;

        validateInputs(pipe);
        executor.indicateProgress(1);

        images = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());
        trajectories = (TrajectoryListPipeItem) pipe.get(this.inputs.get(1).getKey());
        executor.indicateProgress(2);

        runFilter(executor, pipe, images, trajectories);

        executor.indicateProgress(100);
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param  executor      the <code>FilterTreeExecutor</code> that is executing the filter
     * @param  dir           the directory where filter output files should be written
     * @param  images        the set of images with maxima marked
     * @param  trajectories  the set of trajectories to process
     */
    private void runFilter(final FilterTreeExecutor executor, final Pipe pipe,
        final ImageArrayPipeItem images, final TrajectoryListPipeItem trajectories) {

        File dir;

        dir = new File(pipe.getDir(), "analysis");

        if (!dir.exists()) {
            dir.mkdir();
        }

        try {
            extractTrajectories(trajectories);
            executor.indicateProgress(30);

            if (!executor.isCancelled()) {
                genReport(dir);
                executor.indicateProgress(50);
                genImages(images, dir);
                executor.indicateProgress(90);
            }
        } catch (IOException e) {
            LOG.log(Level.WARNING, "Failure in analysis", e);
        }
    }

    /**
     * Performs the analysis and generates a report of findings.
     *
     * @param   dir  the directory where the report is to be saved
     * @throws  IOException  if there is an error reading trajectories or writing the report
     */
    private void genReport(final File dir) throws IOException {

        File report;
        File summary;
        BufferedWriter outRep;
        BufferedWriter outSum;
        Optimizer opt;
        double[] scale;
        double[] guess;
        double[] result;
        double[] site;
        double avg;
        float sse;
        float sst;
        float curve;
        float rSquared;
        float deltaA;
        float deltaB;
        float angle;
        int angleBucket;
        int track;
        float dist;
```

```
float speed;

report = new File(dir, "analysis_report.csv");
summary = new File(dir, "analysis_summary.csv");

report.getParentFile().mkdirs();
makeReadme(dir);

outRep = new BufferedWriter(new FileWriter(report));
outSum = new BufferedWriter(new FileWriter(summary));

try {
    outRep.write("Trajectory analysis" + Pipe.CRLF);
    outSum.write("Summary data" + Pipe.CRLF + Pipe.CRLF);
    outSum.write(
        "track, start frame, length, x(0), y(0), K, alpha, SSE, SST, r^2, angle, angle bucket, distance, av
        + Pipe.CRLF);

    // Each trajectory is an array of arrays of floats, each entry
    // containing: frame, x, y, <x>, <y>, x - <x>, y - <y>, d, <d>,
    // d - <d>, (d - <d>)^2, and <(d - <d>)^2>

    // Curve fit each trajectory to traj[i][11] = 4 K t^alpha
    // where K is the diffusion constant, and alpha is the diffusion
    // exponent. To do this, we minimize the following:
    // [ sum_{i=1}^n (4 K i^{2 alpha} - i^alpha traj[i][11]) ]^2 +
    // [ sum_{i=1}^n (4 K i^{2 alpha} (ln i)
    // - traj[i][11] i^alpha (ln i)) ]^2

    scale = new double[] { 0.1, 0.1 };
    guess = new double[] { 1, 1 };

    track = 1;

    for (float[][] traj : this.paths) {
        opt = new Optimizer(new OptimizableFxn(traj), scale, false); // NOPMD SRB

        try {
            result = opt.optimize(guess, Level.WARNING);

            avg = 0;

            for (int i = 0; i < traj.length; i++) {
                avg += traj[i][10];
            }

            avg /= traj.length;

            sse = 0;
            sst = 0;
            outRep.write(Pipe.CRLF + "Track ");
            outRep.write(Integer.toString(track));
            outRep.write(Pipe.CRLF
                + "t, frame, x, y, <x>, <y>, x - <x>, y - <y>, d, <d>, d - <d>, (d - <d>)^2, <(d - <d>)^2>,
                + Pipe.CRLF);

            for (int i = 0; i < traj.length; i++) {
                curve = (float) (4 * result[0] * Math.pow(i, result[1]));
                outRep.write("  " + Integer.toString(i) + ", " + Float.toString(traj[i][0])
                    + ", " + Float.toString(traj[i][1]) + ", " + Float.toString(traj[i][2])
                    + ", " + Float.toString(traj[i][3]) + ", " + Float.toString(traj[i][4])
                    + ", " + Float.toString(traj[i][5]) + ", " + Float.toString(traj[i][6])
                    + ", " + Float.toString(traj[i][7]) + ", " + Float.toString(traj[i][8])
                    + ", " + Float.toString(traj[i][9]) + ", "
                    + Float.toString(traj[i][10]) + ", " + Float.toString(traj[i][11])
                    + ", " + Float.toString(curve) + Pipe.CRLF);
                sse += (curve - traj[i][11]) * (curve - traj[i][11]);
                sst += (avg - traj[i][11]) * (avg - traj[i][11]);
            }

            rSquared = 1 - (sse / sst);

            deltaA = traj[traj.length - 1][1] - traj[0][1];
            deltaB = traj[traj.length - 1][2] - traj[0][2];
            angle = (float) (180 * Math.atan2(deltaB, deltaA) / Math.PI);
            angleBucket = (int) Math.floor((angle + 12) / 24);
            dist = (float) Math.sqrt((deltaA * deltaA) + (deltaB * deltaB));
            speed = dist / traj.length;

            outRep.write(
                "start frame, length, x(0), y(0), K, alpha, SSE, SST, r^2, angle, angle bucket, distance, a
                + Pipe.CRLF);
            outRep.write(Float.toString(traj[0][0]) + ", " + traj.length + ", "
                + Float.toString(traj[0][1]) + ", " + Float.toString(traj[0][2]) + ", "
                + result[0] + ", " + result[1] + ", " + sse + ", " + sst + ", " + rSquared
                + ", " + angle + ", " + angleBucket + ", " + dist + ", " + speed
                + Pipe.CRLF);
```

```
                site = new double[13]; // NOPMD SRB
                site[0] = traj[0][0]; // Start frame
                site[1] = traj.length; // Length
                site[2] = traj[0][1]; // x(0)
                site[3] = traj[0][2]; // y(0)
                site[4] = result[0]; // K
                site[5] = result[1]; // Alpha
                site[6] = sse; // SSE
                site[7] = sst; // SST
                site[8] = rSquared; // R^2 for curve fit
                site[9] = angle; // angle of motion
                site[10] = angleBucket; // angle bucket of motion
                site[11] = dist; // distance moved
                site[12] = speed; // average speed
                this.siteData.add(site);

                outSum.write(Integer.toString(track) + ", " + Float.toString(traj[0][0]) + ", "
                    + traj.length + ", " + Float.toString(traj[0][1]) + ", "
                    + Float.toString(traj[0][2]) + ", " + result[0] + ", " + result[1] + ", "
                    + sse + ", " + sst + ", " + rSquared + ", " + angle + ", " + angleBucket
                    + ", " + dist + ", " + speed + Pipe.CRLF);

                drawGraphs(dir, traj, result, track);
            } catch (FailedToConvergeException e) {
                LOG.warning("Optimizer failed to converge");
            }

            track++;
        }

    } finally {
        outRep.close();
        outSum.close();
    }
}

/**
 * Extracts trajectories into arrays of values.
 *
 * @param   trajectories  the trajectories to extract
 * @throws  IOException  if there is an error reading trajectories
 */
private void extractTrajectories(final TrajectoryListPipeItem trajectories)
    throws IOException {

    Trajectory trajectory;
    ImagePoint imgPoint;
    float[][] traj;
    float totalX;
    float totalY;
    float totalD;
    float total;

    for (int i = 0; i < trajectories.getNumTrajectories(); i++) {

        trajectory = trajectories.getTrajectory(i);

        traj = new float[trajectory.numPoints()][12]; // NOPMD SRB

        for (int j = 0; j < trajectory.numPoints(); j++) {
            imgPoint = trajectory.getPoint(j);

            traj[j][0] = trajectory.getTimePoint(j);
            traj[j][1] = imgPoint.getXPos();
            traj[j][2] = -imgPoint.getYPos(); // Change sign

            totalX = 0;
            totalY = 0;

            for (int k = 0; k <= j; k++) {
                totalX += traj[k][1];
                totalY += traj[k][2];
            }

            traj[j][3] = totalX / (j + 1); // [3] is <x> so far
            traj[j][4] = totalY / (j + 1); // [4] is <y> so far

            traj[j][5] = traj[j][1] - traj[j][3]; // [5] is x-<x>
            traj[j][6] = traj[j][2] - traj[j][4]; // [6] is y-<y>
            traj[j][7] = (float) Math.sqrt(((traj[j][1] - traj[0][1])
                    * (traj[j][1] - traj[0][1]))
                + ((traj[j][2] - traj[0][2]) * (traj[j][2] - traj[0][2]))); // [7] is d

            totalD = 0;

            for (int k = 0; k <= j; k++) {
                totalD += traj[k][7];
```

```
                }

                traj[j][8] = totalD / (j + 1); // [8] is <d> so far
                traj[j][9] = traj[j][7] - traj[j][8]; // [9] is d-<d>
                traj[j][10] = traj[j][9] * traj[j][9]; // [10] is (d-<d>)^2

                total = 0;

                for (int k = 0; k <= j; k++) {
                    total += traj[k][10];
                }

                traj[j][11] = total / (j + 1); // [11] is <(d - <d>)^2>
            }

            this.paths.add(traj);
        }
    }

    /**
     * Generates images based on colorings of Voronoi cells that correspond to cell movement
     * parameters.
     *
     * @param   images  the set of images with maxima marked
     * @param   dir     the directory where images are to be saved
     * @throws  IOException  if there is an error reading trajectories or writing the report
     */
    private void genImages(final ImageArrayPipeItem images, final File dir) throws IOException {

        BufferedImage base;

        // Load the base image that we will use for overlays
        base = images.getImage(0, 0);

        if (base != null) {
            makeImages(base, dir);
        }
    }

    /**
     * Builds a series of overlay images.
     *
     * @param   base  the base image on which to overlay
     * @param   dir   the directory where images are to be saved
     * @throws  IOException  if there is an error writing an image file
     */
    private void makeImages(final BufferedImage base, final File dir) throws IOException {

        Vertex[] vertices;
        double[] site;
        Voronoi vor;
        List<GraphEdge> edges;
        BufferedImage overlay;
        Graphics2D grx;
        int index;
        Color color;
        Composite orig;
        double min;
        double max;
        Map<Vertex, Path2D> map;
        double sat;

        // Build the vertex list based on the initial points of all
        // trajectories, with Y coordinate negated
        vertices = new Vertex[this.siteData.size()];

        for (int i = 0; i < this.siteData.size(); i++) {
            site = this.siteData.get(i);
            vertices[i] = new Vertex(site[2], -site[3]); // NOPMD SRB
        }

        vor = new Voronoi(1);
        edges = vor.generateVoronoi(vertices);
        map = vor.makeVoronoiPolygons(vertices, edges);

        overlay = new BufferedImage(base.getWidth(), base.getHeight(), BufferedImage.TYPE_INT_RGB);
        grx = (Graphics2D) overlay.getGraphics();
        grx.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

        // Image 1: Just the Voronoi cell boundaries
        grx.drawImage(base, 0, 0, null);
        grx.setColor(Color.YELLOW);

        for (GraphEdge edge : edges) {
            grx.drawLine((int) edge.xPos1, (int) edge.yPos1, (int) edge.xPos2, (int) edge.yPos2);
        }
```

```
grx.setColor(Color.RED);

for (Vertex vert : vertices) {
    grx.fillOval((int) vert.xPos - 1, (int) vert.yPos - 1, 3, 3);
}

ImageIO.write(overlay, PNG, new File(dir, "Voronoi_boundaries.png"));

//
//
//

// Image 2: Color hue based on angle, saturation based on K
min = this.siteData.get(0)[4];
max = min;

for (int i = 1; i < this.siteData.size(); i++) {
    site = this.siteData.get(i);

    if (site[4] < min) {
        min = site[4];
    }

    if (site[4] > max) {
        max = site[4];
    }
}

grx.drawImage(base, 0, 0, null);
orig = grx.getComposite();
grx.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f));

for (Vertex vert : map.keySet()) {

    for (int i = 0; i < this.siteData.size(); i++) {
        site = this.siteData.get(i);

        if ((vert.xPos == site[2]) && (vert.yPos == -site[3])) {

            // Get hue based on angle
            index = (int) site[9];

            while (index < 0) {
                index += 360;
            }

            while (index >= 360) {
                index -= 360;
            }

            color = this.gradient.getColor(index);

            // Adjust saturation based on alpha
            sat = (site[4] - min) / (max - min);

            color = new Color((int) (color.getRed() * sat), // NOPMD SRB
                    (int) (color.getGreen() * sat), (int) (color.getBlue() * sat));

            grx.setColor(color);
            grx.fill(map.get(vert));

            break;
        }
    }
}

grx.setComposite(orig);
grx.setColor(Color.RED);

for (Vertex vert : vertices) {
    grx.fillOval((int) vert.xPos - 1, (int) vert.yPos - 1, 3, 3);
}

drawColorWheel(grx);
ImageIO.write(overlay, PNG, new File(dir, "Voronoi_k.png"));

//
//
//

// Image 3: Color hue based on angle, saturation based on ALPHA
min = this.siteData.get(0)[5];
max = min;

for (int i = 1; i < this.siteData.size(); i++) {
    site = this.siteData.get(i);
```

```
        if (site[5] < min) {
            min = site[5];
        }

        if (site[5] > max) {
            max = site[5];
        }
    }

    grx.drawImage(base, 0, 0, null);
    orig = grx.getComposite();
    grx.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f));

    for (Vertex vert : map.keySet()) {

        for (int i = 0; i < this.siteData.size(); i++) {
            site = this.siteData.get(i);

            if ((vert.xPos == site[2]) && (vert.yPos == -site[3])) {

                // Get hue based on angle
                index = (int) site[9];

                while (index < 0) {
                    index += 360;
                }

                while (index >= 360) {
                    index -= 360;
                }

                color = this.gradient.getColor(index);

                // Adjust saturation based on alpha
                sat = (site[5] - min) / (max - min);

                color = new Color((int) (color.getRed() * sat), // NOPMD SRB
                        (int) (color.getGreen() * sat), (int) (color.getBlue() * sat));

                grx.setColor(color);
                grx.fill(map.get(vert));

                break;
            }
        }
    }

    grx.setComposite(orig);
    grx.setColor(Color.RED);

    for (Vertex vert : vertices) {
        grx.fillOval((int) vert.xPos - 1, (int) vert.yPos - 1, 3, 3);
    }

    drawColorWheel(grx);
    ImageIO.write(overlay, PNG, new File(dir, "Voronoi_alpha.png"));

    //
    //
    //

    // Image 4: Color hue based on angle, saturation based on dist
    min = this.siteData.get(0)[11];
    max = min;

    for (int i = 1; i < this.siteData.size(); i++) {
        site = this.siteData.get(i);

        if (site[11] < min) {
            min = site[11];
        }

        if (site[11] > max) {
            max = site[11];
        }
    }

    grx.drawImage(base, 0, 0, null);
    orig = grx.getComposite();
    grx.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f));

    for (Vertex vert : map.keySet()) {

        for (int i = 0; i < this.siteData.size(); i++) {
            site = this.siteData.get(i);

            if ((vert.xPos == site[2]) && (vert.yPos == -site[3])) {
```

```
                // Get hue based on angle
                index = (int) site[9];

                while (index < 0) {
                    index += 360;
                }

                while (index >= 360) {
                    index -= 360;
                }

                color = this.gradient.getColor(index);

                // Adjust saturation based on alpha
                sat = (site[11] - min) / (max - min);

                color = new Color((int) (color.getRed() * sat), // NOPMD SRB
                        (int) (color.getGreen() * sat), (int) (color.getBlue() * sat));

                grx.setColor(color);
                grx.fill(map.get(vert));

                break;
            }
        }
    }

    grx.setComposite(orig);
    grx.setColor(Color.RED);

    for (Vertex vert : vertices) {
        grx.fillOval((int) vert.xPos - 1, (int) vert.yPos - 1, 3, 3);
    }

    drawColorWheel(grx);
    ImageIO.write(overlay, PNG, new File(dir, "Voronoi_dist.png"));

    //
    //
    //

    // Image 5: Color hue based on angle, saturation based on speed
    min = this.siteData.get(0)[12];
    max = min;

    for (int i = 1; i < this.siteData.size(); i++) {
        site = this.siteData.get(i);

        if (site[12] < min) {
            min = site[12];
        }

        if (site[12] > max) {
            max = site[12];
        }
    }

    grx.drawImage(base, 0, 0, null);
    orig = grx.getComposite();
    grx.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.5f));

    for (Vertex vert : map.keySet()) {

        for (int i = 0; i < this.siteData.size(); i++) {
            site = this.siteData.get(i);

            if ((vert.xPos == site[2]) && (vert.yPos == -site[3])) {

                // Get hue based on angle
                index = (int) site[9];

                while (index < 0) {
                    index += 360;
                }

                while (index >= 360) {
                    index -= 360;
                }

                color = this.gradient.getColor(index);

                // Adjust saturation based on alpha
                sat = (site[12] - min) / (max - min);

                color = new Color((int) (color.getRed() * sat), // NOPMD SRB
                        (int) (color.getGreen() * sat), (int) (color.getBlue() * sat));
```

```
                    grx.setColor(color);
                    grx.fill(map.get(vert));

                    break;
                }
            }
        }

        grx.setComposite(orig);
        grx.setColor(Color.RED);

        for (Vertex vert : vertices) {
            grx.fillOval((int) vert.xPos - 1, (int) vert.yPos - 1, 3, 3);
        }

        drawColorWheel(grx);
        ImageIO.write(overlay, PNG, new File(dir, "Voronoi_speed.png"));

        // site[0] = traj[0][0]; // Start frame
        // site[1] = traj.length; // Length
        // site[2] = traj[0][1]; // x(0)
        // site[3] = traj[0][2]; // y(0)
        // site[4] = result[0]; // K
        // site[5] = result[1]; // Alpha
        // site[6] = sse; // SSE
        // site[7] = sst; // SST
        // site[8] = rSquared; // R^2 for curve fit
        // site[9] = angle; // angle of motion
        // site[10] = angleBucket; // angle bucket of motion
        // site[11] = dist; // distance moved
        // site[12] = speed; // average speed

    }

    /**
     * Draws the color wheel.
     *
     * @param  grx  the <code>Graphics</code> to which to draw
     */
    private void drawColorWheel(final Graphics2D grx) {

        double angle;
        double cos;
        double sin;
        Color color;

        for (int index = 0; index < 360; index++) {
            color = this.gradient.getColor(index);

            angle = index * Math.PI / 180;
            cos = Math.cos(angle);
            sin = Math.sin(angle);

            grx.setColor(color);
            grx.drawLine((int) (30 - (15 * cos)), (int) (30 - (15 * sin)), (int) (30 - (20 * cos)),
                (int) (30 - (20 * sin)));
        }
    }

    /**
     * Computes twice the area of the oriented triangle <code>(xPos1, yPos1), (xPos2, y3), (x3,
     * y3)</code> (the area is positive if the triangle is oriented counterclockwise).
     *
     * @param   xPos1  the first point X coordinate
     * @param   yPos1  the first point Y coordinate
     * @param   xPos2  the second point X coordinate
     * @param   yPos2  the second point Y coordinate
     * @param   xPos3  the third point X coordinate
     * @param   yPos3  the third point Y coordinate
     * @return  twice the oriented area
     */
    public double triArea(final double xPos1, final double yPos1, final double xPos2,
        final double yPos2, final double xPos3, final double yPos3) {

        return ((xPos2 - xPos1) * (yPos3 - yPos1)) - ((yPos2 - yPos1) * (xPos3 - xPos1));
    }

    /**
     * Tests whether the points of a triangle are in counterclockwise order.
     *
     * @param   xPos1  the first point X coordinate
     * @param   yPos1  the first point Y coordinate
     * @param   xPos2  the second point X coordinate
     * @param   yPos2  the second point Y coordinate
     * @param   xPos3  the third point X coordinate
     * @param   yPos3  the third point Y coordinate
```

```
     * @return  <code>true</code> if triangle <code>(xPos1, yPos1), (xPos2, y3), (x3, y3)</code> is
     *          in counterclockwise order
     */
    public boolean isCcw(final double xPos1, final double yPos1, final double xPos2,
        final double yPos2, final double xPos3, final double yPos3) {

        return triArea(xPos1, yPos1, xPos2, yPos2, xPos3, yPos3) > 0;
    }

    /**
     * Generates the "read me" file in the analysis directory.
     *
     * @param   dir  the analysis directory
     * @throws  IOException  if there is an error writing the file
     */
    private void makeReadme(final File dir) throws IOException {

        File readme;
        PrintStream out;

        readme = new File(dir, "analysis_readme.txt");

        out = new PrintStream(new FileOutputStream(readme));

        out.println("Summary of analysis files generated:");
        out.println("");
        out.println("analysis_report.csv:");
        out.println("    This comma-separated-value (CSV) file, which can be opened directly in");
        out.println("Excel or similar programs contains a complete analysis of the trajectories");
        out.println("extracted from the frames. The file is divided into sections, each labeled");
        out.println("'Track n' where n ranges from 1 to the number of trajectories. Within each");
        out.println("section are the following columns:");
        out.println("  t: the time since the trajectory's start");
        out.println("  frame: the frame number");
        out.println("  x: the X coordinate of the cell (corrected for tissue motion for t > 0)");
        out.println("  y: the Y coordinate of the cell (corrected for tissue motion for t > 0)");
        out.println("  a: the X coordinate, rotated through an angle");
        out.println("  b: the Y coordinate, rotated through an angle");
        out.println("  <a>: the mean of the 'a' coordinates so far");
        out.println("  <b>: the mean of the 'b' coordinates so far");
        out.println("  a - <a>: difference between <a> and a");
        out.println("  b - <b>: difference between <b> and b");
        out.println("  d - <d>: difference between distance from start (d) and mean distance");
        out.println("          from start (<d>) so far");
        out.println("  (d - <d>)^2: square of (d - <d>)");
        out.println("  <(d - <d>)^2>: mean squared differences of distance of the cell from");
        out.println("                 its starting point and mean distance from start");
        out.println("  Curve Fit: corresponding y value on best-fit curve of the form");
        out.println("             y = 2 d K t^alpha, where d is dimension (2)");
        out.println("             [Metzler R, Klafter J, \"The restaurant at the end of the");
        out.println("             random walk: Recent developments in the description of");
        out.println("             anomalous transport by fractional dynamics.\", 2004.");
        out.println("             J Phys A, 37(31): R161 R208]");
        out.println("At the bottom of each section is summary data:");
        out.println("  Start frame: the frame number where the trajectory began");
        out.println("  x(0): the X coordinate where the cell began");
        out.println("  y(0): the Y coordinate where the cell began");
        out.println("  K: the diffusion constant in the best-fit curve");
        out.println("  alpha: the diffusion exponent in the best-fit curve");
        out.println("  SSE: the sum of squared errors for the curve fit");
        out.println("  SST: the total variation of the curve fit");
        out.println("  r^2: the coefficient of determination of the curve fit");
        out.println("  angle: the angle of the trajectory (based on end points)");
        out.println("  angle_bucket: assignment of angle into a group of similar angles (*)");
        out.println("");
        out.println("analysis_summary.csv:");
        out.println("    This comma-separated-value (CSV) file, aggregates just the summary");
        out.println("lines that appear in the analysis_report.csv file.");
        out.println("and is provided as a convenience for generating spreadsheet charts.");
        out.println("");
        out.println("report.txt:");
        out.println("    As in all steps of the analysis process, this file records the amount");
        out.println("of time the processing step took, and its presence indicates that a given");
        out.println("phase of the analysis has been completed.  Deleting this file will cause");
        out.println("the alalysis program to re-run this phase of the analysis.");
        out.println("");
        out.println("Track_n_curvefit_lg.png");
        out.println(
            "    This series of large plots graph the <(d - <d>)^2> value (shown in blue)");
        out.println(
            "and the corresponding best-fit curve (shown in dark red) for each trajectory.");
        out.println("Labeling has intentionally been kept to a minimim to make it easier to add");
        out.println("custom labels to the graphs for inclusion in publications.");
        out.println("");
        out.println("Track_n_curvefit_sm.png");
        out.println("    These are simply smaller versions of the Track_n_curvefit_lg.png plots,");
        out.println(
```

```
            "but with the same font size, making them clearer when printed in small size.");
        out.println("");
        out.println("");
        out.println("");
        out.println("* Angles are measured from the positive X axis, with angle increasing");
        out.println("  counterclockwise. Angle bucket allows grouping of trajectories with");
        out.println("  others of similar direction.  Each bucket subtends 24 degrees of arc.");
        out.println("  Bucket 0 represents angles from -12 to +12 degrees.  Bucket +1 goes");
        out.println("  from +12 to +36, and bucket -1 goes from -12 to -36.  Therefore, the");
        out.println("  possible range is from -8 to +8.");
        out.println("");
        out.println("  A useful analysis is to group trajectories by some measure (distance,");
        out.println("  angle, speed, or start position), then generate a histogram of how");
        out.println("  many cells fell into each bucket, correlating this count with the");
        out.println("  selected measure, as a way to correlate directional bias with that");
        out.println("  measure.");

        out.close();
    }

    /**
     * Renders the graphs and writes them out to PNG files.
     *
     * @param  dir        the directory in which to export graph files
     * @param  traj       the list of trajectory points
     * @param  result     the result of the curve fit (K, alpha)
     * @param  trackNum   the track number (matches the CSV file)
     */
    private void drawGraphs(final File dir, final float[][] traj, final double[] result,
        final int trackNum) {

        double[] xCoords;
        double[] actual;
        double[] fit;
        PointListGraph actCurve;
        PointListGraph fitCurve;
        File file;

        xCoords = new double[traj.length];
        actual = new double[traj.length];
        fit = new double[traj.length];

        for (int i = 0; i < traj.length; i++) {
            xCoords[i] = traj[i][0] - traj[0][0];
            actual[i] = traj[i][11];
            fit[i] = (float) (4 * result[0] * Math.pow(i, result[1]));
        }

        actCurve = new PointListGraph(xCoords, actual);
        fitCurve = new PointListGraph(xCoords, fit);

        file = new File(dir, "Track-" + trackNum + "-curvefit-lg.png");
        this.large.graph(actCurve, fitCurve);

        try {
            this.large.export(file);
        } catch (IOException e) {
            LOG.log(Level.WARNING, "Failed to export large graph", e);
        }

        file = new File(dir, "Track-" + trackNum + "-curvefit-sm.png");
        this.small.graph(actCurve, fitCurve);

        try {
            this.small.export(file);
        } catch (IOException e) {
            LOG.log(Level.WARNING, "Failed to export small graph", e);
        }
    }

    /**
     * A function that supports optimization.
     */
    private static class OptimizableFxn implements Optimizable {

        /** the data to which to fit the curve */
        private final transient float[][] data;

        /**
         * Constructs a new <code>OptimizableFxn</code>.
         *
         * @param  theData  the data to which to fit the curve
         */
        private OptimizableFxn(final float[][] theData) {

            this.data = theData.clone();
        }
```

```
        /**
         * Gets the dimension of the function.
         *
         * @return  the dimension
         */
        public int dimension() {

            return 2;
        }

        /**
         * Evaluates the function for a given set of parameters that are to be optimized.
         *
         * @param   parameters  the parameters ([0] is K, [1] is alpha)
         * @return  the function value based on the parameters
         */
        public OutputValue evaluate(final double[] parameters) {

            OutputValue output;
            double term;

            output = new OutputValue();

            for (int i = 1; i < this.data.length; i++) {
                term = (4 * parameters[0] * Math.pow(i, parameters[1])) - this.data[i][11];
                output.setValue(output.getValue() + (term * term));
            }

            if (parameters[1] < 0) {
                output.setOutOfRange(true);
                output.setValue(Double.POSITIVE_INFINITY);
            }

            return output;
        }

        /**
         * Called when the optimizer has found a new set of parameters that result in a lower value
         * during its search for the optimal parameters.
         *
         * @param  parameters  the newly found parameters
         */
        public void updatedParameters(final double[] parameters) {

            // No action
        }
    }

    /**
     * Generates the string representation of the filter.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        return "TrajectoryFilter";
    }
}

package com.srbenoit.microscopy;

import java.awt.image.BufferedImage;
import java.awt.image.WritableRaster;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;

/**
 * A filter that takes an image array and adjusts the brightness and contrast to equalize intensity
 * values over the entire range.
 */
public class IntensityAutoLevelerFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = -3172221340075098136L;

    /**
     * Constructs a new <code>IntensityAutoLevelerFilter</code>.
     */
    public IntensityAutoLevelerFilter() {

        super("Intensity_auto-leveler", IntensityAutoLevelerFilter.class.getName());
```

```
        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Raw images"));
        this.outputs.add(new FilterOutput(ImageArrayPipeItem.class, "Intensity-leveled images",
                "leveled_images"));
        makeRenderer();
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new IntensityAutoLevelerFilter();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ImageArrayPipeItem input;
        ImageArrayPipeItem output;

        validateInputs(pipe);
        executor.indicateProgress(1);

        input = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());
        executor.indicateProgress(2);

        // Install a dummy image array to test for persisted data
        output = new ImageArrayPipeItem(this.outputs.get(0).getKey(),
                "Intensity-leveled images (PNG format)", pipe, "t", "z", input.getXSize(),
                input.getYSize(), "png");
        pipe.add(output);
        executor.indicateProgress(3);

        runFilter(executor, input, output);

        executor.indicateProgress(80);

        if (!executor.isCancelled()) {
            pipe.save(executor);
        }

        executor.indicateProgress(100);
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   input     the images to process
     * @param   output    the destination in which to store processed images
     */
    private void runFilter(final FilterTreeExecutor executor, final ImageArrayPipeItem input,
        final ImageArrayPipeItem output) {

        int width;
        int height;
        double[][] imageMeans;
        double totalInImage;
        double total;
        double totalMean;
        WritableRaster raster;
        int sample;
        int count;
        int totalCount;
        int min;
        int max;
        BufferedImage dest;
        WritableRaster newRaster;
        int pixel;
        double scale;

        width = input.getXSize();
        height = input.getYSize();

        // Scan each image to get mean intensity
        imageMeans = new double[width][height];
```

```
total = 0;
totalCount = 0;
min = Integer.MAX_VALUE;
max = 0;

for (int x = 0; x < width; x++) {

    for (int y = 0; y < height; y++) {

        if (executor.isCancelled()) {
            break;
        }

        executor.indicateProgress(4 + (40 * ((x * height) + y) / (width * height)));

        totalInImage = 0;
        count = 0;
        raster = input.getImage(x, y).getRaster();

        for (int xPix = 0; xPix < raster.getWidth(); xPix++) {

            for (int yPix = 0; yPix < raster.getHeight(); yPix++) {

                for (int b = 0; b < raster.getNumBands(); b++) {
                    sample = raster.getSample(xPix, yPix, b);

                    if (sample > max) {
                        max = sample;
                    }

                    if (sample < min) {
                        min = sample;
                    }

                    totalInImage += sample;
                    total += sample;
                    count++;
                    totalCount++;
                }
            }
        }

        imageMeans[x][y] = totalInImage / count;
    }
}

if (!executor.isCancelled()) {
    totalMean = total / totalCount;

    for (int x = 0; x < width; x++) {

        for (int y = 0; y < height; y++) {

            if (executor.isCancelled()) {
                break;
            }

            executor.indicateProgress(45 + (35 * ((x * height) + y) / (width * height)));

            // Determine a scale to map the entire intensity range into 0-255
            scale = 255.0 * (totalMean / imageMeans[x][y]) / (max - min);

            raster = input.getImage(x, y).getRaster();
            dest = new BufferedImage(raster.getWidth(), raster.getHeight(), // NOPMD SRB
                    BufferedImage.TYPE_INT_RGB);

            newRaster = dest.getRaster();

            if (raster.getNumBands() == 1) {

                for (int xPix = 0; xPix < raster.getWidth(); xPix++) {

                    for (int yPix = 0; yPix < raster.getHeight(); yPix++) {
                        sample = raster.getSample(xPix, yPix, 0);
                        pixel = (int) ((sample - min) * scale);
                        pixel = (int) ((sample - min) * scale);

                        if (pixel > 255) {
                            pixel = 255;
                        }

                        for (int b = 0; b < newRaster.getNumBands(); b++) {
                            newRaster.setSample(xPix, yPix, b, pixel);
                        }
                    }
                }
            } else if (raster.getNumBands() == 3) {
```

```
                        for (int xPix = 0; xPix < raster.getWidth(); xPix++) {

                            for (int yPix = 0; yPix < raster.getHeight(); yPix++) {

                                for (int b = 0; b < 3; b++) {
                                    sample = raster.getSample(xPix, yPix, b);
                                    pixel = (int) ((sample - min) * scale);

                                    if (pixel > 255) { // NOPMD SRB
                                        pixel = 255;
                                    }

                                    newRaster.setSample(xPix, yPix, b, pixel);
                                }
                            }
                        }

                        output.setImage(x, y, dest);
                    }
                }
            }
        }

        /**
         * Generates the string representation of the filter.
         *
         * @return  the string representation
         */
        @Override public String toString() {

            return "IntensityAutoLevelerFilter";
        }
}

package com.srbenoit.microscopy;

import java.awt.Graphics;
import java.awt.Point;
import java.awt.image.BufferedImage;
import java.awt.image.WritableRaster;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;
import com.srbenoit.filter.items.ImagePoint;
import com.srbenoit.filter.items.PointSetArrayPipeItem;

/**
 * A filter that identifies extrema (maxima or minima) in the input image sequence.
 */
public class MaximaFinderFilter extends AbstractFilter {

    /** version number for serialization */
    private final static long serialVersionUID = 2417170213367289037L;

    /** size of blocks for extrema detection */
    private static final int EXTREMA_SIZE = 14;

    /** maximum permitted motion between frames */
    private static final int MAX_FRAME_MOTION = 8;

    /**
     * Constructs a new <code>MaximaFinderFilter</code>.
     */
    public MaximaFinderFilter() {

        super("Intensity Maxima Identifier", MaximaFinderFilter.class.getName());

        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Source images"));
        this.outputs.add(new FilterOutput(ImageArrayPipeItem.class, "Images with maxima marked",
                "marked_images"));
        this.outputs.add(new FilterOutput(PointSetArrayPipeItem.class,
                "Points of maximal intensity", "maxima"));
        makeRenderer();
    }
```

```
/**
 * Duplicates the filter including all of its settings, but returns an independent object.
 *
 * @return  the duplicated object
 */
@Override public AbstractFilter duplicate() {

    return new MaximaFinderFilter();
}

/**
 * Performs the filter operation.
 *
 * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
 * @param   pipe      a pipe containing the input data items
 * @throws  FilterException  if the filter cannot complete
 */
@Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
    throws FilterException {

    ImageArrayPipeItem input;
    ImageArrayPipeItem output;
    PointSetArrayPipeItem maxima;

    validateInputs(pipe);
    executor.indicateProgress(1);

    input = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());

    // Install a dummy image array to test for persisted data
    output = new ImageArrayPipeItem(this.outputs.get(0).getKey(),
            "Images with maxima marked (PNG format)", pipe, "t", "z", input.getXSize(),
            input.getYSize(), "png");
    pipe.add(output);
    executor.indicateProgress(2);

    maxima = new PointSetArrayPipeItem(this.outputs.get(1).getKey(), "Intensity maxima", pipe,
            input.getXSize(), input.getYSize());
    pipe.add(maxima);
    executor.indicateProgress(3);

    runFilter(executor, pipe, input, output, maxima);

    executor.indicateProgress(80);

    if (!executor.isCancelled()) {
        pipe.save(executor);
    }

    executor.indicateProgress(100);
}

/**
 * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
 * the first dimension of which is time, and the second dimension of which is z plane.
 *
 * @param  executor  the <code>FilterTreeExecutor</code> that is executing the filter
 * @param  pipe      a pipe containing the input data items
 * @param  input     the images to process
 * @param  output    the image array in which to store output images
 * @param  maxima    the point set array in which to store maxima that were found
 */
private void runFilter(final FilterTreeExecutor executor, final Pipe pipe,
    final ImageArrayPipeItem input, final ImageArrayPipeItem output,
    final PointSetArrayPipeItem maxima) {

    int width;
    int height;
    BufferedImage orig;
    BufferedImage newImage;
    WritableRaster raster;
    List<Point> frameMaxima;
    int imgWidth;
    int imgHeight;
    Graphics grx;
    int xCoord;
    int yCoord;
    int[] pixel;
    boolean isMax;
    int maxDecrease;
    int deltaX;
    int deltaY;
    int[] test;
    File outfile;
    FileWriter writer;
    ImagePoint point;
```

```
        width = input.getXSize();
        height = input.getYSize();

        frameMaxima = new ArrayList<Point>(30);

        pixel = new int[3];
        test = new int[3];

        for (int x = 0; x < width; x++) {

            for (int y = 0; y < height; y++) {

                if (executor.isCancelled()) {
                    break;
                }

                executor.indicateProgress(5 + (35 * ((x * height) + y) / (width * height)));

                orig = input.getImage(x, y);
                newImage = new BufferedImage(orig.getWidth(), orig.getHeight(), // NOPMD SRB
                        BufferedImage.TYPE_INT_RGB);
                grx = newImage.getGraphics();
                grx.drawImage(orig, 0, 0, null);
                output.setImage(x, y, newImage);
                frameMaxima.clear();

                raster = newImage.getRaster();
                imgWidth = raster.getWidth();
                imgHeight = raster.getHeight();

                for (yCoord = EXTREMA_SIZE; yCoord < (imgHeight - EXTREMA_SIZE); yCoord++) {

                    for (xCoord = EXTREMA_SIZE; xCoord < (imgWidth - EXTREMA_SIZE); xCoord++) {

                        raster.getPixel(xCoord, yCoord, pixel);

                        // Only test points with intensity at least 80
                        if (pixel[0] < 80) {
                            continue;
                        }

                        isMax = true;
                        maxDecrease = 0;

outer1:
                        for (deltaY = -EXTREMA_SIZE; deltaY <= EXTREMA_SIZE; deltaY++) {

                            for (deltaX = -EXTREMA_SIZE; deltaX <= EXTREMA_SIZE; deltaX++) {
                                raster.getPixel(xCoord + deltaX, yCoord + deltaY, test);

                                if ((pixel[0] - test[0]) > maxDecrease) {
                                    maxDecrease = pixel[0] - test[0];
                                }

                                if (test[0] > pixel[0]) {
                                    isMax = false;

                                    break outer1;
                                }
                            }
                        }

                        // Do not consider a point if it has a neighbor that is black,
                        // or that is within 20 in intensity of all surrounding pixels
                        if (isMax && (maxDecrease > 20)) {

outer2:
                            for (deltaY = -2; deltaY <= 2; deltaY++) {

                                for (deltaX = -2; deltaX <= 2; deltaX++) {
                                    raster.getPixel(xCoord + deltaX, yCoord + deltaY, test);

                                    if (test[0] == 0) {
                                        isMax = false;

                                        break outer2;
                                    }
                                }
                            }

                            if (isMax) {
                                frameMaxima.add(new Point(xCoord, yCoord)); // NOPMD SRB
                            }
                        }
                    }
                }
```

```
            combineNearbyMaxima(frameMaxima);

            // Turn the maxima points yellow/green, but leave the red intensity equal to the
            // original point's intensity so we don't lose data
            for (Point p2 : frameMaxima) {

                for (int dx = -1; dx <= 1; dx++) {

                    if (((p2.x + dx) < 0) || ((p2.x + dx) >= raster.getWidth())) {
                        continue;
                    }

                    for (int dy = -1; dy <= 1; dy++) {

                        if (((p2.y + dy) < 0) || ((p2.y + dy) >= raster.getHeight())) {
                            continue;
                        }

                        raster.getPixel(p2.x + dx, p2.y + dy, pixel);
                        pixel[1] = 255;
                        pixel[2] = 0;
                        raster.setPixel(p2.x + dx, p2.y + dy, pixel);
                    }
                }

                grx.fillOval((int) p2.getX() - 1, (int) p2.getY() - 1, 3, 3);
                maxima.addPoint(x, y, new ImagePoint(p2.x, p2.y)); // NOPMD SRB
            }
        }
    }

    executor.indicateProgress(41);

    // At this point, maxima have been identified and stored in the PointSetArrayPipeItem, and
    // the maxima have been highlighted on the images.  The next step is to correlate cells
    // from one frame to the next and associate motion vectors with each cell position where
    // the cell moves from frame to frame.  At the same time, we compute the tissue vector for
    // the same location

    if (!executor.isCancelled()) {

        for (int x = 0; x < (width - 1); x++) {

            for (int y = 0; y < height; y++) {

                if (executor.isCancelled()) {
                    break;
                }

                executor.indicateProgress(42
                    + (((x * height) + y) * 36 / ((width - 1) * height)));

                doExtremaMotion(input, maxima, x, y);
            }
        }

        if (!executor.isCancelled()) {

            // Write the CSV file with the points where cells are in each frame
            outfile = new File(pipe.getDir(), "maxima_and_motion.csv");

            try {
                writer = new FileWriter(outfile);

                writer.write(
                    "time, plane, x, y, move_x, move_y, tissue_move_x, tissue_move_y\r\n");

                for (int x = 0; x < (width - 1); x++) {

                    for (int y = 0; y < height; y++) {

                        for (int j = 0; j < maxima.getNumPoints(x, y); j++) {
                            point = maxima.getPoint(x, y, j);
                            writer.write((x + 1) + ", " + (y + 1) + ", " + point.getXPos()
                                + ", " + point.getYPos() + ", " + point.getXVel() + ", "
                                + point.getYVel() + ", " + point.getXAmbientVel() + ", "
                                + point.getYAmbientVel() + "\r\n");
                        }
                    }
                }

                writer.close();
            } catch (IOException e) {
                LOG.log(Level.WARNING, "Exception generating extremal motion", e);
            }
        }
    }
}
```

```
        executor.indicateProgress(79);
    }

    /**
     * Given a list of maxima identified in the image, we scan for any maxima within <code>
     * EXTREMA_SIZE</code> in any direction, and combine them into a single maxima.
     *
     * @param  maxima  the list of identified maxima
     */
    private void combineNearbyMaxima(final List<Point> maxima) {

        List<Point> nearby;
        Point point;
        int totX;
        int totY;

        nearby = new ArrayList<Point>(10);

        // Now collapse multiple points near a single point down
        for (int i = 0; i < maxima.size(); i++) {

            // Get a test point.
            point = maxima.get(i);
            nearby.clear();

            // Find all nearby points
            for (int j = i + 1; j < maxima.size(); j++) {

                if (point.distance(maxima.get(j)) < EXTREMA_SIZE) {
                    nearby.add(maxima.get(j));
                }
            }

            // No nearby points, so move on
            if (nearby.isEmpty()) {
                continue;
            }

            // Average all nearby points
            totX = (int) (point.getX());
            totY = (int) (point.getY());

            for (Point p2 : nearby) {
                totX += p2.getX();
                totY += p2.getY();
                maxima.remove(p2);
            }

            maxima.set(i, new Point(totX / (nearby.size() + 1), totY / (nearby.size() + 1))); // NOPMD SRB
        }
    }

    /**
     * Computes the motion vectors for one time point.
     *
     * @param  images      the images to process
     * @param  maxima      the point set array in which to store maxima that were found
     * @param  timeIndex   the index of the frame we are processing (we compute vectors from this
     *                     time point to the next time point, so this value will always be at least
     *                     two less than then length of the images array
     * @param  plane       the image plane we are processing
     */
    private void doExtremaMotion(final ImageArrayPipeItem images,
        final PointSetArrayPipeItem maxima, final int timeIndex, final int plane) {

        BufferedImage image1;
        BufferedImage image2;
        int count;
        List<ImagePoint> list1;
        List<ImagePoint> list2;
        int distX;
        int distY;
        double dist;
        float corr;
        float bestCorr;
        int which1;
        int which2;
        ImagePoint pt1;
        ImagePoint pt2;

        // Get the images we are comparing
        image1 = images.getImage(timeIndex, plane);
        image2 = images.getImage(timeIndex + 1, plane);

        // Get the maxima from each image into a temporary array
        count = maxima.getNumPoints(timeIndex, plane);
```

```
        list1 = new ArrayList<ImagePoint>(count);

        for (int i = 0; i < count; i++) {
            list1.add(maxima.getPoint(timeIndex, plane, i));
        }

        count = maxima.getNumPoints(timeIndex + 1, plane);
        list2 = new ArrayList<ImagePoint>(count);

        for (int i = 0; i < count; i++) {
            list2.add(maxima.getPoint(timeIndex + 1, plane, i));
        }

        while ((!list1.isEmpty()) && (!list2.isEmpty())) {
            bestCorr = 0.0f;
            which1 = -1;
            which2 = -1;

            for (int i = 0; i < list1.size(); i++) {

                for (int j = 0; j < list2.size(); j++) {
                    pt1 = list1.get(i);
                    pt2 = list2.get(j);

                    distX = pt2.getXPos() - pt1.getXPos();
                    distY = pt2.getYPos() - pt1.getYPos();
                    dist = Math.sqrt((distX * distX) + (distY * distY));

                    if (dist < MAX_FRAME_MOTION) {
                        corr = correlate(image1, pt1, image2, pt2);

                        if (corr > bestCorr) {
                            bestCorr = corr;
                            which1 = i;
                            which2 = j;
                        }
                    }
                }
            }

            // The best match is found - record the vector
            if (which1 == -1) {

                // No more points within acceptable distance, so quit
                break;
            }

            pt1 = list1.get(which1);
            pt2 = list2.get(which2);
            list1.remove(which1);
            list2.remove(which2);

            pt1.setVel(pt2.getXPos() - pt1.getXPos(), pt2.getYPos() - pt1.getYPos());

            // Now, correlate a region surrounding (but not including) the
            // maxima to see how the surrounding tissue moved between frames
            ambientMotion(image1, image2, pt1);
        }
    }

    /**
     * Correlates a point in one frame with a point in another frame.
     *
     * @param   image1  the first frame
     * @param   pt1     the point in the first frame
     * @param   image2  the second frame
     * @param   pt2     the point in the second frame
     * @return  the correlation coefficient
     */
    private float correlate(final BufferedImage image1, final ImagePoint pt1,
        final BufferedImage image2, final ImagePoint pt2) {

        WritableRaster ras1;
        WritableRaster ras2;
        int delta;
        int[] pixel1;
        int[] pixel2;

        ras1 = image1.getRaster();
        ras2 = image2.getRaster();
        delta = 0;
        pixel1 = new int[3];
        pixel2 = new int[3];

        for (int y = -EXTREMA_SIZE; y < EXTREMA_SIZE; y++) {

            for (int x = -EXTREMA_SIZE; x < EXTREMA_SIZE; x++) {
```

```
                ras1.getPixel(pt1.getXPos() + x, pt1.getYPos() + y, pixel1);
                ras2.getPixel(pt2.getXPos() + x, pt2.getYPos() + y, pixel2);

                if (pixel1[0] > pixel2[0]) {
                    delta += pixel1[0] - pixel2[0];
                } else {
                    delta += pixel2[0] - pixel1[0];
                }
            }
        }

        return (delta == 0) ? Float.MAX_VALUE : (1.0f / delta);
    }

    /**
     * Tests the regions surrounding a point in one frame against the region surrounding a point in
     * another frame to see how the ambient field moves
     *
     * @param  image1  the first frame to examine
     * @param  image2  the second frame to examine
     * @param  point   the point about which to detect ambient movement
     */
    private void ambientMotion(final BufferedImage image1, final BufferedImage image2,
        final ImagePoint point) {

        int bestDx;
        int bestDy;
        int width;
        int height;
        int scanWidth;
        int scanHeight;
        double leastSquares;
        double total;
        int count;
        double normalized;
        int pix1;
        int pix2;

        width = image1.getWidth();
        height = image2.getHeight();
        scanWidth = width / 12;
        scanHeight = height / 12;

        // THIS PRODUCES BAD OUTPUT - PROBLEM HERE SOMEWHERE

        bestDx = 0;
        bestDy = 0;
        leastSquares = Double.MAX_VALUE;

        for (int dx = -MAX_FRAME_MOTION; dx <= MAX_FRAME_MOTION; dx++) {

            for (int dy = -MAX_FRAME_MOTION; dy <= MAX_FRAME_MOTION; dy++) {

                total = 0;
                count = 0;

                for (int x = point.getXPos() - scanWidth; x <= (point.getXPos() + scanWidth);
                        x++) {

                    for (int y = point.getYPos() - scanHeight; y <= (point.getYPos() + scanHeight);
                            y++) {

                        if ((x > (point.getXPos() - MAX_FRAME_MOTION))
                                && (x < (point.getXPos() + MAX_FRAME_MOTION))
                                && (y > (point.getYPos() - MAX_FRAME_MOTION))
                                && (y < (point.getYPos() + MAX_FRAME_MOTION))) {
                            continue;
                        }

                        if ((x < 0) || (y < 0) || (x >= width) || (y >= height)) {
                            continue;
                        }

                        if (((x + dx) < 0) || ((y + dy) < 0) || ((x + dx) >= width)
                                || ((y + dy) >= height)) {
                            continue;
                        }

                        pix1 = image1.getRGB(x, y) & 0x00FF;
                        pix2 = image2.getRGB(x + dx, y + dy) & 0x00FF;

                        if ((pix1 == 0) || (pix2 == 0)) {
                            continue;
                        }

                        count++;
                        total += (pix2 - pix1) * (pix2 - pix1);
```

```
                    }
                }

                normalized = total / count;

                if (normalized < leastSquares) {
                    leastSquares = normalized;
                    bestDx = dx;
                    bestDy = dy;
                }
            }
        }

        point.setAmbientVel(bestDx, bestDy);
    }

    /**
     * Generates the string representation of the filter.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        return "ExtremalFinderFilter";
    }
}

package com.srbenoit.microscopy;

import java.awt.image.BufferedImage;
import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import javax.swing.JOptionPane;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;
import com.srbenoit.filter.items.StringPipeItem;
import com.srbenoit.filter.items.TimeSeriesPipeItem;
import com.srbenoit.util.LocalTime;
import loci.formats.ClassList;
import loci.formats.IFormatReader;
import loci.formats.ImageReader;
import loci.formats.in.MetamorphTiffReader;

/**
 * A filter to scan a directory for a set of MetaMorph image series, prompt the user to select a
 * series from those available, load the series files, then prompt the user to designate
 * subsequences of the available frames.
 */
public class MetaMorphReaderFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = 5382801139981082490L;

    /** Sequence name property */
    private static final String SEQ_NAME = "SequenceName";

    /** zero-length string array for list to array conversion */
    private static final String[] STRING_0 = new String[0];

    /**
     * Constructs a new <code>MetaMorphReaderFilter</code>.
     */
    public MetaMorphReaderFilter() {

        super("MetaMorph Series Reader", MetaMorphReaderFilter.class.getName());

        this.outputs.add(new FilterOutput(StringPipeItem.class, "The name of the sequence",
                "sequence_name"));
        this.outputs.add(new FilterOutput(TimeSeriesPipeItem.class, "A series of time stamps",
                "time_series"));
        this.outputs.add(new FilterOutput(ImageArrayPipeItem.class,
                "The raw images from Metamorph", "raw_images"));
        makeRenderer();

        ImageIO.scanForPlugins();
    }

    /**
```

```
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new MetaMorphReaderFilter();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        String[] sequences;
        String name;
        StringPipeItem string;
        TimeSeriesPipeItem series;
        ImageArrayPipeItem images;
        BufferedImage[][] raw;

        validateInputs(pipe);
        executor.indicateProgress(1);

        // Scan for available sequences
        sequences = getSequenceNames(pipe.getDir());

        if (sequences.length == 0) {
            throw new FilterException("No MetaMorph sequences to read.");
        }

        executor.indicateProgress(2);

        // Figure out which sequence we're analyzing
        name = identifySequence(sequences);

        if (name == null) {
            throw new FilterException("No MetaMorph sequence to read.");
        }

        executor.indicateProgress(3);

        string = new StringPipeItem(this.outputs.get(0).getKey(), "Sequence name", pipe);
        string.setData(name);
        pipe.add(string);
        executor.indicateProgress(4);

        series = new TimeSeriesPipeItem(this.outputs.get(1).getKey(), "Time series", pipe);
        pipe.add(series);
        executor.indicateProgress(5);

        raw = runFilter(executor, pipe.getDir(), name, series);

        if (raw == null) {
            throw new FilterException("Unable to load raw images.");
        }

        images = new ImageArrayPipeItem(this.outputs.get(2).getKey(), "Raw images (TIF format)",
                pipe, "t", "z", raw.length, raw[0].length, "tif");
        pipe.add(images);
        executor.indicateProgress(79);

        for (int x = 0; x < raw.length; x++) {

            for (int y = 0; y < raw[x].length; y++) {
                images.setImage(x, y, raw[x][y]);
            }
        }

        executor.indicateProgress(80);

        if (!executor.isCancelled()) {
            pipe.save(executor);
        }

        executor.indicateProgress(100);
    }

    /**
     * Gets the names of all video sequences in the directory.
     *
     * @param   sourceDir  the directory in which to scan
```

```
     * @return  the names of the sequences
     */
    private String[] getSequenceNames(final File sourceDir) {

        List<String> names;
        File[] files;
        String name;
        String lower;
        int index;

        files = sourceDir.listFiles();
        names = new ArrayList<String>(files.length);

        for (File file : files) {
            name = file.getName();
            lower = name.toLowerCase(Locale.US);
            index = lower.indexOf(".nd");

            if (index != -1) {
                name = name.substring(0, index);

                if (!names.contains(name)) {
                    names.add(name);
                }
            }
        }

        return names.toArray(STRING_0);
    }

    /**
     * Given a nonempty list of available sequence names, determine which we want to process.
     *
     * @param   sequences  the list of available sequence names
     * @return  the selected sequence name
     * @throws  FilterException  if the user canceled during sequence selection
     */
    private String identifySequence(final String[] sequences) throws FilterException {

        String old;
        String name;

        old = getProperty(SEQ_NAME);

        if (sequences.length > 1) {

            if (old == null) {
                name = promptForSeqname(sequences);

                if (name == null) {
                    throw new FilterException("No metaMorph sequence to read.");
                }

                setProperty(SEQ_NAME, name);
            } else {
                boolean hit = false;
                name = old;

                for (String test : sequences) {

                    if (name.equals(test)) {
                        hit = true;

                        break;
                    }
                }

                if (!hit) {

                    // The name in the attributes is not valid
                    name = promptForSeqname(sequences);

                    if (name == null) {
                        throw new FilterException("No metaMorph sequence to read.");
                    }

                    setProperty(SEQ_NAME, name);
                }
            }
        } else {
            name = sequences[0];

            if ((old == null) || (!old.equals(name))) {
                setProperty(SEQ_NAME, name);
            }
        }
```

```
        return name;
    }

    /**
     * Prompts the user to choose from a list of available sequences.
     *
     * @param   sequences  the list of names of the available sequences
     * @return  the selected sequence name
     */
    private String promptForSeqname(final String[] sequences) {

        int index;
        String name;

        index = JOptionPane.showOptionDialog(null, "Which sequence would you like to analyze?",
                "Load MetaMorph series", JOptionPane.OK_CANCEL_OPTION,
                JOptionPane.QUESTION_MESSAGE, null, sequences, null);

        if (index == JOptionPane.CLOSED_OPTION) {
            name = null;
        } else {
            name = sequences[index];
        }

        return name;
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param   executor   the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   sourceDir  the directory from which to load Metamorph files
     * @param   seqName    the sequence name to read
     * @param   series     the time series to build as data values are read
     * @return  the array of extracted images
     * @throws  FilterException  if the filter cannot complete
     */
    private BufferedImage[][] runFilter(final FilterTreeExecutor executor, final File sourceDir,
        final String seqName, final TimeSeriesPipeItem series) throws FilterException {

        List<File> files;
        List<File> actual;
        String test;
        String lower;
        boolean found;
        BufferedImage[][] images;

        // Get the list of files we should scan
        files = listFiles(sourceDir, seqName);
        executor.indicateProgress(6);

        if (files.isEmpty()) {
            throw new FilterException("No metaMorph data files found.");
        }

        // Get an ordered list of the files representing time points
        actual = new ArrayList<File>(files.size());

        for (int t = 1; t <= files.size(); t++) {

            if (executor.isCancelled()) {
                break;
            }

            test = "_t" + t + ".tif";
            found = false;

            for (File file : files) {
                lower = file.getName().toLowerCase(Locale.getDefault());

                if (lower.endsWith(test)) {
                    actual.add(file);
                    found = true;

                    break;
                }
            }

            if (!found) {
                break;
            }
        }

        executor.indicateProgress(7);

        if (actual.isEmpty()) {
```

```
            throw new FilterException("No metaMorph data files found.");
        }

        images = new BufferedImage[actual.size()][];

        // Load the time points
        if (!executor.isCancelled()) {

            for (int t = 1; t <= actual.size(); t++) {
                images[t - 1] = loadTimePoint(actual.get(t - 1), t, series);
                executor.indicateProgress(8 + (t * 70 / actual.size()));
            }
        }

        // Set up time point subsequences

        return images;
    }

    /**
     * Loads a TIF file which may contain more than one image plane.
     *
     * @param   file        the file to read
     * @param   timeIndex   the time index
     * @param   series      the time series to build as data values are read
     * @return  the set of loaded Z planes for the specified time point
     * @throws  FilterException  if the filter cannot complete
     */
    private BufferedImage[] loadTimePoint(final File file, final int timeIndex,
        final TimeSeriesPipeItem series) throws FilterException {

        ClassList<IFormatReader> list;
        ImageReader reader;
        int numPlanes;
        int width;
        int height;
        int bpp;
        int channels;
        int bytesPer;
        byte[] data;
        int[] combined;
        int type;
        BufferedImage[] planes;
        int index;
        Map<String, Object> meta;
        Object obj;
        LocalTime time;

        list = new ClassList<IFormatReader>(IFormatReader.class);
        list.addClass(MetamorphTiffReader.class);

        reader = new ImageReader(list);

        try {
            reader.setId(file.getAbsolutePath());

            width = reader.getSizeX();
            height = reader.getSizeY();
            bpp = reader.getBitsPerPixel();

            // Extract the time of the exposure
            meta = reader.getGlobalMetadata();

            if (meta == null) {
                time = new LocalTime();
                time.setMillis(timeIndex);
            } else {
                obj = meta.get("acquisition-time-local");

                if (obj == null) {
                    obj = meta.get("DateTime");

                    if (obj == null) {
                        obj = meta.get("modification-time-local");
                    }
                }

                if (obj == null) {
                    time = new LocalTime();
                    time.setMillis(timeIndex);
                } else {
                    time = extractTime(obj.toString());
                }
            }

            series.addTimePoint(time);
```

```
            bytesPer = (bpp + 7) / 8;
            channels = reader.getRGBChannelCount();
            numPlanes = reader.getImageCount();
            planes = new BufferedImage[numPlanes];

            for (int z = 0; z < numPlanes; z++) {

                data = reader.openBytes(z);
                combined = new int[data.length / bytesPer]; // NOPMD SRB

                index = 0;

                for (int i = 0; i < combined.length; i++) {

                    combined[i] = data[index] & 0x00FF;

                    for (int j = 1; j < bytesPer; j++) {
                        combined[i] += (data[index + j] & 0x00FF) << (8 * j);
                    }

                    index += bytesPer;
                }

                if (channels == 1) {

                    switch (bytesPer) {

                    case 1:
                        type = BufferedImage.TYPE_BYTE_GRAY;

                        break;

                    case 2:
                        type = BufferedImage.TYPE_USHORT_GRAY;

                        break;

                    default:

                        continue;
                    }
                } else if (channels == 3) {

                    switch (bytesPer) {

                    case 3:
                        type = BufferedImage.TYPE_INT_RGB;

                        break;

                    case 4:
                        type = BufferedImage.TYPE_INT_ARGB;

                        break;

                    default:

                        continue;
                    }
                } else {
                    continue;
                }

                planes[z] = new BufferedImage(width, height, type); // NOPMD SRB

                index = 0;

                for (int y = 0; y < height; y++) {

                    for (int x = 0; x < width; x++) {
                        planes[z].getRaster().setSample(x, y, 0, combined[index]);
                        index++;
                    }
                }
            }

        } catch (Exception e) {
            LOG.log(Level.WARNING, "Exception reading file", e);
            throw new FilterException("Exception reading file", e);
        }

        return planes;
    }

    /**
     * Retrieves a list of the source files in the target directory for the target sequence.
     *
```

```
     * @param   dir   the directory in which to search
     * @param   name  the sequence name
     * @return  the list of files
     */
    private List<File> listFiles(final File dir, final String name) {

        List<File> list;
        File[] files;
        String lower;

        // list the files and cull those that are not relevant
        files = dir.listFiles();

        list = new ArrayList<File>(files.length);

        for (int i = 0; i < files.length; i++) {
            lower = files[i].getName().toLowerCase(Locale.getDefault());

            if (files[i].getName().startsWith(name) && (!files[i].isDirectory())
                    && (!lower.endsWith(".nd")) && (!lower.contains("_thumb_"))) {
                list.add(files[i]);
            }
        }

        return list;
    }

    /**
     * Converts a timestamp string into a <code>LocalTime</code>. The input data will be in the
     * format: '20100902 08:09:13.787'.
     *
     * @param   time  the time string to parses
     * @return  the fixed time
     * @throws  FilterException  if the timestamp cannot be parsed
     */
    private LocalTime extractTime(final String time) throws FilterException {

        LocalTime fixed;

        fixed = new LocalTime();

        if ((time.length() >= 19) && (time.charAt(8) == '_') && (time.charAt(11) == ':')
                && (time.charAt(14) == ':') && (time.charAt(17) == '.')) {

            try {
                fixed.setYear(Integer.parseInt(time.substring(0, 4)));
                fixed.setMonth(Integer.parseInt(time.substring(4, 6)));
                fixed.setDay(Integer.parseInt(time.substring(6, 8)));
                fixed.setHour(Integer.parseInt(time.substring(9, 11)));
                fixed.setMinute(Integer.parseInt(time.substring(12, 14)));
                fixed.setSecond(Integer.parseInt(time.substring(15, 17)));
                fixed.setMillis(Integer.parseInt(time.substring(18)));
            } catch (NumberFormatException e) {
                throw new FilterException("Invalid timestamp: '" + time + "'", e);
            }
        } else {
            throw new FilterException("Invalid timestamp: '" + time + "'");
        }

        return fixed;
    }

    /**
     * Generates the string representation of the filter.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        return "MetaMorphReaderFilter";
    }
}

package com.srbenoit.microscopy;

import java.awt.image.BufferedImage;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;

/**
 * A filter that compensates for slight motions in the video content, potentially increasing the
 * size of the frames to contain the shifted images.
```

```
 */
public class MotionCompensationFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = -3712792142848530198L;

    /**
     * Constructs a new <code>MotionCompensationFilter</code>.
     */
    public MotionCompensationFilter() {

        super("Motion Compensation", MotionCompensationFilter.class.getName());

        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Images to stabilize"));
        this.outputs.add(new FilterOutput(ImageArrayPipeItem.class, "Stabilized images",
                "stabilized images"));
        makeRenderer();
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new MotionCompensationFilter();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ImageArrayPipeItem input;
        ImageArrayPipeItem output;

        validateInputs(pipe);
        executor.indicateProgress(1);

        input = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());
        executor.indicateProgress(2);

        // First two pipe items are passed through to the output - the image
        // item must be re-created since the number of Z planes changes
        output = new ImageArrayPipeItem(this.outputs.get(0).getKey(),
                "Motion-compensated images (PNG format)", pipe, input.getXLabel(),
                input.getYLabel(), input.getXSize(), 1, "png");
        pipe.add(output);
        executor.indicateProgress(3);

        runFilter(executor, input, output);

        executor.indicateProgress(80);

        if (!executor.isCancelled()) {
            pipe.save(executor);
        }

        executor.indicateProgress(100);
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param  executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param  input     the images to process
     * @param  output    the image array in which to store the processed images
     */
    private void runFilter(final FilterTreeExecutor executor, final ImageArrayPipeItem input,
        final ImageArrayPipeItem output) {

        BufferedImage source;
        int width;
        int height;
        int[] bestDx;
        int[] bestDy;
        int minX;
        int maxX;
        int minY;
        int maxY;
```

```
double leastSquares;
int cumX;
int cumY;
double normalized;
int newWidth;
int newHeight;
BufferedImage newImage;
int testDx;
int testDy;

source = input.getImage(0, 0);
width = source.getWidth();
height = source.getHeight();

// Now we take each pair of frames and compute the best (dx, dy)
bestDx = new int[input.getXSize()];
bestDy = new int[input.getXSize()];

for (int t = 1; t < input.getXSize(); t++) {

    if (executor.isCancelled()) {
        break;
    }

    // Scan the range of possible (dx, dy) and for each, compute the
    // difference between images. If this difference is less than the
    // current least difference, this is the new best (dx, dy).
    leastSquares = Long.MAX_VALUE;

    for (int dx = -18; dx <= 18; dx += 3) {

        for (int dy = -18; dy <= 18; dy += 3) {
            normalized = difference(input.getImage(t - 1, 0), input.getImage(t, 0), dx,
                    dy);

            if (normalized < leastSquares) {
                leastSquares = normalized;
                bestDx[t] = dx;
                bestDy[t] = dy;
            }
        }
    }

    executor.indicateProgress(5 + (40 * t / input.getXSize()));

    testDx = bestDx[t];
    testDy = bestDy[t];

    for (int dx = testDx - 3; dx <= (testDx + 3); dx++) {

        for (int dy = testDy - 3; dy <= (testDy + 3); dy++) {

            if ((dx == testDx) && (dy == testDy)) {
                continue;
            }

            normalized = difference(input.getImage(t - 1, 0), input.getImage(t, 0), dx,
                    dy);

            if (normalized < leastSquares) {
                leastSquares = normalized;
                bestDx[t] = dx;
                bestDy[t] = dy;
            }
        }
    }
}

// Now that we have the best DX, DY for each frame, compute the
// window that the cumulative moves would occupy
if (!executor.isCancelled()) {
    minX = 0;
    maxX = 0;
    minY = 0;
    maxY = 0;
    cumX = 0;
    cumY = 0;

    for (int t = 1; t < input.getXSize(); t++) {
        cumX += bestDx[t];
        cumY += bestDy[t];

        if (cumX > maxX) {
            maxX = cumX;
        }

        if (cumX < minX) {
```

```
                minX = cumX;
            }

            if (cumY > maxY) {
                maxY = cumY;
            }

            if (cumY < minY) {
                minY = cumY;
            }
        }

        // Now allocate new images and draw the original images at the proper
        // cumulative offsets
        cumX = -minX;
        cumY = -minY;
        newWidth = width + (maxX - minX);
        newHeight = height + (maxY - minY);

        // Ensure width and height are multiples of 16 pixels
        newWidth = ((newWidth + 15) / 16) * 16;
        newHeight = ((newHeight + 15) / 16) * 16;

        for (int x = 0; x < input.getXSize(); x++) {

            for (int y = 0; y < input.getYSize(); y++) {

                if (executor.isCancelled()) {
                    break;
                }

                cumX += bestDx[x];
                cumY += bestDy[x];

                executor.indicateProgress(45 + (35 * x / input.getXSize()));

                newImage = new BufferedImage(newWidth, newHeight, // NOPMD SRB
                        BufferedImage.TYPE_INT_RGB);
                newImage.getGraphics().drawImage(input.getImage(x, y), cumX, cumY, null);
                output.setImage(x, y, newImage);
            }
        }
    }
}

/**
 * Compute the mean difference between two images with the second offset by a particular
 * vector. This is computed as the average of the square of the difference in intensity between
 * the center 1/4 of the two images, after the offset has been applied.
 *
 * @param   first   the first image
 * @param   second  the second image
 * @param   xOff    the X offset to apply to the second image
 * @param   yOff    the T offset to apply to the second image
 * @return  the correlation (normalized square of intensity differences)
 */
private double difference(final BufferedImage first, final BufferedImage second,
    final int xOff, final int yOff) {

    int minX;
    int maxX;
    int minY;
    int maxY;
    int avgX;
    int avgY;
    int rgb1;
    int rgb2;
    int diff;
    long total;

    // Compute range (in first image) of correlation
    minX = (xOff > 0) ? xOff : 0;
    minY = (yOff > 0) ? yOff : 0;
    maxX = (xOff < 0) ? (first.getWidth() + xOff) : first.getWidth();
    maxY = (yOff < 0) ? (first.getHeight() + yOff) : first.getHeight();

    // We correlate only the middle 1/4 of the image
    avgX = (minX + maxX) / 2;
    avgY = (minY + maxY) / 2;
    diff = maxX - minX;
    minX = avgX - (diff / 8);
    maxX = avgX + (diff / 8);
    diff = maxY - minY;
    minY = avgY - (diff / 8);
    maxY = avgY + (diff / 8);

    total = 0;
```

```
            for (int x = minX; x < maxX; x++) {

                for (int y = minY; y < maxY; y++) {
                    rgb1 = first.getRGB(x, y);
                    rgb2 = second.getRGB(x - xOff, y - yOff);
                    diff = (rgb1 & 0x00FF) - (rgb2 & 0x00FF);
                    total += diff * diff;
                }
            }

            return (double) total / ((maxY - minY) * (maxX - minX));
        }

        /**
         * Generates the string representation of the filter.
         *
         * @return  the string representation
         */
        @Override public String toString() {

            return "MotionCompensationFilter";
        }
}

package com.srbenoit.microscopy;

import java.awt.image.BufferedImage;
import java.io.File;
import java.util.logging.Level;
import javax.media.MediaLocator;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;
import com.srbenoit.media.movie.MakeMovie;
import com.srbenoit.media.movie.MovieMakingException;

/**
 * A filter that generates QuickTime movies for each X series in an image array.
 */
public class QuicktimeBuilderFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = 9105886485480586659L;

    /**
     * Constructs a new <code>QuicktimeBuilderFilter</code>.
     */
    public QuicktimeBuilderFilter() {

        super("QuickTime Builder", QuicktimeBuilderFilter.class.getName());

        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Movie frames"));
        makeRenderer();
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new QuicktimeBuilderFilter();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ImageArrayPipeItem images;

        validateInputs(pipe);
        executor.indicateProgress(1);

        images = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());
        executor.indicateProgress(2);
```

```
        runFilter(executor, pipe, images);

        // No need to save - we don't add anything to the pipe

        executor.indicateProgress(100);
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param  executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param  pipe      a pipe containing the input data items
     * @param  images    the images to process
     */
    private void runFilter(final FilterTreeExecutor executor, final Pipe pipe,
        final ImageArrayPipeItem images) {

        int numMovies;
        int numFrames;
        BufferedImage[] frames;
        int count;
        File file;
        MediaLocator loc;
        MakeMovie maker;

        numFrames = images.getXSize();
        numMovies = images.getYSize();
        maker = new MakeMovie();

        frames = new BufferedImage[numFrames];

        for (int i = 0; i < numMovies; i++) {

            if (executor.isCancelled()) {
                break;
            }

            executor.indicateProgress(5 + (i * 70 / numMovies));
            count = 0;

            for (int j = 0; j < numFrames; j++) {
                frames[j] = images.getImage(j, i);

                if (frames[j] == null) {
                    break;
                }

                count++;
            }

            if (numMovies > 1) {
                file = new File(pipe.getDir(), // NOPMD SRB
                        this.getInputFormat(0).getKey() + "_movie_plane_" + (i + 1) + ".mov");
            } else {
                file = new File(pipe.getDir(), // NOPMD SRB
                        this.getInputFormat(0).getKey() + "_movie.mov");
            }

            loc = MakeMovie.createMediaLocator("file:" + file.getAbsolutePath());

            if (count > 0) {

                try {
                    maker.doItBuffered(frames[0].getWidth(), frames[0].getHeight(), count, frames,
                        loc);
                } catch (MovieMakingException ex) {
                    LOG.log(Level.WARNING, "Exception creating movie", ex);
                }
            }
        }
    }

    /**
     * Generates the string representation of the filter.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        return "QuicTimeBuilderFilter";
    }
}

package com.srbenoit.microscopy;

import java.awt.image.BufferedImage;
```

```
import java.awt.image.ConvolveOp;
import java.awt.image.Kernel;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;

/**
 * A filter that applies a Gaussian smoothing kernel to every image in an image array.
 */
public class SmootherFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = 9105886485480586654L;

    /** size of kernel in smoothing operation. */
    private static final int KERNEL_SIZE = 5;

    /**
     * Constructs a new <code>SmootherFilter</code>.
     */
    public SmootherFilter() {

        super("Gaussian Smoother", SmootherFilter.class.getName());

        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Images to smooth"));
        this.outputs.add(new FilterOutput(ImageArrayPipeItem.class, "Smoothed imags",
                "smoothed images"));
        makeRenderer();
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new SmootherFilter();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ImageArrayPipeItem input;
        ImageArrayPipeItem output;

        validateInputs(pipe);
        executor.indicateProgress(1);

        input = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());
        executor.indicateProgress(2);

        // First two pipe items are passed through to the output - the image
        // item must be re-created since the number of Z planes changes
        output = new ImageArrayPipeItem(this.outputs.get(0).getKey(),
                "Gaussian-smoothed images (PNG format)", pipe, input.getXLabel(),
                input.getYLabel(), input.getXSize(), input.getYSize(), "png");
        pipe.add(output);
        executor.indicateProgress(3);

        runFilter(executor, input, output);

        executor.indicateProgress(80);

        if (!executor.isCancelled()) {
            pipe.save(executor);
        }

        executor.indicateProgress(100);
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param  executor  the <code>FilterTreeExecutor</code> that is executing the filter
```

```
     * @param  pipe       a pipe containing the input data items
     * @param  input      the images to process
     * @param  output     the image array in which to store the processed images
     */
    private void runFilter(final FilterTreeExecutor executor, final ImageArrayPipeItem input,
        final ImageArrayPipeItem output) {

        ConvolveOp convOp;
        int width;
        int height;
        BufferedImage orig;
        BufferedImage fixed;
        BufferedImage dest;

        convOp = makeKernel();

        width = input.getXSize();
        height = input.getYSize();

        // Scan each image to get mean intensity
        for (int x = 0; x < width; x++) {

            for (int y = 0; y < height; y++) {

                if (executor.isCancelled()) {
                    break;
                }

                executor.indicateProgress(5 + (75 * ((x * height) + y) / (width * height)));

                orig = input.getImage(x, y);
                fixed = new BufferedImage(orig.getWidth(), orig.getHeight(), // NOPMD SRB
                        BufferedImage.TYPE_INT_RGB);
                fixed.getGraphics().drawImage(orig, 0, 0, null);
                dest = new BufferedImage(orig.getWidth(), orig.getHeight(), // NOPMD SRB
                        BufferedImage.TYPE_INT_RGB);
                convOp.filter(fixed, dest);
                output.setImage(x, y, dest);
            }
        }
    }

    /**
     * Builds the Gaussian kernel that will be used to convolve the image.
     *
     * @return  the convolution kernel
     */
    private ConvolveOp makeKernel() {

        float[] data;
        double gVal;
        float total;

        // Build the kernel
        // First, we compute a 2-D Gaussian kernel (SIGMA = 1)
        // G(x,y) = (1/(2 Pi)) EXP (-x^2-y^2)
        data = new float[KERNEL_SIZE * KERNEL_SIZE];
        total = 0.0f;

        for (int x = 0; x < KERNEL_SIZE; x++) {

            for (int y = 0; y < KERNEL_SIZE; y++) {
                gVal = (1 / (2 * Math.PI)) + Math.exp(-Math.pow(x - 3, 2) - Math.pow(y - 3, 2));
                data[(y * KERNEL_SIZE) + x] = (float) gVal;
                total += data[(y * KERNEL_SIZE) + x];
            }
        }

        // Normalize the kernel
        for (int x = 0; x < KERNEL_SIZE; x++) {

            for (int y = 0; y < KERNEL_SIZE; y++) {
                data[(y * KERNEL_SIZE) + x] /= total;
            }
        }

        return new ConvolveOp(new Kernel(KERNEL_SIZE, KERNEL_SIZE, data), ConvolveOp.EDGE_NO_OP,
                null);
    }

    /**
     * Generates the string representation of the filter.
     *
     * @return  the string representation
     */
    @Override public String toString() {
```

```
        return "SmootherFilter";
    }
}

package com.srbenoit.microscopy;

import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;
import com.srbenoit.filter.items.ImagePoint;
import com.srbenoit.filter.items.PointSetArrayPipeItem;
import com.srbenoit.filter.items.Trajectory;
import com.srbenoit.filter.items.TrajectoryListPipeItem;

/**
 * A filter that generates a set of trajectories from a set of maxima point arrays.
 */
public class TrajectoryFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = 9105886485480586655L;

    /** width of border around thumbnails */
    private static final int BORDER = 15;

    /** the set of paths we have found so far */
    private final List<Trajectory> paths;

    /**
     * Constructs a new <code>TrajectoryFilter</code>.
     */
    public TrajectoryFilter() {

        super("Trajectory Assembler", TrajectoryFilter.class.getName());

        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Motion-compensated images"));
        this.inputs.add(new FilterInput(PointSetArrayPipeItem.class,
                "Points of maximal intensity"));
        this.outputs.add(new FilterOutput(TrajectoryListPipeItem.class, "Assembled trajectories",
                "trajectories"));
        this.outputs.add(new FilterOutput(ImageArrayPipeItem.class,
                "Thumbnail images around each trajectory", "thumbnails"));
        makeRenderer();

        this.paths = new ArrayList<Trajectory>(10);
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new TrajectoryFilter();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ImageArrayPipeItem images;
        PointSetArrayPipeItem maxima;

        validateInputs(pipe);
        executor.indicateProgress(1);
```

```
        images = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());
        maxima = (PointSetArrayPipeItem) pipe.get(this.inputs.get(1).getKey());
        executor.indicateProgress(2);

        runFilter(executor, pipe, images, maxima);

        executor.indicateProgress(80);

        if (!executor.isCancelled()) {
            pipe.save(executor);
        }

        executor.indicateProgress(100);
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param  executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param  pipe      a pipe containing the input data items
     * @param  images    the images to process
     * @param  maxima    the maxima to assemble into a trajectories
     */
    private void runFilter(final FilterTreeExecutor executor, final Pipe pipe,
        final ImageArrayPipeItem images, final PointSetArrayPipeItem maxima) {

        TrajectoryListPipeItem trajectories;
        ImageArrayPipeItem thumbnails;
        File outfile;
        Trajectory traj;
        FileWriter writer;
        ImagePoint point;
        Point relative;

        trajectories = new TrajectoryListPipeItem(this.outputs.get(0).getKey(),
                "Trajectories of maxima", pipe);
        pipe.add(trajectories);

        // Do the trajectory analysis before we filter the frames
        aggregateExtremaVectors(executor, maxima, trajectories);

        thumbnails = new ImageArrayPipeItem(this.outputs.get(1).getKey(),
                "Thumbnails surrounding each maxima (PNG format)", pipe, "t", "traj",
                images.getXSize(), this.paths.size(), "png");
        pipe.add(thumbnails);

        // Mark up the images with the trajectories
        for (int i = 0; i < this.paths.size(); i++) {

            if (executor.isCancelled()) {
                break;
            }

            executor.indicateProgress(50 + ((i * 20) / this.paths.size()));
            makeThumbnails(i, this.paths.get(i), images, thumbnails);
        }

        if (!executor.isCancelled()) {
            outfile = new File(pipe.getDir(), "trajectories.csv");

            try {
                writer = new FileWriter(outfile);

                writer.write(
                    "Trajectory, Time Index, Plane, X, Y, Vel X, Vel Y, Tissue Vel X, Tissue Vel Y, Relative X, Rel

                for (int i = 0; i < this.paths.size(); i++) {
                    executor.indicateProgress(70 + ((i * 10) / this.paths.size()));

                    traj = this.paths.get(i);

                    for (int j = 0; j < traj.numPoints(); j++) {

                        point = traj.getPoint(j);
                        relative = traj.getRelative(j);

                        writer.write((i + 1) + ", " + traj.getTimePoint(j) + ", "
                            + traj.getPlane(j) + ", " + point.getXPos() + ", " + point.getYPos()
                            + ", " + point.getXVel() + ", " + point.getYVel() + ", "
                            + point.getXAmbientVel() + ", " + point.getYAmbientVel() + ", "
                            + relative.x + ", " + relative.y + "\r\n");
                    }
                }

                writer.close();
            } catch (IOException e) {
```

```
            LOG.log(Level.WARNING, "Exception writing trajectories.csv file", e);
        }
    }
}

/**
 * Searches for lists of maxima that can be connected and extract trajectories.
 *
 * @param  executor      the <code>FilterTreeExecutor</code> that is executing the filter
 * @param  maxima        the array of maxima values and vectors
 * @param  trajectories  the list to which to add discovered trajectories
 */
public void aggregateExtremaVectors(final FilterTreeExecutor executor,
    final PointSetArrayPipeItem maxima, final TrajectoryListPipeItem trajectories) {

    ImagePoint point;
    boolean found;
    Trajectory traj;
    int numTraj;
    Rectangle extents;

    for (int time = 0; time < (maxima.getXSize() - 1); time++) {

        if (executor.isCancelled()) {
            break;
        }

        executor.indicateProgress(5 + ((time * 45) / (maxima.getXSize() - 1)));

        for (int plane = 0; plane < maxima.getYSize(); plane++) {

            for (int i = 0; i < maxima.getNumPoints(time, plane); i++) {
                point = maxima.getPoint(time, plane, i);

                // See if the point can tie in with an existing trajectory
                found = false;

                // Test within the same plane first...
                for (Trajectory path : this.paths) {

                    if ((path.getCurrentX() == point.getXPos())
                            && (path.getCurrentY() == point.getYPos())
                            && (path.getCurrentPlane() == plane)) {

                        found = true;
                        path.addPoint(time, plane, point);

                        break;
                    }
                }

                if (!found) {

                    // Test adjacent planes next...
                    for (Trajectory path : this.paths) {

                        if ((path.getCurrentX() == point.getXPos())
                                && (path.getCurrentY() == point.getYPos())
                                && ((path.getCurrentPlane() == (plane - 1))
                                        || (path.getCurrentPlane() == (plane + 1)))) {

                            found = true;
                            path.addPoint(time, plane, point);

                            break;
                        }
                    }
                }

                // If no existing trajectory matched, start a new one
                if (!found) {
                    traj = new Trajectory(); // NOPMD SRB
                    traj.addPoint(time, plane, point);
                    this.paths.add(traj);
                }
            }
        }
    }

    // Delete any trajectories of less than 10 steps or whose total motion is less than 5
    if (!executor.isCancelled()) {
        numTraj = this.paths.size();

        for (int i = numTraj - 1; i >= 0; i--) {
            traj = this.paths.get(i);
            extents = traj.extents();
```

```
                if (traj.points.size() < 10) {
                    this.paths.remove(i);
                } else if ((extents.width < 5) && (extents.height < 5)) {
                    this.paths.remove(i);
                } else {
                    trajectories.addTrajectory(traj);
                }
            }
        }
    }

    /**
     * Generates the thumbnail frames for a trajectory.
     *
     * @param  index        the index of the trajectory
     * @param  traj         the trajectory whose thumbnail frames are to be generated
     * @param  srcImages    the source images (with maxima marked)
     * @param  destImages   an image array in which to add the thumbnail frames
     */
    private void makeThumbnails(final int index, final Trajectory traj,
        final ImageArrayPipeItem srcImages, final ImageArrayPipeItem destImages) {

        Rectangle extents;
        ImagePoint point;
        Point relative;
        BufferedImage src;
        BufferedImage dest;
        Graphics2D grx;
        int xPos;
        int yPos;

        extents = traj.extents();

        for (int i = 0; i < traj.numPoints(); i++) {

            point = traj.getPoint(i);
            relative = traj.getRelative(i);

            src = srcImages.getImage(traj.getTimePoint(i), traj.getPlane(i));
            dest = new BufferedImage(extents.width + (2 * BORDER), // NOPMD SRB
                    extents.height + (2 * BORDER), BufferedImage.TYPE_INT_RGB);
            grx = (Graphics2D) dest.getGraphics();

            xPos = -point.getXPos() + (relative.x - extents.x) + BORDER;
            yPos = -point.getYPos() + (relative.y - extents.y) + BORDER;

            grx.drawImage(src, xPos, yPos, null);
            destImages.setImage(traj.getTimePoint(i), index, dest);
        }
    }

    /**
     * Generates the string representation of the filter.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        return "TrajectoryFilter";
    }
}

package com.srbenoit.microscopy;

import java.awt.image.BufferedImage;
import com.srbenoit.filter.AbstractFilter;
import com.srbenoit.filter.FilterException;
import com.srbenoit.filter.FilterInput;
import com.srbenoit.filter.FilterOutput;
import com.srbenoit.filter.FilterTreeExecutor;
import com.srbenoit.filter.Pipe;
import com.srbenoit.filter.items.ImageArrayPipeItem;

/**
 * A filter that merges all z planes in an image array, producing a new image array that has 1
 * plane per time point.
 */
public class ZPlaneMergerFilter extends AbstractFilter {

    /** version number for serialization */
    private static final long serialVersionUID = 4347190635894369722L;

    /**
     * Constructs a new <code>ZPlaneMergerFilter</code>.
     */
    public ZPlaneMergerFilter() {
```

```
        super("Z-plane merger", ZPlaneMergerFilter.class.getName());

        this.inputs.add(new FilterInput(ImageArrayPipeItem.class, "Images to be merged"));
        this.outputs.add(new FilterOutput(ImageArrayPipeItem.class, "Merged images",
                "merged images"));
        makeRenderer();
    }

    /**
     * Duplicates the filter including all of its settings, but returns an independent object.
     *
     * @return  the duplicated object
     */
    @Override public AbstractFilter duplicate() {

        return new ZPlaneMergerFilter();
    }

    /**
     * Performs the filter operation.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   pipe      a pipe containing the input data items
     * @throws  FilterException  if the filter cannot complete
     */
    @Override public void filter(final FilterTreeExecutor executor, final Pipe pipe)
        throws FilterException {

        ImageArrayPipeItem input;
        ImageArrayPipeItem output;

        validateInputs(pipe);
        executor.indicateProgress(1);

        input = (ImageArrayPipeItem) pipe.get(this.inputs.get(0).getKey());
        executor.indicateProgress(2);

        output = new ImageArrayPipeItem(this.outputs.get(0).getKey(),
                "Z-plane merged images (PNG format)", pipe, input.getXLabel(), input.getYLabel(),
                input.getXSize(), 1, "png");
        pipe.add(output);
        executor.indicateProgress(3);

        runFilter(executor, input, output);

        executor.indicateProgress(80);

        if (!executor.isCancelled()) {
            pipe.save(executor);
        }

        executor.indicateProgress(100);
    }

    /**
     * Runs the filter, reading the source Metamorph TIF files and extracting an array of images,
     * the first dimension of which is time, and the second dimension of which is z plane.
     *
     * @param   executor  the <code>FilterTreeExecutor</code> that is executing the filter
     * @param   input     the images to process
     * @param   output    the image array in which to store the merged images
     */
    private void runFilter(final FilterTreeExecutor executor, final ImageArrayPipeItem input,
        final ImageArrayPipeItem output) {

        int width;
        BufferedImage[] src;
        BufferedImage merged;

        width = input.getXSize();

        for (int x = 0; x < width; x++) {

            if (executor.isCancelled()) {
                break;
            }

            executor.indicateProgress(5 + (75 * x / width));
            src = input.getImages(x);
            merged = mergeImages(src);
            output.setImage(x, 0, merged);
        }
    }

    /**
     * Merge the images in all planes into a composite merged image. The maximum intensity value
     * from the set of planes is accepted as the merged image intensity value.
```

```
     *
     * @param   source  the source images to be merged
     * @return  the merged image
     */
    public static BufferedImage mergeImages(final BufferedImage[] source) {

        BufferedImage result;
        int width;
        int height;
        int effWidth;
        int effHeight;
        int rgb;
        int red;
        int green;
        int blue;

        width = source[0].getWidth();
        height = source[0].getHeight();

        result = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

        // Determine the effective width and height of the merged image
        effWidth = width;
        effHeight = height;

        for (int i = 0; i < source.length; i++) {

            if (source[i].getWidth() < effWidth) {
                effWidth = source[i].getWidth();
            }

            if (source[i].getHeight() < effHeight) {
                effHeight = source[i].getHeight();
            }
        }

        // Perform the merge operation
        for (int x = 0; x < effWidth; x++) {

            for (int y = 0; y < effHeight; y++) {
                red = 0;
                green = 0;
                blue = 0;

                for (int i = 0; i < source.length; i++) {
                    rgb = source[i].getRGB(x, y);

                    if ((rgb & 0x000000FF) > red) {
                        red = rgb & 0x000000FF;
                    }

                    if ((rgb & 0x0000FF00) > green) {
                        green = rgb & 0x0000FF00;
                    }

                    if ((rgb & 0x00FF0000) > blue) {
                        blue = rgb & 0x00FF0000;
                    }
                }

                rgb = red | green | blue;
                result.setRGB(x, y, rgb);
            }
        }

        return result;
    }

    /**
     * Generates the string representation of the filter.
     *
     * @return  the string representation
     */
    @Override public String toString() {

        return "ZPlaneMergerFilter";
    }
}
```

E.6 Mathematical Modeling

E.6.1 Grids for Neighbor Finding (com.srbenoit.modeling.grid)

This package provides classes to manage a grid that manages a objects within a two dimensional rectangle for molecular and granular dynamics simulations. Each object is assigned to a grid cell based on its coordinates, making the location of nearby objects much faster than searching all objects in the grid.

```
package com.srbenoit.modeling.grid;

import com.srbenoit.log.LoggedObject;
import com.srbenoit.sparsearray.SparseArray;
import com.srbenoit.sparsearray.SparseArrayListener;

/**
 * The base class for grid that may contain objects . Contained objects must be derived from <code>
 * GridMember</code>. This class provides a very fast way to iterate over the set of potential
 * neighbors of an object.
 *
 * <p>The grid manages a sparse array data structure that assigns each added object an array index
 * that does not change while the object is part of the grid. Removing objects may open gaps in the
 * array, which newly added objects will fill first before expanding the array
 */
public abstract class AbstractGrid2D extends LoggedObject implements SparseArrayListener {

    /** number of objects in an allocated block (a multiple of 64) */
    protected static final int BLOCK_SIZE = 128;

    /** a pre-allocated array of zero integers */
    protected static final int[] ZERO_INTS = new int[0];

    /** width and height of the grid cells (they must be square) */
    protected final double gridCellSize;

    /** the list containing grid objects (size >= N) */
    private SparseArray<GridMember2Int> objects;

    /**
     * Constructs a new <code>AbstractGrid</code>.
     *
     * @param  cellSize  width and height of the grid cells
     */
    public AbstractGrid2D(final double cellSize) {

        this.gridCellSize = cellSize;
        this.objects = new SparseArray<GridMember2Int>(GridMember2Int.class, 128);

        this.objects.addListener(this);
    }

    /**
     * Gets the size of a grid cell.
     *
     * @return  the grid cell size
     */
    public double getGridCellSize() {

        return this.gridCellSize;
    }

    /**
     * Given an index, returns the index of the next <code>GridMember</code>.
     *
     * @param   index  the index to test
     * @return  the index of the next set bit after (or including) the given index
     */
    public int getNextFilled(final int index) {

        return this.objects.nextFilled(index);
    }

    /**
```

```
     * Gets a particular member of the grid.
     *
     * @param    index  the index of the member to retrieve
     * @return   the grid member, or <code>null</code> if that index is empty
     */
    public GridMember2Int get(final int index) {

        return this.objects.get(index);
    }

    /**
     * Adds a member object to the grid.
     *
     * @param    member  the grid member to add to the grid
     * @return   the immutable integer index of the member in the grid's object array
     */
    public int add(final GridMember2Int member) {

        int index;

        index = this.objects.add(member);

        // Let subclasses process the addition
        memberAdded(index);

        return index;
    }

    /**
     * Gets the number of objects in the grid.
     *
     * @return   the number of objects
     */
    public int getNumObjects() {

        return this.objects.size();
    }

    /**
     * Called when a member has been added - subclasses should compute grid cells for the object
     * and add the object to sorted index arrays
     *
     * @param   index   the index of the member that has just been added
     */
    protected abstract void memberAdded(final int index);

    /**
     * Removes a member object from the grid.
     *
     * @param   index   the index of the grid member to remove
     */
    public void remove(final int index) {

        if (this.objects.remove(index) == null) {
            LOG.warning("Attempt to remove from empty index");
        } else {
            memberRemoved(index);
        }
    }

    /**
     * Called when a member is about to be removed - subclasses should clean up actions performed
     * in <code>memberAdded</code>. The object still exists in the grid's object array when this
     * method is called.
     *
     * @param   index   the index of the member that is being removed
     */
    protected abstract void memberRemoved(final int index);

    /**
     * Updates the grid address for a grid member that has moved.
     *
     * @param   index   the index of the grid member that moved
     * @param   newX    the grid member's new X coordinate
     * @param   newY    the grid member's new Y coordinate
     */
    public abstract void move(final int index, final double newX, final double newY);

    /**
     * Converts a coordinate value to a grid address.
     *
     * @param    coord    the coordinate to convert
     * @param    min      the minimum coordinate value of the grid on the coordinate axis
     * @param    extent   the extent of the grid in the dimension of interest
     * @return   the address of the coordinate
     */
    protected int coordinateToAddress(final double coord, final double min, final int extent) {
```

```
        double addr;
        int result;

        addr = (coord - min) / this.gridCellSize;

        if (addr < 0) {
            result = 0;
        } else if (addr >= extent) {
            result = extent - 1;
        } else {
            result = (int) addr;
        }

        return result;
    }

    /**
     * Called when the capacity of the sparse array is changed, to allow other programs that need
     * to keep arrays of the same size to adjust their arrays.
     *
     * @param  array         the array whose capacity is changing
     * @param  newCapacity   the new capacity of the sparse array
     */
    public abstract void capacityChanged(SparseArray<?> array, int newCapacity);
}
```

```
package com.srbenoit.modeling.grid;

import java.awt.Color;
import java.util.logging.Level;
import com.srbenoit.geom.BasedVector2;

/**
 * A basic grid member based on <code>BasedVector2</code>.
 */
public class BasedVectorGridMember2 extends BasedVector2 implements GridMember2Int {

    /** the grid in which this member object is installed */
    private Grid2D grid;

    /** the index of the member in the grid */
    private int gridIndex;

    /** the number of neighbors this object currently has */
    private int numNeighbors;

    /** the objects that are neighbors of this object */
    private GridMember2Int[] neighbors;

    /**
     * Constructs a new <code>BasedVectorGridMember2</code>.
     *
     * @param  baseX   the X coordinate of the base point
     * @param  baseY   the Y coordinate of the base point
     * @param  vecX    the X component of the vector
     * @param  vecY    the Y component of the vector
     */
    public BasedVectorGridMember2(final double baseX, final double baseY, final double vecX,
        final double vecY) {

        super(baseX, baseY, vecX, vecY);

        this.grid = null;
        this.gridIndex = -1;

        this.numNeighbors = 0;
        this.neighbors = new GridMember2Int[4];
    }

    /**
     * Gets this grid this object is installed in.
     *
     * @return  the grid
     */
    public Grid2D getGrid() {

        return this.grid;
    }

    /**
     * Gets the index of the object in the grid where it is installed.
     *
     * @return  the grid index
     */
    public int getGridIndex() {
```

```
    return this.gridIndex;
}

/**
 * Gets the radius to use when drawing the grid member.
 *
 * @return  the radius (0 to draw as a point)
 */
public double getRadius() {

    return 0;
}

/**
 * Gets the color in which to render the element.
 *
 * @return  the color
 */
public Color getColor() {

    return Color.BLACK;
}

/**
 * Gets the fill color in which to render the element.
 *
 * @return  the color, or <code>null</code> for no fill
 */
public Color fillColor() {

    return null;
}

/**
 * Installs the object in a grid. If the object is a member of a grid when this method is
 * called, the object is first removed from that grid, then added to <code>theGrid</code>.
 *
 * @param  theGrid  the grid in which the object is being installed
 */
public void installInGrid(final Grid2D theGrid) {

    if (this.grid != null) {
        removeFromGrid();
    }

    if (theGrid == null) {
        this.gridIndex = -1;
    } else {
        this.gridIndex = theGrid.add(this);
    }

    this.grid = theGrid;
}

/**
 * Removes the object from the grid in which it is installed, if any.
 */
public void removeFromGrid() {

    if (this.grid == null) {
        LOG.warning("Attempt to remove member from grid that was not in a grid");
    } else {
        this.grid.remove(this.gridIndex);
        this.grid = null;
        this.gridIndex = -1;
    }
}

/**
 * Moves the object to a new position. If the object is installed in a grid, the grid is
 * notified so it can update the object's grid cell address.
 *
 * @param  xCoord  the new X coordinate
 * @param  yCoord  the new Y coordinate
 */
@Override public void setPos(final double xCoord, final double yCoord) {

    if (Double.isNaN(xCoord) || Double.isNaN(yCoord)) {
        LOG.log(Level.WARNING, "Attempt to set grid member position to NaN", new Exception());
        System.exit(1);
    }

    if (Double.isInfinite(xCoord) || Double.isInfinite(xCoord)) {
        LOG.log(Level.WARNING, "Attempt to set grid member position by Infinity",
            new Exception());
        System.exit(1);
    }
```

```
        super.setPos(xCoord, yCoord);

        if (this.grid != null) {
            this.grid.move(this.gridIndex, xCoord, yCoord);
        }
    }

    /**
     * Moves the object, by adjusting its position by a specified amount. If the object is
     * installed in a grid, the grid is notified so it can update the object's grid cell address.
     *
     * @param   deltaX   the change in X coordinate
     * @param   deltaY   the change in Y coordinate
     */
    @Override public void move(final double deltaX, final double deltaY) {

        if (Double.isNaN(deltaX) || Double.isNaN(deltaY)) {
            LOG.log(Level.WARNING, "Attempt to move grid member position by NaN", new Exception());
            System.exit(1);
        }

        if (Double.isInfinite(deltaX) || Double.isInfinite(deltaY)) {
            LOG.log(Level.WARNING, "Attempt to move grid member position by Infinity",
                new Exception());
            System.exit(1);
        }

        super.move(deltaX, deltaY);

        if (this.grid != null) {
            this.grid.move(this.gridIndex, getPosX(), getPosY());
        }
    }

    /**
     * Clears the list of this object's neighbors.
     */
    public void clearNeighbors() {

        this.numNeighbors = 0;
    }

    /**
     * Adds an object to the list of this object's neighbors.
     *
     * @param   neighbor   the neighbor to add
     */
    public void addNeighbor(final GridMember2Int neighbor) {

        GridMember2Int[] newList;

        if (this.numNeighbors == this.neighbors.length) {
            newList = new GridMember2Int[this.numNeighbors + 4];
            System.arraycopy(this.neighbors, 0, newList, 0, this.numNeighbors);
            this.neighbors = newList;
        }

        this.neighbors[this.numNeighbors] = neighbor;
        this.numNeighbors++;
    }

    /**
     * Gets the number of neighbors the member has.
     *
     * @return   the number of neighbors
     */
    public int getNumNeighbors() {

        return this.numNeighbors;
    }

    /**
     * Gets a neighbor grid member.
     *
     * @param    index   the index of the neighbor
     * @return   the neighbor
     */
    public GridMember2Int getNeighbor(final int index) {

        return this.neighbors[index];
    }
}

package com.srbenoit.modeling.grid;

import java.util.logging.Level;
```

```
import com.srbenoit.geom.Vector2;

/**
 * A base class for objects that may be contained in a 2-dimensional grid. Each object has an X and
 * Y position, a radius, and a grid address.
 */
public class DynamicGridMember2D extends PointGridMember2 {

    /** a type so we can search for neighbors of a single type, ignoring others */
    private final EnumElementType type;

    /** the radius of the object */
    private double radius;

    /** the ID of the structure this member belongs to */
    private final int structId;

    /** the object velocity (r') */
    private final Vector2 vel;

    /** the object acceleration (r'') */
    private final Vector2 accel;

    /** the force on the object */
    private final Vector2 force;

    /** the object mass */
    private double mass;

    /**
     * Constructs a new <code>GridMember2D</code>.
     *
     * @param  xCoord        the X coordinate
     * @param  yCoord        the Y coordinate
     * @param  rad           the radius of the object
     * @param  theType       the element type
     * @param  structureId   the ID of the structure this member belongs to
     * @param  theMass       the mass of the object
     */
    public DynamicGridMember2D(final double xCoord, final double yCoord, final double rad,
        final EnumElementType theType, final int structureId, final double theMass) {

        super(xCoord, yCoord);

        this.radius = rad;

        this.vel = new Vector2();
        this.accel = new Vector2();
        this.force = new Vector2();

        this.type = theType;
        this.structId = structureId;
        this.mass = theMass;
    }

    /**
     * Sets the radius of the object.
     *
     * @param  rad  the radius
     */
    public void setRadius(final double rad) {

        this.radius = rad;
    }

    /**
     * Gets the radius of the object.
     *
     * @return  the radius
     */
    @Override public double getRadius() {

        return this.radius;
    }

    /**
     * Gets the member type.
     *
     * @return  the type
     */
    public EnumElementType getType() {

        return this.type;
    }

    /**
     * Gets this structure ID of this object.
```

```
     *
     * @return  the structure ID
     */
    public int getStructId() {

        return this.structId;
    }

    /**
     * Updates the object velocity.
     *
     * @param  newX  the new X component of velocity
     * @param  newY  the new Y component of velocity
     */
    public void setVel(final double newX, final double newY) {

        if (Double.isNaN(newX) || Double.isNaN(newY)) {
            LOG.log(Level.WARNING, "Attempt to set grid member velocity to NaN", new Exception());
            System.exit(1);
        }

        if (Double.isInfinite(newX) || Double.isInfinite(newX)) {
            LOG.log(Level.WARNING, "Attempt to set grid member velocity by Infinity",
                new Exception());
            System.exit(1);
        }

        this.vel.setVec(newX, newY);
    }

    /**
     * Adjusts the object velocity by a specified amount.
     *
     * @param  deltaX  the change in X component of velocity
     * @param  deltaY  the change in Y component of velocity
     */
    public void addVel(final double deltaX, final double deltaY) {

        if (Double.isNaN(deltaX) || Double.isNaN(deltaY)) {
            LOG.log(Level.WARNING, "Attempt to move grid member velocity by NaN", new Exception());
            System.exit(1);
        }

        if (Double.isInfinite(deltaX) || Double.isInfinite(deltaY)) {
            LOG.log(Level.WARNING, "Attempt to move grid member velocity by Infinity",
                new Exception());
            System.exit(1);
        }

        this.vel.addVec(deltaX, deltaY);
    }

    /**
     * Gets the current X component of the object velocity.
     *
     * @return  the X component
     */
    public double getXVel() {

        return this.vel.getVecX();
    }

    /**
     * Gets the current Y component of the object velocity.
     *
     * @return  the Y component
     */
    public double getYVel() {

        return this.vel.getVecY();
    }

    /**
     * Gets the speed of the particle.
     *
     * @return  the speed
     */
    public double getSpeed() {

        return this.vel.length();
    }

    /**
     * Gets the acceleration of the particle.
     *
     * @return  the acceleration
     */
```

```
public double getAccel() {

    return this.accel.length();
}

/**
 * Gets the current X component of the object force.
 *
 * @return  the X component
 */
public double getXForce() {

    return this.force.getVecX();
}

/**
 * Gets the current Y component of the object force.
 *
 * @return  the Y component
 */
public double getYForce() {

    return this.force.getVecY();
}

/**
 * Gets the magnitude of the object force.
 *
 * @return  the magnitude of the force
 */
public double getForceMag() {

    return this.force.length();
}

/**
 * Sets the object force.
 *
 * @param  newX  the new X component
 * @param  newY  the new Y component
 */
public void setForce(final double newX, final double newY) {

    if (Double.isNaN(newX) || Double.isNaN(newY)) {
        LOG.log(Level.WARNING, "Attempt to set grid member force to NaN", new Exception());
        System.exit(1);
    }

    if (Double.isInfinite(newX) || Double.isInfinite(newY)) {
        LOG.log(Level.WARNING, "Attempt to set grid member force to Infinity",
            new Exception());
        System.exit(1);
    }

    this.force.setVec(newX, newY);
}

/**
 * Adds velocity components to the current force.
 *
 * @param  deltaX  the change in X component
 * @param  deltaY  the change in Y component
 */
public void addForce(final double deltaX, final double deltaY) {

    if (Double.isNaN(deltaX) || Double.isNaN(deltaY)) {
        LOG.log(Level.WARNING, "Attempt to adjust grid member force by NaN", new Exception());
        System.exit(1);
    }

    if (Double.isInfinite(deltaX) || Double.isInfinite(deltaY)) {
        LOG.log(Level.WARNING, "Attempt to adjust grid member force by Infinity",
            new Exception());
        System.exit(1);
    }

    this.force.addVec(deltaX, deltaY);
}

/**
 * Clears any existing forces and computes the forces on the element due to interactions with
 * outside elements. Subclasses should override. This does not include the internal forces of
 * the structure to which the element belongs, which should be done after this method is
 * called.
 */
public void interactionForce() {
```

```
        this.vel.setVec(0, 0);
    }

    /**
     * Performs the first step in Velocity Verlet integration. Here, we compute new positions based
     * on velocities and accelerations computed in the last step.
     *
     * <pre>
     *   x(t + dt) = x(t) + v(t) dt + (1/2) a(t) dt^2
     * </pre>
     *
     * @param  deltaT  the time step
     */
    public void predict(final double deltaT) {

        double dt2over2;
        double deltaX;
        double deltaY;

        dt2over2 = deltaT * deltaT / 2;

        deltaX = (deltaT * this.vel.getVecX()) + (dt2over2 * this.accel.getVecX());
        deltaY = (deltaT * this.vel.getVecY()) + (dt2over2 * this.accel.getVecY());

        move(deltaX, deltaY);
    }

    /**
     * Performs the second step in Velocity Verlet integration, to be done after forces are
     * computed.
     *
     * <pre>
     *   v(t + dt) = v(t) + (1/2)(a(t) + f/m) dt
     *   a(t + dt) = f/m
     * </pre>
     *
     * @param  deltaT  the time step
     */
    public void correct(final double deltaT) {

        double realAccelX;
        double realAccelY;

        realAccelX = this.force.getVecX() / this.mass;
        realAccelY = this.force.getVecY() / this.mass;

        // the 0.1 in the next should be 0.5 - testing damping
        addVel(0.1 * (this.accel.getVecX() + realAccelX) * deltaT,
            0.1 * (this.accel.getVecY() + realAccelY) * deltaT);

        this.accel.setVec(realAccelX, realAccelY);
    }
}
```

```
package com.srbenoit.modeling.grid;

/**
 * The possible element types in a simulation.
 */
public enum EnumElementType {

    /** a fixed element that cannot move */
    FIXED,

    /** an element that moves at a fixed rate */
    MOVING,

    /** a simple granule that interacts with Lennard-Jones interactions */
    LJ_GRANULE,

    /** an element within a cell or organelle membrane */
    MEMBRANE,

    /** an element of cortical actin cytoskeleton */
    ACTIN,

    /** a diffusing signal particle */
    SIGNAL,

    /** an ECM element */
    ECM,

    /** an element that exerts directional pressure */
    PRESSER;
}
```

```
package com.srbenoit.modeling.grid;
```

```
import com.srbenoit.math.grapher.Graphable;
import com.srbenoit.math.grapher.Grapher;

/**
 * A class that precomputes Lennard-Jones forces and energies at some number of discrete points
 * then returns an energy or force magnitude based on a square of the ratio of actual distance to
 * equilibrium distance. This class adjusts the forces and energies so they go smoothly to zero at
 * a distance of 2.5 * the equilibrium distance. For any distance ratio greater than 2.5, the class
 * returns zero energy and force.
 *
 * <p>u = (dist / EQ_DIST)^2
 *
 * <p>E = WELL_DEPTH [ (1 / u)^6 - (2 / u)^3 ]
 *
 * <p>EQ_DIST * |F| = 12 (WELL_DEPTH / SQRT(u)) [ (1 / u)^6 - (1 / u)^3 ] where force is directed
 * away from the particle (positive forces are repulsive).
 */
public class FastLennardJones implements Graphable {

    /** the number of discrete points to use in the representation of the potential */
    private final static int NUM_POINTS = 500;

    /** the default domain for graphing the function */
    private final double[] domain;

    /** the default range for graphing the function */
    private final double[] range;

    /** the precomputed energies */
    private final double[] energies;

    /** the precomputed forces */
    private final double[] forces;

    /**
     * Constructs a new <code>FastLennardJones</code>.
     *
     * @param  wellDepth  the potential well depth.
     */
    public FastLennardJones(final double wellDepth) {

        double uVal;
        double ratio;
        double square;
        double sixth;
        double twelfth;
        int last;

        this.energies = new double[NUM_POINTS];
        this.forces = new double[NUM_POINTS];

        // Compute energies
        for (int i = 0; i < NUM_POINTS; i++) {

            // if dist ranges from 0 to 2.5*EQ_DIST, u ranges from 0 to 6.25
            uVal = (i + 0.5) * 6.25 / NUM_POINTS;

            square = 1 / uVal;
            sixth = square * square * square;
            twelfth = sixth * sixth;

            this.energies[i] = wellDepth * (twelfth - (2 * sixth));
            this.forces[i] = 12 * wellDepth * (twelfth - sixth) / Math.sqrt(uVal);
        }

        // Now adjust so the last term is zero
        last = NUM_POINTS - 1;

        for (int i = 0; i < NUM_POINTS; i++) {

            this.energies[i] -= this.energies[last];
            this.forces[i] -= this.forces[last];
        }

        this.domain = new double[] { 0, 7 };
        this.range = new double[] { -3 * wellDepth, 5 * wellDepth };
    }

    /**
     * Computes the energy given a ratio of distance / equilibrium distance.
     *
     * @param   uVal  the square of the ratio of actual distance to equilibrium distance
     * @return  the energy
     */
    public double energy(final double uVal) {
```

```
        int index;

        index = (int) (uVal / 6.25 * NUM_POINTS);

        if (index < 0) {
            index = 0;
        } else if (index >= NUM_POINTS) {
            index = NUM_POINTS - 1;
        }

        return this.energies[index];
    }

    /**
     * Computes the force given a ratio of distance / equilibrium distance.
     *
     * @param   uVal  the square of the ratio of actual distance to equilibrium distance
     * @return  the force
     */
    public double forceTimesEqDist(final double uVal) {

        int index;

        index = (int) (uVal / 6.25 * NUM_POINTS);

        if (index < 0) {
            index = 0;
        } else if (index >= NUM_POINTS) {
            index = NUM_POINTS - 1;
        }

        return this.forces[index];
    }

    /**
     * Gets the number of dimensions of the graph (2 for a function of a single value).
     *
     * @return  the number of dimensions of the graph
     */
    public int graphDimensions() {

        return 2;
    }

    /**
     * Gets a default domain over which the graph will show the main features of the function. This
     * domain should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the left and right endpoints of the default domain
     */
    public double[] defaultDomain() {

        return this.domain.clone();
    }

    /**
     * Gets a default range over which the graph will show the main features of the function. This
     * range should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the lower and upper limits of the default range
     */
    public double[] defaultRange() {

        return this.range.clone();
    }

    /**
     * Computes the graph values at a coordinate or coordinates. The number of coordinates needed
     * is one less than the dimension.
     *
     * @param   coordinates  the list of coordinates
     * @return  the graph values at that location
     */
    public double valueAt(final double... coordinates) {

//          return energy(coordinates[0]);
        return forceTimesEqDist(coordinates[0]);
    }

    /**
     * Main method to graph the force vs. distance.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        FastLennardJones obj;
```

```
        Grapher grapher;

        obj = new FastLennardJones(1);

        grapher = new Grapher(500, 500);
        grapher.graph(obj);
        grapher.showInFrame("Fast Lennerd-Jones Force");
    }
}

package com.srbenoit.modeling.grid;

import com.srbenoit.math.grapher.Graphable;
import com.srbenoit.math.grapher.Grapher;

/**
 * A class that precomputes soft-sphere forces and energies at some number of discrete points then
 * returns an energy or force magnitude based on a square of the ratio of actual distance to
 * equilibrium distance. For any distance ratio greater than 1, the class returns zero energy and
 * force.
 *
 * <p>u = dist / EQ_DIST
 *
 * <p>E = WELL_DEPTH [ (1 / u)^12 - 1 ]
 *
 * <p>EQ_DIST * |F| = 12 (WELL_DEPTH / u) (1 / u)^12 where force is directed away from the particle
 * (positive forces are repulsive).
 */
public class FastSoftSphere implements Graphable {

    /** the number of discrete points to use in the representation of the potential */
    private final static int NUM_POINTS = 500;

    /** the default domain for graphing the function */
    private final double[] domain;

    /** the default range for graphing the function */
    private final double[] range;

    /** the precomputed energies */
    private final double[] energies;

    /** the precomputed forces */
    private final double[] forces;

    /**
     * Constructs a new <code>FastSoftSphere</code>.
     *
     * @param  wellDepth  the potential well depth.
     */
    public FastSoftSphere(final double wellDepth) {

        double uVal;
        double square;
        double sixth;
        double twelfth;
        int last;

        this.energies = new double[NUM_POINTS];
        this.forces = new double[NUM_POINTS];

        // Compute energies
        for (int i = 0; i < NUM_POINTS; i++) {

            // if dist ranges from 0 to EQ_DIST, u ranges from 0 to 1
            uVal = (i + 0.5) / NUM_POINTS;

            square = 1 / uVal;
            sixth = square * square * square;
            twelfth = sixth * sixth;

            this.energies[i] = wellDepth * (square - 1);
            this.forces[i] = 12 * wellDepth * square / uVal;
        }

        // Now adjust so the last term is zero
        last = NUM_POINTS - 1;
        for (int i = 0; i < NUM_POINTS; i++) {
            this.forces[i] -= this.forces[last];
        }

        this.domain = new double[] { 0, 1.5 };
        this.range = new double[] { -wellDepth, 30 * wellDepth };
    }

    /**
     * Computes the energy given a ratio of distance / equilibrium distance.
```

```
     *
     * @param   uVal  the square of the ratio of actual distance to equilibrium distance
     * @return  the energy
     */
    public double energy(final double uVal) {

        int index;

        index = (int) (uVal * NUM_POINTS);

        if (index < 0) {
            index = 0;
        } else if (index >= NUM_POINTS) {
            index = NUM_POINTS - 1;
        }

        return this.energies[index];
    }

    /**
     * Computes the force given a ratio of distance / equilibrium distance.
     *
     * @param   uVal  the square of the ratio of actual distance to equilibrium distance
     * @return  the force
     */
    public double forceTimesEqDist(final double uVal) {

        int index;

        index = (int) (uVal * NUM_POINTS);

        if (index < 0) {
            index = 0;
        } else if (index >= NUM_POINTS) {
            index = NUM_POINTS - 1;
        }

        return this.forces[index];
    }

    /**
     * Gets the number of dimensions of the graph (2 for a function of a single value).
     *
     * @return  the number of dimensions of the graph
     */
    public int graphDimensions() {

        return 2;
    }

    /**
     * Gets a default domain over which the graph will show the main features of the function. This
     * domain should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the left and right endpoints of the default domain
     */
    public double[] defaultDomain() {

        return this.domain.clone();
    }

    /**
     * Gets a default range over which the graph will show the main features of the function. This
     * range should be computed based on known attributes of the function.
     *
     * @return  a two-double array containing the lower and upper limits of the default range
     */
    public double[] defaultRange() {

        return this.range.clone();
    }

    /**
     * Computes the graph values at a coordinate or coordinates. The number of coordinates needed
     * is one less than the dimension.
     *
     * @param   coordinates  the list of coordinates
     * @return  the graph values at that location
     */
    public double valueAt(final double... coordinates) {

//  return energy(coordinates[0]);
        return forceTimesEqDist(coordinates[0]);
    }

    /**
     * Main method to graph the force vs. distance.
```

```
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        FastSoftSphere obj;
        Grapher grapher;

        obj = new FastSoftSphere(1);

        grapher = new Grapher(500, 500);
        grapher.graph(obj);
        grapher.showInFrame("Fast Soft Sphere");
    }
}

package com.srbenoit.modeling.grid;

import java.util.Random;
import javax.swing.JFrame;
import com.srbenoit.log.LoggedObject;

/**
 * Run a time evolution of a collection of granules.
 */
public class Granular2D extends LoggedObject {

    /** the number of frames to compute between renderings */
    public static final int FRAMES_PER_RENDER = 200;

    /** the number of frames to compute between exported frames (0 to skip exports) */
    public static final int FRAMES_PER_EXPORT = 500000; // must be nultiple of FRAMES_PER_RENDER

    /** the frame showing the evolution */
    private JFrame frame;

    /** the panel showing the evolution */
    private GridPanel2D panel;

    /** a grid used to accelerate neighbor testing */
    private final transient Grid2D grid;

    /**
     * Constructs a new <code>Granular2D</code>.
     *
     * @param  width   the number of granules to place in each row
     * @param  height  the number of rows of granules
     */
    public Granular2D(final int width, final int height) {

        Random rnd;
        LinkedListGridMember2D prior;
        LinkedListGridMember2D mem;
        Granule gran;
        double angle;

        rnd = new Random();

        // Spheres are radius 1, diameter 2.  We will place the array of spheres on a grid with
        // spacing 4 between centers, and a boundary of fixed spheres outside that.  The boundary
        // runs from 0 to 4 * (width + 1) in the X direction, and from 0 to 4 * (height + 1) in
        // the Y direction.  The interaction distance of spheres is 2.5 (r1+r2) = 5, so we use a
        // grid of cell size 6

        int xSize;
        int ySize;

        xSize = 4 * (width + 1);
        ySize = 4 * (height + 1);

        this.grid = new Grid2D(-2, -2, 6, (xSize + 6) / 6, (ySize + 6) / 6);

        // Build the fixed walls
        prior = new LinkedListGridMember2D(0, 0, 1, EnumElementType.FIXED, 0, 1);
        prior.installInGrid(this.grid);

        for (int x = 2; x < xSize; x += 2) {
            mem = new LinkedListGridMember2D(x, 0, 1, EnumElementType.FIXED, 0, 1);
            mem.addAfter(prior);
            mem.installInGrid(this.grid);
            prior = mem;
        }

        for (int y = 0; y < ySize; y += 2) {
            mem = new LinkedListGridMember2D(xSize, y, 1, EnumElementType.FIXED, 0, 1);
            mem.addAfter(prior);
            mem.installInGrid(this.grid);
```

```
            prior = mem;
        }

        for (int x = xSize; x > 0; x -= 2) {
            mem = new LinkedListGridMember2D(x, ySize, 1, EnumElementType.FIXED, 0, 1);
            mem.addAfter(prior);
            mem.installInGrid(this.grid);
            prior = mem;
        }

        for (int y = ySize; y > 0; y -= 2) {
            mem = new LinkedListGridMember2D(0, y, 1, EnumElementType.FIXED, 0, 1);
            mem.addAfter(prior);
            mem.installInGrid(this.grid);
            prior = mem;
        }

        // Build the granules
        for (int i = 0; i < width; i++) {

            for (int j = 0; j < height; j++) {
                gran = new Granule(4 + (i * 4), 4 + (j * 4), 1, 1, 1);
                angle = Math.PI * 2 * rnd.nextDouble();
                gran.setVel(Math.cos(angle), Math.sin(angle));
                gran.installInGrid(this.grid);
            }
        }

        buildFrame();
    }

    /**
     * Constructs the frame and panel
     */
    private void buildFrame() {

        this.frame = new JFrame("Granular Dynamics Simulation");
        this.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.panel = new GridPanel2D(600, 600, this.grid);
        this.frame.setContentPane(this.panel);
        this.frame.pack();
        this.frame.setVisible(true);
    }

    /**
     * Runs the evolution simulation.
     */
    public void evolve() {

        double maxMove;
        long frameCount;
        int frameNum;
        GridIterator2D iter;
        DynamicGridMember2D elem;
        long start;
        long end;
        double speed;
        double maxSpeed;
        double accel;
        double maxAccel;
        double force;
        double maxForce;
        double timestep;
        double time1;
        double time2;

        // Now to do the evolution
        iter = new GridIterator2D(this.grid);
        maxMove = 1.0 / 200;

        start = System.currentTimeMillis() - 1;
        frameCount = 0;
        frameNum = 1;

        while (this.frame.isVisible()) {

            // Since our max move (set above) is 1/20 of a grid cell size, we scan for neighbors in
            // a radius of 3 times our object's radius, then those neighbor relations are good for
            // at least 10 loops ("good" means we won't miss any neighbors; we may get extras)
            if ((frameCount % 20) == 0) {

                // recompuate neighbor relationships for all but fixed and moving elements
                iter.reset();

                while (iter.hasNext()) {
                    elem = (DynamicGridMember2D)iter.next();
```

```
                    if (elem.getType() != EnumElementType.FIXED) {
                        this.grid.getNeighborsOf(elem);
                    }
                }
            }

            // Scan for the maximum velocity and acceleration to choose a time step
            iter.reset();

            maxSpeed = 0;
            maxAccel = 0;

            while (iter.hasNext()) {
                elem = (DynamicGridMember2D)iter.next();
                speed = elem.getSpeed();
                accel = elem.getAccel();

                if (speed > maxSpeed) {
                    maxSpeed = speed;
                }

                if (accel > maxAccel) {
                    maxAccel = accel;
                }
            }

            if (maxSpeed == 0) {
                timestep = 1e-10;
            } else {
                time1 = maxMove / maxSpeed;
                time2 = Math.sqrt(2 * maxMove / maxAccel);
                timestep = (time1 < time2) ? time1 : time2;
            }

// LOG.warning("Max speed = " + maxSpeed + ", time step: " + timestep);

            // Run the predictor step in the Gear algorithm
            iter.reset();

            while (iter.hasNext()) {
                elem = (DynamicGridMember2D)iter.next();
                elem.predict(timestep);
            }

            // Compute all element  forces
            iter.reset();

            while (iter.hasNext()) {
                elem = (DynamicGridMember2D)iter.next();
                elem.interactionForce();
            }

            // Diagnostic output: print the largest force computed
            iter.reset();

            maxForce = 0;

            while (iter.hasNext()) {
                elem = (DynamicGridMember2D)iter.next();
                force = elem.getForceMag();

                if (force > maxForce) {
                    maxForce = force;
                }
            }

// LOG.info("LARGEST FORCE: " + maxForce);

            // Run the corrector step in the Gear algorithm
            iter.reset();

            while (iter.hasNext()) {
                elem = (DynamicGridMember2D)iter.next();
                elem.correct(timestep);
            }

            if ((frameCount % FRAMES_PER_RENDER) == 0) {
                end = System.currentTimeMillis();
                this.panel.update(frameCount * 1000.0 / (end - start));

                if ((FRAMES_PER_EXPORT != 0) && ((frameCount % FRAMES_PER_EXPORT) == 0)) {
                    this.panel.exportFrame(frameNum);
                    frameNum++;
                }
            }

            frameCount++;
```

```
        }
    }

    /**
     * Main method to run the test.
     *
     * @param  args  Command-line arguments.
     */
    public static void main(final String[] args) {

        new Granular2D(20, 20).evolve();
    }
}

package com.srbenoit.modeling.grid;

import java.awt.Color;

/**
 * A granule that interacts with Lennard-Jones potential.
 */
public class Granule extends DynamicGridMember2D {

    /** well depth for Lennard-Jones and soft-sphere interactions */
    private static final double WELL_DEPTH = 0.1;

    /** a class to compute Lennard-Jones forces quickly */
    private static final FastLennardJones LJFORCE;

    /** a class to compute soft sphere forces quickly */
    private static final FastSoftSphere SSFORCE;

    /** viscosity-based damping force */
    private static final double VISC = 0.1;

    static {
        LJFORCE = new FastLennardJones(WELL_DEPTH);
        SSFORCE = new FastSoftSphere(WELL_DEPTH);
    }

    /**
     * Constructs a new <code>Granule</code>
     *
     * @param  xCoord     the X coordinate
     * @param  yCoord     the Y coordinate
     * @param  rad        the radius of the object
     * @param  structId   the ID of the structure (cell) this membrane belongs to
     * @param  mass       the mass of the granule
     */
    public Granule(final double xCoord, final double yCoord, final double rad, final int structId,
        final double mass) {

        super(xCoord, yCoord, rad, EnumElementType.LJ_GRANULE, structId, mass);
    }

    /**
     * Gets the color in which to render the element.
     *
     * @return  the color
     */
    @Override public Color getColor() {

        return Color.BLUE;
    }

    /**
     * Computes the forces on the element.
     */
    @Override public void interactionForce() {

        int count;
        EnumElementType type;
        DynamicGridMember2D near;
        double scale;
        double eps;
        double epsSq;
        double distX;
        double distY;
        double distSq;
        double frcX;
        double frcY;

        frcX = 0;
        frcY = 0;

        count = getNumNeighbors();
```

```
            for (int i = 0; i < count; i++) {
                near = (DynamicGridMember2D) getNeighbor(i);
                type = near.getType();

                if (type == EnumElementType.LJ_GRANULE) {

                    eps = getRadius() + near.getRadius();
                    epsSq = eps * eps;
                    distX = near.getPosX() - getPosX();
                    distY = near.getPosY() - getPosY();
                    distSq = (distX * distX) + (distY * distY);
                    scale = -LJFORCE.forceTimesEqDist(distSq / epsSq) / eps;

                    frcX += distX * scale;
                    frcY += distY * scale;
                } else if (type == EnumElementType.FIXED) {

                    eps = getRadius() + near.getRadius();
                    epsSq = eps * eps;
                    distX = near.getPosX() - getPosX();
                    distY = near.getPosY() - getPosY();
                    distSq = (distX * distX) + (distY * distY);
                    scale = -SSFORCE.forceTimesEqDist(distSq / epsSq) / eps;

                    frcX += distX * scale;
                    frcY += distY * scale;
                }
            }

            // Damping
            frcX -= VISC * this.getXVel();
            frcY -= VISC * this.getYVel();

            setForce(frcX, frcY);
        }
    }

    package com.srbenoit.modeling.grid;

    import java.util.logging.Level;
    import com.srbenoit.sparsearray.SparseArray;

    /**
     * A two-dimensional grid that may contain objects. Contained objects must be derived from <code>
     * GridMember</code>. This class provides a fast way to iterate over the set of potential neighbors
     * of an object.
     *
     * <p>The grid manages a sparse array data structure that assigns each added object an array index
     * that does not change while the object is part of the grid. Removing objects may open gaps in the
     * array, which newly added objects will fill first before expanding the array. There are three
     * parallel arrays, all the same length, and each with the same indices populated with data:
     *
     * <dl>
     * <dt><strong>objects</strong></dt>
     * <dd>the sparse array of <code>GridMember</code> objects</dd>
     *
     * <dt><strong>xAddresses</strong></dt>
     * <dd>the X address (from 0 to gridWidth - 1) of each object</dd>
     *
     * <dt><strong>yAddresses</strong></dt>
     * <dd>the Y address (from 0 to gridHeight - 1) of each object</dd>
     * </dl>
     *
     * <p>The <strong>sortedIndices</strong> array supports fast access to neighbors lists for a member
     * as follows. Each element in this array is an array of integers containing the indices of all
     * objects with a particular X address. Within these inner arrays, indices are organized in blocks
     * by Y address (all indices with Y address 0 first, all indices with Y address 1 next, and so
     * forth). The starting index of each block is stored in the two-dimensional array <strong>
     * sortedIndexStarts</strong>, which is gridWidth x gridHeight in size, and the index after the
     * ending index of each block is stored in <strong>sortedIndexEnds</strong> (meaning that if the
     * values in <strong>sortedIndexStarts</strong> and <strong>sortedIndexEnds</strong> are equal,
     * there are no items in the list for that Y address).
     */
    public final class Grid2D extends AbstractGrid2D {

        /** the X coordinate of the left edge of the grid */
        private final double minX;

        /** the Y coordinate of the bottom edge of the grid */
        private final double minY;

        /** the X coordinate of the right edge of the grid */
        private final double maxX;

        /** the Y coordinate of the top edge of the grid */
        private final double maxY;
```

```
/** the width of the grid as a number of grid cells */
private final int gridWidth;

/** the height of the grid as a number of grid cells */
private final int gridHeight;

/** the X address of each object in the grid (size = objects.length) */
private int[] xAddresses;

/** the Y address of each object in the grid (size = objects.length) */
private int[] yAddresses;

/**
 * an array of lists of object indices, where [0] holds the list of indices of all objects with
 * X address 0, [1] holds the list of indices of all objects with X address [1], and so forth
 * (the length of this array is gridWidth). Within each list, indices of all objects with Y
 * address 0 appear first, then all with Y address 1, and so forth.
 */
private int[][] sortedIndices;

/**
 * the starting index of each block in sortedIndices (this array is gridWidth x gridHeight, and
 * the element at [i][j] is the index into the sortedIndices[i] array of the first object whose
 * Y address is j)
 */
private final int[][] sortedIndexStarts;

/**
 * the ending index of each block in sortedIndices (this array is gridWidth x gridHeightm, and
 * the element at [i][j] is the index into the sortedIndices[i] array after the last object
 * whose Y address is j - may be equal to array length)
 */
private final int[][] sortedIndexEnds;

/**
 * Constructs a new <code>AbstractGrid</code>.
 *
 * @param  minXCoord  the X coordinate of the left edge of the grid
 * @param  minYCoord  the Y coordinate of the bottom edge of the grid
 * @param  cellSize   width and height of the grid cells
 * @param  width      the width of the grid in cells
 * @param  height     the height of the grid in cells
 */
public Grid2D(final double minXCoord, final double minYCoord, final double cellSize,
    final int width, final int height) {

    super(cellSize);

    this.minX = minXCoord;
    this.minY = minYCoord;
    this.maxX = minXCoord + (width * cellSize);
    this.maxY = minYCoord + (height * cellSize);

    this.gridWidth = width;
    this.gridHeight = height;

    this.xAddresses = new int[BLOCK_SIZE];
    this.yAddresses = new int[BLOCK_SIZE];

    this.sortedIndices = new int[width][BLOCK_SIZE / 4];
    this.sortedIndexStarts = new int[width][height];
    this.sortedIndexEnds = new int[width][height];
}

/**
 * Gets the X coordinate of the left edge of the grid.
 *
 * @return  the minimum X coordinate
 */
public double getMinX() {

    return this.minX;
}

/**
 * Gets the X coordinate of the left edge of the grid.
 *
 * @return  the minimum X coordinate
 */
public double getMaxX() {

    return this.maxX;
}

/**
 * Gets the Y coordinate of the bottom edge of the grid.
 *
```

```
     * @return  the minimum Y coordinate
     */
    public double getMinY() {

        return this.minY;
    }

    /**
     * Gets the Y coordinate of the bottom edge of the grid.
     *
     * @return  the minimum Y coordinate
     */
    public double getMaxY() {

        return this.maxY;
    }

    /**
     * Gets the width of the grid as a number of cells.
     *
     * @return  the width of the grid
     */
    public int getGridWidth() {

        return this.gridWidth;
    }

    /**
     * Gets the height of the grid as a number of cells.
     *
     * @return  the height of the grid
     */
    public int getGridHeight() {

        return this.gridHeight;
    }

    /**
     * Called when a member has been added - subclasses should compute grid cells for the object
     * and add the object to sorted index arrays
     *
     * @param  index  the index of the member that has just been added
     */
    protected void memberAdded(final int index) {

        GridMember2Int member;
        int xAddr;
        int yAddr;

        member = get(index);

        xAddr = coordinateToAddress(member.getPosX(), this.minX, this.gridWidth);
        yAddr = coordinateToAddress(member.getPosY(), this.minY, this.gridHeight);
        this.xAddresses[index] = xAddr;
        this.yAddresses[index] = yAddr;

        // Update the sortedIndices arrays to include the new object
        addSortedIndex(index);
    }

    /**
     * Gets the X address of the element at a particular index.
     *
     * @param   index  the index of the element
     * @return  the X address of that element
     */
    public int getXAddress(final int index) {

        return this.xAddresses[index];
    }

    /**
     * Gets the Y address of the element at a particular index.
     *
     * @param   index  the index of the element
     * @return  the Y address of that element
     */
    public int getYAddress(final int index) {

        return this.yAddresses[index];
    }

    /**
     * Called when a member is about to be removed - subclasses should clean up actions performed
     * in <code>memberAdded</code>. The object still exists in the grid's object array when this
     * method is called.
     *
```

```
     * @param  index  the index of the member that is being removed
     */
    protected void memberRemoved(final int index) {

        removeSortedIndex(index);
    }

    /**
     * Updates the grid address for a grid member that has moved.
     *
     * @param  index  the index of the grid member that moved
     * @param  newX   the grid member's new X coordinate
     * @param  newY   the grid member's new Y coordinate
     */
    public void move(final int index, final double newX, final double newY) {

        int xAddr;
        int yAddr;

        xAddr = coordinateToAddress(newX, this.minX, this.gridWidth);
        yAddr = coordinateToAddress(newY, this.minY, this.gridHeight);

        // Only take action if addresses have changed
        if ((xAddr != this.xAddresses[index]) || (yAddr != this.yAddresses[index])) {
            removeSortedIndex(index);
            this.xAddresses[index] = xAddr;
            this.yAddresses[index] = yAddr;
            addSortedIndex(index);
        }
    }

    /**
     * Identifies the neighboring objects to a specified object, and sets the bits corresponding to
     * those potential neighbors in the <code>temp</code> bit set.
     *
     * @param  member  the grid member whose neighbors are requested
     * @param  range   the range within which we consider points to be neighbors (must be no more
     *                 than one grid cell size)
     */
    public void getNeighborsOf(final GridMember2Int member, final double range) {

        int index;
        double targetX;
        double targetY;
        double targetMinX;
        double targetMaxX;
        double targetMinY;
        double targetMaxY;
        int xAddr;
        int yAddr;
        GridMember2Int test;
        int testIndex;
        int[] starts;
        int[] ends;
        int[] array;
        int end;
        int xPos;
        int yPos;
        int which;

        member.clearNeighbors();

        index = member.getGridIndex();
        targetX = member.getPosX();
        targetY = member.getPosY();

        targetMinX = targetX - range;
        targetMaxX = targetX + range;
        targetMinY = targetY - range;
        targetMaxY = targetY + range;

        xAddr = this.xAddresses[index];
        yAddr = this.yAddresses[index];

        // Test the column to the left of the object's cell
        xPos = xAddr - 1;

        if (xPos >= 0) {
            array = this.sortedIndices[xPos];
            starts = this.sortedIndexStarts[xPos];
            ends = this.sortedIndexEnds[xPos];

            // Test the cell below and left of the object's cell
            yPos = yAddr - 1;

            if (yPos >= 0) {
                end = ends[yPos];
```

```
            for (which = starts[yPos]; which < end; which++) {
                testIndex = array[which];
                test = get(testIndex);

                if ((test.getPosX() > targetMinX) && (test.getPosY() > targetMinY)) { // NOPMD SRB
                    member.addNeighbor(test);
                }
            }
        }

        // Test the cell to the left of the object's cell
        end = ends[yAddr];

        for (which = starts[yAddr]; which < end; which++) {
            testIndex = array[which];
            test = get(testIndex);

            if (test.getPosX() > targetMinX) {
                member.addNeighbor(test);
            }
        }

        // Test the cell above and to the left of the object's cell
        yPos = yAddr + 1;

        if (yPos < this.gridHeight) {
            end = ends[yPos];

            for (which = starts[yPos]; which < end; which++) {
                testIndex = array[which];
                test = get(testIndex);

                if ((test.getPosX() > targetMinX) && (test.getPosY() < targetMaxY)) { // NOPMD SRB
                    member.addNeighbor(test);
                }
            }
        }
    }

    // Test the column containing the object's cell
    array = this.sortedIndices[xAddr];
    starts = this.sortedIndexStarts[xAddr];
    ends = this.sortedIndexEnds[xAddr];

    // Test the cell below the object's cell
    yPos = yAddr - 1;

    if (yPos >= 0) {
        end = ends[yPos];

        for (which = starts[yPos]; which < end; which++) {
            testIndex = array[which];
            test = get(testIndex);

            if (test.getPosY() > targetMinY) {
                member.addNeighbor(test);
            }
        }
    }

    // Test the object's cell
    end = ends[yAddr];

    for (which = starts[yAddr]; which < end; which++) {
        testIndex = array[which];

        if (testIndex != index) {
            test = get(testIndex);
            member.addNeighbor(test);
        }
    }

    // Test the cell above the object's cell
    yPos = yAddr + 1;

    if (yPos < this.gridHeight) {
        end = ends[yPos];

        for (which = starts[yPos]; which < end; which++) {
            testIndex = array[which];
            test = get(testIndex);

            if (test.getPosY() < targetMaxY) {
                member.addNeighbor(test);
            }
        }
```

```
        }

        // Test the column to the right of the object's cell
        xPos = xAddr + 1;

        if (xPos < this.gridWidth) {
            array = this.sortedIndices[xPos];
            starts = this.sortedIndexStarts[xPos];
            ends = this.sortedIndexEnds[xPos];

            // Test the cell below and right of the object's cell
            yPos = yAddr - 1;

            if (yPos >= 0) {
                end = ends[yPos];

                for (which = starts[yPos]; which < end; which++) {
                    testIndex = array[which];
                    test = get(testIndex);

                    if ((test.getPosX() < targetMaxX) && (test.getPosY() > targetMinY)) { // NOPMD SRB
                        member.addNeighbor(test);
                    }
                }
            }

            // Test the cell to the right of the object's cell
            end = ends[yAddr];

            for (which = starts[yAddr]; which < end; which++) {
                testIndex = array[which];
                test = get(testIndex);

                if (test.getPosX() < targetMaxX) {
                    member.addNeighbor(test);
                }
            }

            // Test the cell above and to the right of the object's cell
            yPos = yAddr + 1;

            if (yPos < this.gridHeight) {
                end = ends[yPos];

                for (which = starts[yPos]; which < end; which++) {
                    testIndex = array[which];
                    test = get(testIndex);

                    if ((test.getPosX() < targetMaxX) && (test.getPosY() < targetMaxY)) { // NOPMD SRB
                        member.addNeighbor(test);
                    }
                }
            }
        }
    }

    /**
     * Identifies the neighboring objects to a specified object, and sets the bits corresponding to
     * those potential neighbors in the <code>temp</code> bit set.
     *
     * @param  member  the grid member whose neighbors are requested
     */
    public void getNeighborsOf(final GridMember2Int member) {

        int index;
        int xAddr;
        int yAddr;
        int testIndex;
        int[] starts;
        int[] ends;
        int[] array;
        int end;
        int xPos;
        int yPos;
        int which;
        GridMember2Int test;

        member.clearNeighbors();

        index = member.getGridIndex();

        xAddr = this.xAddresses[index];
        yAddr = this.yAddresses[index];

        // Test the column to the left of the object's cell
        xPos = xAddr - 1;
```

```
if (xPos >= 0) {
    array = this.sortedIndices[xPos];
    starts = this.sortedIndexStarts[xPos];
    ends = this.sortedIndexEnds[xPos];

    // Test the cell below and left of the object's cell
    yPos = yAddr - 1;

    if (yPos >= 0) {
        end = ends[yPos];

        for (which = starts[yPos]; which < end; which++) {
            testIndex = array[which];
            test = get(testIndex);
            member.addNeighbor(test);
        }
    }

    // Test the cell to the left of the object's cell
    end = ends[yAddr];

    for (which = starts[yAddr]; which < end; which++) {
        testIndex = array[which];
        test = get(testIndex);
        member.addNeighbor(test);
    }

    // Test the cell above and to the left of the object's cell
    yPos = yAddr + 1;

    if (yPos < this.gridHeight) {
        end = ends[yPos];

        for (which = starts[yPos]; which < end; which++) {
            testIndex = array[which];
            test = get(testIndex);
            member.addNeighbor(test);
        }
    }
}

// Test the column containing the object's cell
array = this.sortedIndices[xAddr];
starts = this.sortedIndexStarts[xAddr];
ends = this.sortedIndexEnds[xAddr];

// Test the cell below the object's cell
yPos = yAddr - 1;

if (yPos >= 0) {
    end = ends[yPos];

    for (which = starts[yPos]; which < end; which++) {
        testIndex = array[which];
        test = get(testIndex);
        member.addNeighbor(test);
    }
}

// Test the object's cell
end = ends[yAddr];

for (which = starts[yAddr]; which < end; which++) {
    testIndex = array[which];

    if (testIndex != index) {
        test = get(testIndex);
        member.addNeighbor(test);
    }
}

// Test the cell above the object's cell
yPos = yAddr + 1;

if (yPos < this.gridHeight) {
    end = ends[yPos];

    for (which = starts[yPos]; which < end; which++) {
        testIndex = array[which];
        test = get(testIndex);
        member.addNeighbor(test);
    }
}

// Test the column to the right of the object's cell
xPos = xAddr + 1;
```

```
        if (xPos < this.gridWidth) {
            array = this.sortedIndices[xPos];
            starts = this.sortedIndexStarts[xPos];
            ends = this.sortedIndexEnds[xPos];

            // Test the cell below and right of the object's cell
            yPos = yAddr - 1;

            if (yPos >= 0) {
                end = ends[yPos];

                for (which = starts[yPos]; which < end; which++) {
                    testIndex = array[which];
                    test = get(testIndex);
                    member.addNeighbor(test);
                }
            }

            // Test the cell to the right of the object's cell
            end = ends[yAddr];

            for (which = starts[yAddr]; which < end; which++) {
                testIndex = array[which];
                test = get(testIndex);
                member.addNeighbor(test);
            }

            // Test the cell above and to the right of the object's cell
            yPos = yAddr + 1;

            if (yPos < this.gridHeight) {
                end = ends[yPos];

                for (which = starts[yPos]; which < end; which++) {
                    testIndex = array[which];
                    test = get(testIndex);
                    member.addNeighbor(test);
                }
            }
        }
    }

    /**
     * Called when the capacity of the sparse array is changed, to allow other programs that need
     * to keep arrays of the same size to adjust their arrays.
     *
     * @param  array         the array whose capacity is changing
     * @param  newCapacity   the new capacity of the sparse array
     */
    public void capacityChanged(final SparseArray<?> array, final int newCapacity) {

        int len;
        int[] newXAddr;
        int[] newYAddr;

        len = this.xAddresses.length;

        newXAddr = new int[newCapacity];
        newYAddr = new int[newCapacity];

        System.arraycopy(this.xAddresses, 0, newXAddr, 0, len);
        System.arraycopy(this.yAddresses, 0, newYAddr, 0, len);

        this.xAddresses = newXAddr;
        this.yAddresses = newYAddr;
    }

    /**
     * Adds a member index to the set of sorted indices. The object, X address, and Y address
     * should be populated before calling this method.
     *
     * @param  index  the index to add
     */
    private void addSortedIndex(final int index) {

        int xAddr;
        int yAddr;
        int start;
        int end;
        int[] array;
        int nextStart;
        int priorEnd;
        int[] newArray;
        int size;
        int pos;

        xAddr = this.xAddresses[index];
```

```
        yAddr = this.yAddresses[index];

        // Get array and current start and end of the block within the array
        array = this.sortedIndices[xAddr];
        start = this.sortedIndexStarts[xAddr][yAddr];
        end = this.sortedIndexEnds[xAddr][yAddr];

        for (int test = start; test < end; test++) {

            if (array[test] == index) {
                LOG.log(Level.WARNING, "Request to add object to sorted list it is already in",
                        new Exception());

                return;
            }
        }

        // Find the start of the next block
        if (yAddr < (this.gridHeight - 1)) {
            nextStart = this.sortedIndexStarts[xAddr][yAddr + 1];
        } else {
            nextStart = array.length;
        }

        // Find the end of the prior block
        if (yAddr > 0) {
            priorEnd = this.sortedIndexEnds[xAddr][yAddr - 1];
        } else {
            priorEnd = 0;
        }

        // If there is room after the current block, insert the element at the end of the current
        // block and advance its end position.  Otherwise, if there is room before the block,
        // insert the element before the block and move its start position back one.  If neither
        // can be done, expand storage to make room at the end of the block then add the element.

        if (nextStart > end) {
            array[end] = index;
            this.sortedIndexEnds[xAddr][yAddr] = end + 1;
        } else if (priorEnd < start) {
            array[start - 1] = index;
            this.sortedIndexStarts[xAddr][yAddr] = start - 1;
        } else {

            // If we're going to reallocate, size the array so there are four empty slots at the
            // end of each block (this keeps arrays from growing too large while ensuring we only
            // do this at most every fourth add)
            size = 0;

            for (int y = 0; y < this.gridHeight; y++) {
                size += this.sortedIndexEnds[xAddr][y] - this.sortedIndexStarts[xAddr][y] + 4;
            }

            newArray = new int[size];

            // Copy the array data into the new array
            pos = 0;

            for (int y = 0; y < this.gridHeight; y++) {
                size = this.sortedIndexEnds[xAddr][y] - this.sortedIndexStarts[xAddr][y];

                if (size > 0) {
                    System.arraycopy(array, this.sortedIndexStarts[xAddr][y], newArray, pos, size);
                }

                this.sortedIndexStarts[xAddr][y] = pos;
                this.sortedIndexEnds[xAddr][y] = pos + size;
                pos += size + 4;
            }

            this.sortedIndices[xAddr] = newArray;

            // Now there's room to add the new element
            end = this.sortedIndexEnds[xAddr][yAddr];
            newArray[end] = index;
            this.sortedIndexEnds[xAddr][yAddr] = end + 1;
        }
    }

    /**
     * Removes a member index from the set of sorted indices.
     *
     * @param  index  the index to remove
     */
    private void removeSortedIndex(final int index) {

        int xAddr;
```

```
            int yAddr;
            int start;
            int end;
            int[] array;
            int pos;

            xAddr = this.xAddresses[index];
            yAddr = this.yAddresses[index];

            array = this.sortedIndices[xAddr];
            start = this.sortedIndexStarts[xAddr][yAddr];
            end = this.sortedIndexEnds[xAddr][yAddr];

            for (pos = start; pos < end; pos++) {

                if (array[pos] == index) {
                    array[pos] = -1;

                    break;
                }
            }

            if (pos == start) {

                // If it was the first item, just shift the start ahead one
                this.sortedIndexStarts[xAddr][yAddr] = start + 1;
            } else if (pos == (end - 1)) {

                // If it was the last item, just shift the end back by one
                this.sortedIndexEnds[xAddr][yAddr] = end - 1;
            } else if (pos < (end - 1)) {

                // If it was in the middle, shift the end to the position we found,
                // then shift the end back by one
                array[pos] = array[end - 1];
                this.sortedIndexEnds[xAddr][yAddr] = end - 1;
            } else {
                LOG.warning("index not found while removing from sorted arrays");
            }
        }
    }

    package com.srbenoit.modeling.grid;

    import java.util.Iterator;
    import java.util.NoSuchElementException;

    /**
     * An iterator that will iterate over a grid's elements.
     */
    public final class GridIterator2D implements Iterator<GridMember2Int> {

        /** the source grid being iterated */
        private final Grid2D source;

        /** the index of the current element, -1 if finished or not yet started */
        private int currentElement;

        /** the index of the next element to be returned, -1 if finished */
        private int nextElement;

        /**
         * Constructs a new <code>GridIterator</code>.
         *
         * @param  sourceGrid  the source array being iterated
         */
        public GridIterator2D(final Grid2D sourceGrid) {

            this.source = sourceGrid;
            this.nextElement = sourceGrid.getNextFilled(0);
            this.currentElement = -1;
        }

        /**
         * Resets the iterator to iterate over the grid again. This puts the iterator into the same
         * state it would be in if it had just been created. This saves object creations when iterating
         * a grid multiple times.
         */
        public void reset() {

            this.nextElement = this.source.getNextFilled(0);
            this.currentElement = -1;
        }

        /**
         * Returns <code>true</code> if the iteration has more elements. In other words, returns <code>
         * true</code> if <code>next</code> would return an element rather than throwing an exception.
```

```
     *
     * @return  <code>true</code> if the iteration has more elements; <code>false</code> otherwise
     */
    @Override public boolean hasNext() {

        return this.nextElement != -1;
    }

    /**
     * Returns the next element in the iteration.
     *
     * @return  the next element in the iteration
     * @throws  NoSuchElementException  if the iteration has no more elements
     */
    @Override public GridMember2Int next() throws NoSuchElementException {

        GridMember2Int result;

        result = this.source.get(this.nextElement);
        this.currentElement = this.nextElement;
        this.nextElement = this.source.getNextFilled(this.currentElement + 1);

        return result;
    }

    /**
     * Removes from the underlying collection the last element returned by the iterator (optional
     * operation). This method can be called only once per call to <code>next</code>.
     *
     * <p>The behavior of an iterator is unspecified if the underlying collection is modified while
     * the iteration is in progress in any way other than by calling this method.
     *
     * @throws  IllegalStateException  if the <code>next</code> method has not yet been called, or
     *                                 the <code>remove</code> method has already been called after
     *                                 the last call to the <code>next</code> method
     */
    @Override public void remove() throws IllegalStateException {

        if (this.currentElement == -1) {
            throw new IllegalStateException();
        }

        this.source.remove(this.currentElement);
        this.currentElement = -1;
    }
}

package com.srbenoit.modeling.grid;

import java.awt.Color;
import com.srbenoit.geom.Point2Int;

/**
 * A base class for objects that may be contained in a 2-dimensional grid. Each object has an X and
 * Y position, a radius, and a grid address.
 */
public interface GridMember2Int extends Point2Int {

    /**
     * Gets this grid this object is installed in.
     *
     * @return  the grid
     */
    Grid2D getGrid();

    /**
     * Gets the index of the object in the grid where it is installed.
     *
     * @return  the grid index
     */
    int getGridIndex();

    /**
     * Gets the radius to use when drawing the grid member.
     *
     * @return  the radius (0 to draw as a point)
     */
    double getRadius();

    /**
     * Gets the color in which to render the element.
     *
     * @return  the color
     */
    Color getColor();

    /**
```

```
     * Gets the fill color in which to render the element.
     *
     * @return  the color, or <code>null</code> for no fill
     */
    Color fillColor();

    /**
     * Installs the object in a grid. If the object is a member of a grid when this method is
     * called, the object is first removed from that grid, then added to <code>theGrid</code>.
     *
     * @param  theGrid  the grid in which the object is being installed
     */
    void installInGrid(Grid2D theGrid);

    /**
     * Removes the object from the grid in which it is installed, if any.
     */
    void removeFromGrid();

    /**
     * Clears the list of this object's neighbors.
     */
    void clearNeighbors();

    /**
     * Adds an object to the list of this object's neighbors.
     *
     * @param  neighbor  the neighbor to add
     */
    void addNeighbor(GridMember2Int neighbor);

    /**
     * Gets the number of neighbors the member has.
     *
     * @return  the number of neighbors
     */
    int getNumNeighbors();

    /**
     * Gets a neighbor grid member.
     *
     * @param   index  the index of the neighbor
     * @return  the neighbor
     */
    GridMember2Int getNeighbor(int index);
}

package com.srbenoit.modeling.grid;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.Stroke;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.geom.Ellipse2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import javax.imageio.ImageWriter;
import javax.imageio.stream.ImageOutputStream;
import com.srbenoit.log.LoggedPanel;
import com.srbenoit.render.LineSegment2D;

/**
 * A panel that renders the state of a grid.
 */
public class GridPanel2D extends LoggedPanel implements MouseListener, MouseMotionListener {

    /** version number for serialization */
    private static final long serialVersionUID = 1940457897064190922L;

    /** flag to control whether grid is drawn */
    private static final boolean DRAW_GRID = false;

    /** flag to control whether circles are drawn on linked list elements */
    private static final boolean CIRCLES_ON_LINKED = false;

    /** flag to control whether antialiasing is performed */
    private static final boolean ANTIALIAS = false;
```

```
/** the grid this panel renders */
private final Grid2D grid;

/** an iterator that traverses the grid's member objects */
private final GridIterator2D iter;

/** the pixels per grid cell */
private int pixPerCell;

/** the offscreen image */
private final transient BufferedImage offscreen;

/** the <code>Graphics2D</code> object for the offscreen image */
private final transient Graphics2D graphics;

/** a stroke to use to render lines */
private final transient BasicStroke stroke;

/** the grid member being dragged */
private GridMember2Int dragging;

/** the scale at which to draw force vectors (0 for no vectors) */
private double vectorScale;

/** the color in which to draw the grid lines */
private final Color gridColor;

/**
 * Constructs a new <code>GridPanel</code>.
 *
 * @param  width     the width of the window
 * @param  height    the height of the window
 * @param  theGrid   the grid this panel will render
 */
public GridPanel2D(final int width, final int height, final Grid2D theGrid) {

    super();

    int pixPerCellX;
    int pixPerCellY;
    int actWidth;
    int actHeight;

    this.grid = theGrid;
    this.iter = new GridIterator2D(this.grid);
    this.dragging = null;
    this.vectorScale = 0;

    // Determine the axis with the largest number of blocks
    pixPerCellX = width / theGrid.getGridWidth();
    pixPerCellY = height / theGrid.getGridHeight();

    // Make the grid cells square - the smaller pixel size from above
    this.pixPerCell = (pixPerCellX < pixPerCellY) ? pixPerCellX : pixPerCellY;

    if (this.pixPerCell == 0) {
        this.pixPerCell = 1;
    }

    actWidth = (theGrid.getGridWidth() * this.pixPerCell) + 1;
    actHeight = (theGrid.getGridHeight() * this.pixPerCell) + 1;
    setPreferredSize(new Dimension(actWidth, actHeight));

    this.offscreen = new BufferedImage(actWidth, actHeight, BufferedImage.TYPE_INT_RGB);
    this.graphics = (Graphics2D) (this.offscreen.getGraphics());
    this.graphics.setColor(Color.black);
    this.graphics.fillRect(0, 0, actWidth, actHeight);

    if (ANTIALIAS) {
        this.graphics.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
    }

    this.stroke = new BasicStroke(2);
    this.gridColor = new Color(220, 220, 255);

    setBackground(Color.WHITE);
}

/**
 * Sets the scale at which to draw force vectors
 *
 * @param  scale  the new scale
 */
public void setVectorScale(final double scale) {
```

```
        this.vectorScale = scale;
    }

    /**
     * Renders the component.
     *
     * @param  grx  The <code>Graphics</code> to which to draw.
     */
    @Override public void paintComponent(final Graphics grx) {

        synchronized (this.offscreen) {
            grx.drawImage(this.offscreen, 0, 0, null);
        }
    }

    /**
     * Renders the offscreen bitmap
     *
     * @param  framesPerSec  an optional frame rate to display on the panel (if zero, frame rate
     *                       will not be displayed)
     */
    public void update(final double framesPerSec) {

        int width;
        int height;
        double rad;
        GridMember2Int member;
        LinkedListGridMember2D linked;
        DynamicGridMember2D dynamic;
        double minX;
        double minY;
        double cellSize;
        int xPix;
        int yPix;
        int xPix2;
        int yPix2;
        Ellipse2D circle;
        StringBuilder str;
        LineSegment2D seg;
        int count;
        Stroke orig;

        this.graphics.setColor(Color.WHITE);
        this.graphics.fillRect(0, 0, getWidth(), getHeight());

        width = this.grid.getGridWidth();
        height = this.grid.getGridHeight();

        minX = this.grid.getMinX();
        minY = this.grid.getMinY();
        cellSize = this.grid.getGridCellSize();

        if (DRAW_GRID) {
            this.graphics.setColor(this.gridColor);
            xPix = 0;
            yPix = (height * this.pixPerCell);

            for (int i = 0; i <= width; i++) {
                this.graphics.drawLine(xPix, 0, xPix, yPix);
                xPix += this.pixPerCell;
            }

            yPix = 0;
            xPix = (width * this.pixPerCell);

            for (int i = 0; i <= height; i++) {
                this.graphics.drawLine(0, yPix, xPix, yPix);
                yPix += this.pixPerCell;
            }
        }

        // Draw the items
        circle = new Ellipse2D.Double(0, 0, 1, 1);

        this.iter.reset();
        count = 0;

        while (this.iter.hasNext()) {
            member = this.iter.next();

            count++;

            this.graphics.setColor(member.getColor());
            xPix = (int) ((member.getPosX() - minX) / cellSize * this.pixPerCell);
            yPix = (height * this.pixPerCell)
                - (int) ((member.getPosY() - minY) / cellSize * this.pixPerCell);
```

```
            if (member instanceof LinkedListGridMember2D) {
                linked = (LinkedListGridMember2D) member;

                if (linked.getNext() != null) {
                    xPix2 = (int) ((linked.getNext().getPosX() - minX) / cellSize
                            * this.pixPerCell);
                    yPix2 = (height * this.pixPerCell)
                        - (int) ((linked.getNext().getPosY() - minY) / cellSize * this.pixPerCell);
                    orig = this.graphics.getStroke();
                    this.graphics.setStroke(this.stroke);
                    this.graphics.drawLine(xPix, yPix, xPix2, yPix2);
                    this.graphics.setStroke(orig);
                }

                if (CIRCLES_ON_LINKED) {
                    rad = member.getRadius() / this.grid.getGridCellSize() * this.pixPerCell;

                    if (rad < 0.75) {
                        this.graphics.drawLine(xPix, yPix, xPix, yPix);
                    } else {
                        circle.setFrame(xPix - rad, yPix - rad, 2 * rad, 2 * rad);

                        if (member.fillColor() != null) {
                            this.graphics.setColor(member.fillColor());
                            this.graphics.fill(circle);
                            this.graphics.setColor(member.getColor());
                        }

                        this.graphics.draw(circle);
                    }
                }
            } else {
                rad = member.getRadius() / this.grid.getGridCellSize() * this.pixPerCell;

                if (rad < 0.75) {
                    this.graphics.drawLine(xPix, yPix, xPix, yPix);
                } else {
                    circle.setFrame(xPix - rad, yPix - rad, 2 * rad, 2 * rad);

                    if (member.fillColor() != null) {
                        this.graphics.setColor(member.fillColor());
                        this.graphics.fill(circle);
                        this.graphics.setColor(member.getColor());
                    }

                    this.graphics.draw(circle);
                }
            }

            if ((this.vectorScale > 0) && (member instanceof DynamicGridMember2D)) {

                dynamic = (DynamicGridMember2D) member;

                xPix2 = (int) ((member.getPosX() + (dynamic.getXForce() * this.vectorScale) - minX)
                        / cellSize * this.pixPerCell);
                yPix2 = (height * this.pixPerCell)
                    - (int) ((member.getPosY() + (dynamic.getYForce() * this.vectorScale) - minY)
                        / cellSize * this.pixPerCell);
                seg = new LineSegment2D(xPix, yPix, xPix2, yPix2);
                seg.clip(0, 0, getWidth(), getHeight());
                xPix = (int) seg.getX1();
                yPix = (int) seg.getY1();
                xPix2 = (int) seg.getX2();
                yPix2 = (int) seg.getY2();
                this.graphics.drawLine(xPix, yPix, xPix2, yPix2);
            }
        }

        if (framesPerSec != 0) {
            str = new StringBuilder(24);
            str.append("Frames/Sec: ");
            str.append(Float.toString((float) framesPerSec));
            str.append(" (");
            str.append(count);
            str.append(" model elements)");
            this.graphics.setColor(Color.GRAY);
            this.graphics.drawString(str.toString(), 10, this.offscreen.getHeight() - 10);
        }

        repaint();
    }

    /**
     * Exports a single frame of animation to a JPEG file.
     *
     * @param  frameNum  The integer frame number
     */
```

```
public void exportFrame(final int frameNum) {

    File file;
    ImageWriter writer = null;
    Iterator<?> writers;
    ImageOutputStream ios;

    file = new File("/imp/frames/frame_" + Integer.toString(frameNum / 1000)
            + Integer.toString((frameNum / 100) % 10) + Integer.toString((frameNum / 10) % 10)
            + Integer.toString(frameNum % 10) + ".png");

    synchronized (this) {
        writers = ImageIO.getImageWritersByFormatName("PNG");

        if (!writers.hasNext()) {
            LOG.warning("No PNG Writers Available");

            return;
        }

        writer = (ImageWriter) writers.next();

        if (file.exists()) {
            file.delete();
        }

        try {
            ios = ImageIO.createImageOutputStream(file);
            writer.setOutput(ios);
            writer.write(this.offscreen);
            ios.flush();
            writer.dispose();
            ios.close();
        } catch (IOException e) {
            LOG.log(Level.WARNING, "Exception writing frame", e);
        }
    }
}

/**
 * Handles mouse click events.
 *
 * @param  evt  the event
 */
public void mouseClicked(final MouseEvent evt) {

    // No action
}

/**
 * Handles mouse press events.
 *
 * @param  evt  the event
 */
public void mousePressed(final MouseEvent evt) {

    double xPos;
    double yPos;
    double radius;
    GridIterator2D iterator;
    GridMember2Int mem;

    // Get (X,Y) coordinates in object space of the mouse position
    xPos = ((double) evt.getX() / this.pixPerCell * this.grid.gridCellSize)
        + this.grid.getMinX();
    yPos = ((double) (getHeight() - evt.getY()) / this.pixPerCell * this.grid.gridCellSize)
        + this.grid.getMinY();
    radius = 3 * (this.grid.getMaxX() - this.grid.getMinX()) / getWidth();

    // See if there is a grid object within a couple of pixels of the press, and if so, set
    // that object as the one being dragged (NOTE: can't use global iterator since we're in
    // the AWT thread, and a non-AWT thread might be calling update)
    iterator = new GridIterator2D(this.grid);

    while (iterator.hasNext()) {
        mem = iterator.next();

        if ((mem.getPosX() >= (xPos - radius)) && (mem.getPosX() <= (xPos + radius))
                && (mem.getPosY() >= (yPos - radius)) && (mem.getPosY() <= (yPos + radius))) {
            this.dragging = mem;

            break;
        }
    }
}

/**
```

```
     * Handles mouse release events.
     *
     * @param  evt  the event
     */
    public void mouseReleased(final MouseEvent evt) {

        this.dragging = null;
    }

    /**
     * Handles mouse enter events.
     *
     * @param  evt  the event
     */
    public void mouseEntered(final MouseEvent evt) {

        // No action
    }

    /**
     * Handles mouse exit events.
     *
     * @param  evt  the event
     */
    public void mouseExited(final MouseEvent evt) {

        // No action
    }

    /**
     * Handles mouse drag events.
     *
     * @param  evt  the event
     */
    public void mouseDragged(final MouseEvent evt) {

        double xPos;
        double yPos;

        if (this.dragging != null) {

            // Get (X,Y) coordinates in object space of the mouse position
            xPos = ((double) evt.getX() / this.pixPerCell * this.grid.gridCellSize)
                + this.grid.getMinX();
            yPos = ((double) (getHeight() - evt.getY()) / this.pixPerCell * this.grid.gridCellSize)
                + this.grid.getMinY();

            this.dragging.setPos(xPos, yPos);
        }
    }

    /**
     * Handles mouse move events.
     *
     * @param  evt  the event
     */
    public void mouseMoved(final MouseEvent evt) {

        // No action
    }
}

package com.srbenoit.modeling.grid;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import com.srbenoit.log.LoggedObject;

/**
 * A class to test the <code>Grid</code> class by constructing a grid and randomly adding and
 * removing many objects, then testing the grid state for correctness and ensuring that neighbor
 * lists returned are valid.
 */
public class GridTest2D extends LoggedObject implements Runnable {

    /** the number of test iterations */
    private static final int NUM_ITERATIONS = 200000;

    /** the maximum number of objects to have in the grid at a time */
    private static final int MAX_OBJECTS = 100000;

    /** the grid under test */
```

```
private final Grid2D grid;

/** the panel that will render the grid */
private GridPanel2D panel;

/**
 * Constructs a new <code>GridTest</code>.
 */
public GridTest2D() {

    // build test grid: X ranges from 0-20, Y ranges from 0-10
    this.grid = new Grid2D(0, 0, 1.0, 20, 10);
}

/**
 * Gets the grid being tested.
 *
 * @return  the grid
 */
public Grid2D getGrid() {

    return this.grid;
}

/**
 * Builds the frame and panel in the AWT event thread.
 */
@Override public void run() {

    JFrame frame;

    frame = new JFrame("Grid_Test");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    this.panel = new GridPanel2D(800, 800, this.grid);
    frame.setContentPane(this.panel);
    frame.pack();
    frame.setVisible(true);
}

/**
 * Executes the tests.
 */
public void execute() {

    Random rnd;
    List<GridMember2Int> objects;
    float pctFull;
    int index;
    boolean addNew;
    boolean move;
    boolean delete;
    GridMember2Int member;
    GridMember2Int test;
    GridIterator2D iter;
    int count;
    int xAddr;
    int yAddr;
    double distX;
    double distY;

    try {
        SwingUtilities.invokeAndWait(this);
    } catch (Exception ex) {
        Logger.getLogger(GridTest2D.class.getName()).log(Level.SEVERE, null, ex);

        return;
    }

    rnd = new Random(System.currentTimeMillis());
    objects = new ArrayList<GridMember2Int>(MAX_OBJECTS);

    for (int i = 0; i < NUM_ITERATIONS; i++) {
        pctFull = (float) objects.size() / MAX_OBJECTS;

        addNew = false;
        move = false;
        delete = false;

        // If the list is empty, we always add a node
        if (rnd.nextFloat() < pctFull) {

            if (rnd.nextBoolean()) {
                move = true;
            } else {
                delete = true;
            }
```

```
            } else {
                addNew = true;
            }

            if (addNew) {
                member = new DynamicGridMember2D(rnd.nextDouble() * 20, rnd.nextDouble() * 10, // NOPMD SRB
                        0, EnumElementType.FIXED, 1, 1);
                objects.add(member);
                member.installInGrid(this.grid);
            } else {
                index = rnd.nextInt(objects.size());

                if (move) {
                    member = objects.get(index);
                    member.move(rnd.nextDouble() * 20, rnd.nextDouble() * 10);
                } else if (delete) {
                    member = objects.remove(index);
                    member.removeFromGrid();
                }
            }
        }

        this.panel.update(0);
        LOG.log(Level.FINE, "After test, {0}/{1} objects",
            new Object[] { objects.size(), getGrid().getNumObjects() });

        // See if all elements in the grid have the appropriate address
        iter = new GridIterator2D(grid);
        count = 0;

        while (iter.hasNext()) {
            member = iter.next();

            xAddr = grid.getXAddress(member.getGridIndex());
            yAddr = grid.getYAddress(member.getGridIndex());

            if (xAddr != (int) member.getPosX()) {
                LOG.log(Level.WARNING, "X address mismatch: {0},{1}",
                    new Object[] { xAddr, member.getPosX() }); // NOPMD SRB
            }

            if (yAddr != (int) member.getPosY()) {
                LOG.log(Level.WARNING, "Y address mismatch: {0},{1}",
                    new Object[] { yAddr, member.getPosX() }); // NOPMD SRB
            }

            count++;
        }

        // Get all the neighbors of the first item
        index = this.grid.getNextFilled(0);
        member = this.grid.get(index);
        this.grid.getNeighborsOf(member, 0.5);

        LOG.log(Level.FINE, "{0} neighbors of point at ({1}, {2}) {3}",
            new Object[] { member.getNumNeighbors(), member.getPosX(), member.getPosY(), count });

        for (int i = 0; i < member.getNumNeighbors(); i++) {
            test = member.getNeighbor(i);

            distX = test.getPosX() - member.getPosX();
            distY = test.getPosY() - member.getPosY();

            if ((distX < -1.5) || (distX > 1.5) || (distY < -1.5) || (distY > 1.5)) {
                LOG.log(Level.FINE, "  ({0}, {1}) [{2}, {3}]",
                    new Object[] { test.getPosX(), test.getPosY(), distX, distY }); // NOPMD SRB
            }
        }
    }

    /**
     * Main method to run the test.
     *
     * @param  args  Command-line arguments (ignored).
     */
    public static void main(final String... args) {

        new GridTest2D().execute();
    }
}

package com.srbenoit.modeling.grid;

/**
 * A base class for objects that may be contained in a 2-dimensional grid. Each object has an X and
 * Y position, a radius, and a grid address.
 */
```

```
public class LinkedListGridMember2D extends DynamicGridMember2D {

    /** the next element in the list */
    private LinkedListGridMember2D flink;

    /** the prior element in the list */
    private LinkedListGridMember2D blink;

    /**
     * Constructs a new <code>LinkedListGridMember</code>.
     *
     * @param  xCoord        the X coordinate
     * @param  yCoord        the Y coordinate
     * @param  rad           the radius of the object
     * @param  theType       the element type
     * @param  structureId   the ID of the structure this member belongs to
     * @param  theMass       the mass of the object
     */
    public LinkedListGridMember2D(final double xCoord, final double yCoord, final double rad,
        final EnumElementType theType, final int structureId, final double theMass) {

        super(xCoord, yCoord, rad, theType, structureId, theMass);

        this.flink = this;
        this.blink = this;
    }

    /**
     * Adds this member in the linked list before a given element.
     *
     * @param  elem  the element before which to add this element
     */
    public void addBefore(final LinkedListGridMember2D elem) {

        this.setPrior(elem.getPrior());
        this.setNext(elem);
        elem.getPrior().setNext(this);
        elem.setPrior(this);
    }

    /**
     * Adds this member in the linked list after a given element.
     *
     * @param  elem  the element after which to add this element
     */
    public void addAfter(final LinkedListGridMember2D elem) {

        setNext(elem.getNext());
        setPrior(elem);
        elem.getNext().setPrior(this);
        elem.setNext(this);
    }

    /**
     * Removes this element from the list in which it is installed.
     */
    public void remove() {

        this.getNext().setPrior(getPrior());
        this.getPrior().setNext(getNext());
        setNext(this);
        setPrior(this);
    }

    /**
     * Gets the next item in the linked list.
     *
     * @return  the next item
     */
    public LinkedListGridMember2D getNext() {

        return this.flink;
    }

    /**
     * Gets the prior item in the linked list.
     *
     * @return  the prior item
     */
    public LinkedListGridMember2D getPrior() {

        return this.blink;
    }

    /**
     * Sets the next item in the linked list.
     *
```

```
     * @param  next  the next item
     */
    public void setNext(final LinkedListGridMember2D next) {

        this.flink = next;
    }

    /**
     * Sets the prior item in the linked list.
     *
     * @param  prior  the prior item
     */
    public void setPrior(final LinkedListGridMember2D prior) {

        this.blink = prior;
    }
}

package com.srbenoit.modeling.grid;

import java.awt.Color;
import java.util.logging.Level;
import com.srbenoit.geom.Point2;

/**
 * A basic grid member based on <code>Point2</code>.
 */
public class PointGridMember2 extends Point2 implements GridMember2Int {

    /** the grid in which this member object is installed */
    private Grid2D grid;

    /** the index of the member in the grid */
    private int gridIndex;

    /** the number of neighbors this object currently has */
    private int numNeighbors;

    /** the objects that are neighbors of this object */
    private GridMember2Int[] neighbors;

    /**
     * Constructs a new <code>PointGridMember2</code>.
     *
     * @param  xCoord  the X coordinate
     * @param  yCoord  the Y coordinate
     */
    public PointGridMember2(final double xCoord, final double yCoord) {

        super(xCoord, yCoord);

        this.grid = null;
        this.gridIndex = -1;

        this.numNeighbors = 0;
        this.neighbors = new GridMember2Int[4];
    }

    /**
     * Gets this grid this object is installed in.
     *
     * @return  the grid
     */
    public Grid2D getGrid() {

        return this.grid;
    }

    /**
     * Gets the index of the object in the grid where it is installed.
     *
     * @return  the grid index
     */
    public int getGridIndex() {

        return this.gridIndex;
    }

    /**
     * Gets the radius to use when drawing the grid member.
     *
     * @return  the radius (0 to draw as a point)
     */
    public double getRadius() {

        return 0;
    }
```

```
/**
 * Gets the color in which to render the element.
 *
 * @return  the color
 */
public Color getColor() {

    return Color.BLACK;
}

/**
 * Gets the fill color in which to render the element.
 *
 * @return  the color, or <code>null</code> for no fill
 */
public Color fillColor() {

    return null;
}

/**
 * Installs the object in a grid. If the object is a member of a grid when this method is
 * called, the object is first removed from that grid, then added to <code>theGrid</code>.
 *
 * @param  theGrid  the grid in which the object is being installed
 */
public void installInGrid(final Grid2D theGrid) {

    if (this.grid != null) {
        removeFromGrid();
    }

    if (theGrid == null) {
        this.gridIndex = -1;
    } else {
        this.gridIndex = theGrid.add(this);
    }

    this.grid = theGrid;
}

/**
 * Removes the object from the grid in which it is installed, if any.
 */
public void removeFromGrid() {

    if (this.grid == null) {
        LOG.warning("Attempt to remove member from grid that was not in a grid");
    } else {
        this.grid.remove(this.gridIndex);
        this.grid = null;
        this.gridIndex = -1;
    }
}

/**
 * Moves the object to a new position. If the object is installed in a grid, the grid is
 * notified so it can update the object's grid cell address.
 *
 * @param  xCoord  the new X coordinate
 * @param  yCoord  the new Y coordinate
 */
@Override public void setPos(final double xCoord, final double yCoord) {

    if (Double.isNaN(xCoord) || Double.isNaN(yCoord)) {
        LOG.log(Level.WARNING, "Attempt to set grid member position to NaN", new Exception());
        System.exit(1);
    }

    if (Double.isInfinite(xCoord) || Double.isInfinite(xCoord)) {
        LOG.log(Level.WARNING, "Attempt to set grid member position by Infinity",
            new Exception());
        System.exit(1);
    }

    super.setPos(xCoord, yCoord);

    if (this.grid != null) {
        this.grid.move(this.gridIndex, xCoord, yCoord);
    }
}

/**
 * Moves the object, by adjusting its position by a specified amount. If the object is
 * installed in a grid, the grid is notified so it can update the object's grid cell address.
 *
```

```
     * @param  deltaX  the change in X coordinate
     * @param  deltaY  the change in Y coordinate
     */
    @Override public void move(final double deltaX, final double deltaY) {

        if (Double.isNaN(deltaX) || Double.isNaN(deltaY)) {
            LOG.log(Level.WARNING, "Attempt to move grid member position by NaN", new Exception());
            System.exit(1);
        }

        if (Double.isInfinite(deltaX) || Double.isInfinite(deltaY)) {
            LOG.log(Level.WARNING, "Attempt to move grid member position by Infinity",
                new Exception());
            System.exit(1);
        }

        super.move(deltaX, deltaY);

        if (this.grid != null) {
            this.grid.move(this.gridIndex, getPosX(), getPosY());
        }
    }

    /**
     * Clears the list of this object's neighbors.
     */
    public void clearNeighbors() {

        this.numNeighbors = 0;
    }

    /**
     * Adds an object to the list of this object's neighbors.
     *
     * @param  neighbor  the neighbor to add
     */
    public void addNeighbor(final GridMember2Int neighbor) {

        GridMember2Int[] newList;

        if (this.numNeighbors == this.neighbors.length) {
            newList = new GridMember2Int[this.numNeighbors + 4];
            System.arraycopy(this.neighbors, 0, newList, 0, this.numNeighbors);
            this.neighbors = newList;
        }

        this.neighbors[this.numNeighbors] = neighbor;
        this.numNeighbors++;
    }

    /**
     * Gets the number of neighbors the member has.
     *
     * @return  the number of neighbors
     */
    public int getNumNeighbors() {

        return this.numNeighbors;
    }

    /**
     * Gets a neighbor grid member.
     *
     * @param   index  the index of the neighbor
     * @return  the neighbor
     */
    public GridMember2Int getNeighbor(final int index) {

        return this.neighbors[index];
    }
}
```

E.6.2 Triangulated Meshes (com.srbenoit.modeling.mesh/code¿)

This package provides the beginnings of a triangulated mesh representation of a manifold for rendering in a three dimensional visualization. This is currently just enough

to beign simulating cell membranes in 3-D.

```
package com.srbenoit.modeling.mesh;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Toolkit;
import java.lang.reflect.InvocationTargetException;
import java.util.logging.Level;
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JSlider;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import com.srbenoit.geom.BasedVector3;
import com.srbenoit.geom.Vector3;
import com.srbenoit.render.Camera;
import com.srbenoit.render.Light;
import com.srbenoit.render.RenderPanel;
import com.srbenoit.render.Scene;
import com.srbenoit.render.WorldFace;
import com.srbenoit.render.WorldVertex;

/**
 * A class that builds a small test patch consisting of 6 faces, 7 vertices, and 12 edges (actually
 * 24 edges, including each direction).
 */
public class HexPatch extends Scene implements ChangeListener, Runnable {

    /** equilibrium separation of elements (\varepsilon in paper) */
    public static final double EPS = 1e-7;

    /** the Lennard-Jones well depth for element interactions */
    public static final double KLJ = 1e3;

    /** the bulk modulus */
    public static final double KB = 6750;

    /** the tension */
    public static final double TENSION = 3e-5;

    /** the elastic modulus */
    // public static final double KC = 7e-20;
    public static final double KC = 7e1;

    /** the soft-sphere well depth for anti-self-intersection interactions */
    public static final double KSS = 1e3;

    /** scale to apply to force vector for display */
    public static final double FORCE_SCALE = 1e-18;

    /** the indices of the seven vertices that make up the patch */
    private final MembraneVertex[] vertices;

    /** the indices of the six faces that make up the patch */
    private final MembraneFace[] faces;

    /** the panel that will render the system */
    private RenderPanel panel;

    /** the camera that will be used in rendering the scene */
    private Camera camera;

    /** slider to control X coordinate of central point */
    private JSlider xSlider;

    /** slider to control Y coordinate of central point */
    private JSlider ySlider;

    /** slider to control Z coordinate of central point */
    private JSlider zSlider;

    /** a based vector to display a force */
    private BasedVector3 vec;

    /** radio button to select element interaction force */
    private JRadioButton elemFrc;

    /** radio button to select pressure force */
    private JRadioButton presFrc;
```

```
/** radio button to select tension force */
private JRadioButton tensFrc;

/** radio button to select curvature force */
private JRadioButton curvFrc;

/** radio button to select self-intersection force */
private JRadioButton siFrc;

/** the equilibrium volume of the cell */
private double defVolume;

/** the current volume of the cell */
private double curVolume;

/**
 * Constructs a new <code>HexPatch</code>.
 */
public HexPatch() {

    super(16);

    double r3o2;

    r3o2 = Math.sqrt(3) / 2;

    this.vertices = new MembraneVertex[12];
    this.faces = new MembraneFace[16];

    this.vertices[0] = new MembraneVertex(0.5 * EPS, -r3o2 * EPS, 0, 1, EPS);
    this.vertices[1] = new MembraneVertex(EPS, 0, 0, 1, EPS);
    this.vertices[2] = new MembraneVertex(0.5 * EPS, r3o2 * EPS, 0, 1, EPS);
    this.vertices[3] = new MembraneVertex(-0.5 * EPS, r3o2 * EPS, 0, 1, EPS);
    this.vertices[4] = new MembraneVertex(-EPS, 0, 0, 1, EPS);
    this.vertices[5] = new MembraneVertex(-0.5 * EPS, -r3o2 * EPS, 0, 1, EPS);
    this.vertices[6] = new MembraneVertex(0, 0, 0.5 * EPS, 1, EPS);
    this.vertices[7] = new MembraneVertex(0, 0, -1.1 * EPS, 1, EPS);

    this.vertices[8] = new MembraneVertex(0.3 * EPS, 0.3 * EPS, 1.6 * EPS, 2, 0.5 * EPS);
    this.vertices[9] = new MembraneVertex(0.3 * EPS, -0.3 * EPS, 1.6 * EPS, 2, 0.5 * EPS);
    this.vertices[10] = new MembraneVertex(-0.3 * EPS, -0.3 * EPS, 1.6 * EPS, 2, 0.5 * EPS);
    this.vertices[11] = new MembraneVertex(-0.3 * EPS, 0.3 * EPS, 1.6 * EPS, 2, 0.5 * EPS);

    for (int i = 0; i < this.vertices.length; i++) {
        addVertex(this.vertices[i]);
    }

    // the top surface of the "cell"
    this.faces[0] = new MembraneFace(this.vertices[0], this.vertices[1], this.vertices[6], 1);
    this.faces[1] = new MembraneFace(this.vertices[1], this.vertices[2], this.vertices[6], 1);
    this.faces[2] = new MembraneFace(this.vertices[2], this.vertices[3], this.vertices[6], 1);
    this.faces[3] = new MembraneFace(this.vertices[3], this.vertices[4], this.vertices[6], 1);
    this.faces[4] = new MembraneFace(this.vertices[4], this.vertices[5], this.vertices[6], 1);
    this.faces[5] = new MembraneFace(this.vertices[5], this.vertices[0], this.vertices[6], 1);

    // the bottom surface of the "cell"
    this.faces[6] = new MembraneFace(this.vertices[1], this.vertices[0], this.vertices[7], 1);
    this.faces[7] = new MembraneFace(this.vertices[2], this.vertices[1], this.vertices[7], 1);
    this.faces[8] = new MembraneFace(this.vertices[3], this.vertices[2], this.vertices[7], 1);
    this.faces[9] = new MembraneFace(this.vertices[4], this.vertices[3], this.vertices[7], 1);
    this.faces[10] = new MembraneFace(this.vertices[5], this.vertices[4], this.vertices[7], 1);
    this.faces[11] = new MembraneFace(this.vertices[0], this.vertices[5], this.vertices[7], 1);

    // an object for the cell to interact with
    this.faces[12] = new MembraneFace(this.vertices[8], this.vertices[9], this.vertices[10],
            2);
    this.faces[13] = new MembraneFace(this.vertices[8], this.vertices[10], this.vertices[11],
            2);
    this.faces[14] = new MembraneFace(this.vertices[8], this.vertices[10], this.vertices[9],
            2);
    this.faces[15] = new MembraneFace(this.vertices[8], this.vertices[11], this.vertices[10],
            2);

    for (int i = 0; i < this.faces.length; i++) {
        addFace(this.faces[i]);
    }

    // Create a camera that is slightly elevated above the X-Y plane, looking at the origin
    this.camera = new Camera(4 * EPS);
    this.camera.setPolarAngle(Math.PI / 3, true);

    addLight(new Light(80 * EPS, -80 * EPS, 100 * EPS, Color.WHITE));

    this.vec = new BasedVector3(0, 0, 0.5 * EPS, 0, 0, EPS);
    addBasedVector(vec);

    this.defVolume = computeVolume(1);
```

```
        LOG.log(Level.INFO, "Defeault volume = {0}", this.defVolume);
    }

    /**
     * Method to be called in the AWT event thread to construct the user interface.
     */
    @Override public void run() {

        JFrame frame;
        JPanel content;
        JPanel radios;
        Dimension screen;
        Dimension size;
        ButtonGroup group;

        frame = new JFrame("Hex Patch Visualization");
        content = new JPanel(new BorderLayout());
        frame.setContentPane(content);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.panel = new RenderPanel(500, 400, this.camera);
        this.panel.addMouseListener(this.panel);
        this.panel.addMouseMotionListener(this.panel);

        content.add(this.panel, BorderLayout.CENTER);

        this.xSlider = new JSlider(SwingConstants.HORIZONTAL, -1000, 1000, 0);
        this.xSlider.addChangeListener(this);
        content.add(this.xSlider, BorderLayout.SOUTH);

        this.ySlider = new JSlider(SwingConstants.VERTICAL, -1000, 1000, 0);
        this.ySlider.addChangeListener(this);
        content.add(this.ySlider, BorderLayout.WEST);

        this.zSlider = new JSlider(SwingConstants.VERTICAL, -1000, 1000, 500);
        this.zSlider.addChangeListener(this);
        content.add(this.zSlider, BorderLayout.EAST);

        radios = new JPanel(new GridLayout(0, 5));
        this.elemFrc = new JRadioButton("elem");
        this.presFrc = new JRadioButton("pres");
        this.tensFrc = new JRadioButton("tens");
        this.curvFrc = new JRadioButton("curv");
        this.siFrc = new JRadioButton("si");
        radios.add(this.elemFrc);
        radios.add(this.presFrc);
        radios.add(this.tensFrc);
        radios.add(this.curvFrc);
        radios.add(this.siFrc);
        group = new ButtonGroup();
        group.add(this.elemFrc);
        group.add(this.presFrc);
        group.add(this.tensFrc);
        group.add(this.curvFrc);
        group.add(this.siFrc);
        this.curvFrc.setSelected(true);
        content.add(radios, BorderLayout.NORTH);

        frame.pack();
        screen = Toolkit.getDefaultToolkit().getScreenSize();
        size = frame.getSize();
        frame.setLocation((screen.width - size.width) / 2, (screen.height - size.height) / 2);
        frame.setVisible(true);
    }

    /**
     * Renders the scene.
     */
    public void go() {

        while (this.panel.isVisible()) {

            this.curVolume = computeVolume(1); // this refreshes face normals

            // TODO: Compute forces given current node positions, and update the based vector that
            // displays the force magnitude.
            if (this.elemFrc.isSelected()) {
                computeElemForce(6, this.vec);
            } else if (this.presFrc.isSelected()) {
                computePresForce(6, this.vec);
            } else if (this.tensFrc.isSelected()) {
                computeTensForce(6, this.vec);
            } else if (this.curvFrc.isSelected()) {
                computeCurvForce(6, this.vec);
            } else if (this.siFrc.isSelected()) {
                computeSiForce(6, this.vec);
            }
```

```
            vec.scaleVec(FORCE_SCALE);

            this.panel.render(this);

            try {
                Thread.sleep(20);
            } catch (InterruptedException ex) {
                // No action
            }
        }
    }

    /**
     * Computes the element interaction force on a vertex.
     *
     * @param  index  the index of the vertex whose force to compute
     * @param  vec    the vector in which to store the computed force
     */
    private void computeElemForce(final int index, final BasedVector3 vec) {

        MembraneVertex target;
        MembraneVertex vert;
        Vector3 delta;
        double dist;
        double ratio;
        double cube;
        double sixth;
        double twelfth;
        double scale;

        vec.setVec(0, 0, 0);
        delta = new Vector3();

        target = this.vertices[index];

        for (int i = 0; i < this.vertices.length; i++) {
            vert = this.vertices[i];

            if (vert.getMembraneId() == target.getMembraneId()) {
                continue;
            }

            delta.vectorBetween(target, vert);
            dist = delta.length();
            ratio = (EPS + vert.getRadius()) / dist;

            cube = ratio * ratio * ratio;
            sixth = cube * cube;
            twelfth = sixth * sixth;

            scale = -KLJ * (twelfth - sixth) / (dist * dist);
            vec.addVecScaled(scale, delta);
        }
    }

    /**
     * Computes the pressure force on a vertex.
     *
     * @param  index  the index of the vertex whose force to compute
     * @param  vec    the vector in which to store the computed force
     */
    private void computePresForce(final int index, final BasedVector3 vec) {

        MembraneVertex vert;
        double scale;
        WorldFace face;
        int indexInFace;
        Vector3 other1;
        Vector3 other2;
        WorldVertex world;
        Vector3 cross;

        vert = this.vertices[index];
        other1 = new Vector3();
        other2 = new Vector3();
        cross = new Vector3();

        // TODO: Look up default volume based on membrane ID
        scale = (-KB / (6 * this.defVolume)) * (this.curVolume - this.defVolume);

        vec.setVec(0, 0, 0);

        for (int i = 0; i < vert.getNumFaces(); i++) {
            face = vert.getFace(i);
            indexInFace = vert.getIndexInFace(i);
            world = face.getVertex(indexInFace + 1);
```

```
            other1.setVec(world.getPosX(), world.getPosY(), world.getPosZ());
            world = face.getVertex(indexInFace + 1);
            other2.setVec(world.getPosX(), world.getPosY(), world.getPosZ());
            cross.cross(other1, other2);
            vec.addVecScaled(scale, cross);
        }
    }

    /**
     * Computes the tension force on a vertex.
     *
     * @param  index  the index of the vertex whose force to compute
     * @param  vec    the vector in which to store the computed force
     */
    private void computeTensForce(final int index, final BasedVector3 vec) {

        MembraneVertex vert;
        double scale;
        WorldFace face;
        int indexInFace;
        WorldVertex other1;
        WorldVertex other2;
        Vector3 diff;
        Vector3 cross;

        vert = this.vertices[index];
        diff = new Vector3();
        cross = new Vector3();

        scale = -TENSION / 4;

        vec.setVec(0, 0, 0);

        for (int i = 0; i < vert.getNumFaces(); i++) {
            face = vert.getFace(i);
            indexInFace = vert.getIndexInFace(i);
            other1 = face.getVertex(indexInFace + 1);
            other2 = face.getVertex(indexInFace - 1);
            diff.vectorBetween(other1, other2);
            cross.cross(face, diff);
            vec.addVecScaled(scale, cross);
        }
    }

    /**
     * Computes the curvature force on a vertex.
     *
     * @param  index  the index of the vertex whose force to compute
     * @param  vec    the vector in which to store the computed force
     */
    private void computeCurvForce(final int index, final BasedVector3 vec) {

        MembraneVertex vert;
        double scale;
        int numFaces;
        WorldFace face1;
        WorldFace face2;
        WorldFace face3;
        int indexInFace1;
        int indexInFace2;
        WorldVertex end1;
        WorldVertex end2;
        WorldVertex end3;
        Vector3 ej0;
        Vector3 ej1;
        Vector3 ej2;
        Vector3 ej1minusej0;
        Vector3 ej2minusej1;
        Vector3 cross01;
        Vector3 cross12;
        Vector3 left1;
        Vector3 right1;
        Vector3 left;
        Vector3 right;
        Vector3 cross;
        double dot;
        Vector3 term;
        double coeff;
        boolean hit;

        vert = this.vertices[index];
        ej0 = new Vector3();
        ej1 = new Vector3();
        ej2 = new Vector3();
        ej1minusej0 = new Vector3();
        ej2minusej1 = new Vector3();
        cross01 = new Vector3();
```

```
cross12 = new Vector3();
left1 = new Vector3();
right1 = new Vector3();
left = new Vector3();
right = new Vector3();
cross = new Vector3();
term = new Vector3();

scale = 4 * KC / EPS;

vec.setVec(0, 0, 0);

numFaces = vert.getNumFaces();

for (int i = 0; i < numFaces; i++) {
    face1 = vert.getFace(i);
    indexInFace1 = vert.getIndexInFace(i);

    end1 = face1.getVertex(indexInFace1 + 1);
    end2 = face1.getVertex(indexInFace1 - 1);

    // Find another face whose right edge is face1's left edge
    for (int j = 0; j < numFaces; j++) {

        if (j == i) {
            continue;
        }

        face2 = vert.getFace(j);
        indexInFace2 = vert.getIndexInFace(j);

        if (face2.getVertex(indexInFace2 + 1) == end2) {

            term.setVec(0, 0, 0);

            end3 = face2.getVertex(indexInFace2 - 1);

            // Comupte the contribution of the edge between two faces
            ej0.vectorBetween(vert, end1);
            ej1.vectorBetween(vert, end2);
            ej2.vectorBetween(vert, end3);
            ej1minusej0.subVec(ej1, ej0);
            ej2minusej1.subVec(ej2, ej1);

            dot = face1.dot(face2);
            term.addVecScaled((1 - dot) / ((1 + dot) * ej1.length()), ej1);

            coeff = 2 * ej1.length() / ((1 + dot) * (1 + dot));
            cross01.cross(ej0, ej1);
            cross12.cross(ej1, ej2);

            left1.setVec(face2);
            left1.addVecScaled(-dot, face1);
            left1.scaleVec(1 / cross01.length());
            left.cross(left1, ej1minusej0);

            right1.setVec(face1);
            right1.addVecScaled(-dot, face2);
            right1.scaleVec(1 / cross12.length());
            right.cross(right1, ej2minusej1);

            left.addVec(right);
            term.addVecScaled(coeff, left);

            // Find the face that shares the edge opposite the vertex
            for (int inx = 0; inx < end1.getNumFaces(); inx++) {
                face3 = end1.getFace(inx);

                if (face3 == face1) {
                    continue;
                }

                if ((face3.getVertex0() == end2) || (face3.getVertex1() == end2)
                        || (face3.getVertex2() == end2)) {

                    // face3 is the face with normal vector $\eta_j$.
                    dot = face1.dot(face3);
                    coeff = 2 * ej1minusej0.length() / ((1 + dot) * (1 + dot));

                    left.setVec(face3);
                    left.addVecScaled(-dot, face1);

                    right.setVec(ej1minusej0);
                    right.scaleVec(1 / cross01.length());

                    cross.cross(left, right);
                    term.addVecScaled(coeff, cross);
```

```
                            break;
                        }
                    }

                    vec.addVecScaled(scale, term);

                    break;
                }
            }
        }
    }

    /**
     * Computes the anti-self-intersection force on a vertex.
     *
     * @param  index  the index of the vertex whose force to compute
     * @param  vec    the vector in which to store the computed force
     */
    private void computeSiForce(final int index, final BasedVector3 vec) {

        MembraneVertex target;
        MembraneVertex vert;
        int count1;
        int count2;
        Vector3 delta;
        double dist;
        double ratio;
        double cube;
        double sixth;
        double twelfth;
        double scale;
        boolean shared;

        vec.setVec(0, 0, 0);
        delta = new Vector3();

        target = this.vertices[index];
        count1 = target.getNumFaces();

        for (int i = 0; i < this.vertices.length; i++) {
            vert = this.vertices[i];

            if (vert.getMembraneId() != target.getMembraneId()) {
                continue;
            }

            // If the vertex shares a face with the target vertex, ignore it
            count2 = vert.getNumFaces();
            shared = false;

outer:
            for (int j = 0; j < count1; j++) {

                for (int k = 0; k < count2; k++) {

                    if (target.getFace(j) == vert.getFace(k)) {
                        shared = true;

                        break outer;
                    }
                }
            }

            if (shared) {
                continue;
            }

            delta.vectorBetween(target, vert);
            dist = delta.length();
            ratio = EPS / dist;

            cube = ratio * ratio * ratio;
            sixth = cube * cube;
            twelfth = sixth * sixth;

            scale = -12 * KSS * twelfth / (dist * dist);
            vec.addVecScaled(scale, delta);
        }
    }

    /**
     * Handler for state changes from sliders.
     *
     * @param  evt  the change event
     */
    @Override public void stateChanged(final ChangeEvent evt) {
```

```
        double xCoord;
        double yCoord;
        double zCoord;

        xCoord = EPS * this.xSlider.getValue() / 500.0;
        yCoord = EPS * this.ySlider.getValue() / 1000.0;
        zCoord = EPS * this.zSlider.getValue() / 1000.0;

        this.vertices[6].setPos(xCoord, yCoord, zCoord);
        this.vec.setPos(xCoord, yCoord, zCoord);

        for (int i = 0; i < this.faces.length; i++) {
            this.faces[i].computeNormal();
        }
    }

    /**
     * Computes the current volume of a membrane.
     *
     * @param   index  the index of the membrane
     * @return  the volume
     */
    private double computeVolume(final int index) {

        double volume;
        MembraneFace face;
        WorldVertex vert0;
        WorldVertex vert1;
        WorldVertex vert2;
        Vector3 vector;
        Vector3 vec01;
        Vector3 vec02;
        double dot;
        Vector3 cross;

        volume = 0;
        vector = new Vector3();
        vec01 = new Vector3();
        vec02 = new Vector3();
        cross = new Vector3();

        for (int i = 0; i < this.faces.length; i++) {
            face = this.faces[i];

            if (face.getMembraneId() != index) {
                continue;
            }

            face.computeNormal();

            vert0 = face.getVertex0();
            vert1 = face.getVertex1();
            vert2 = face.getVertex2();

            vec01.vectorBetween(vert0, vert1);
            vec02.vectorBetween(vert0, vert2);

            vector.setVec(vert0.getPosX(), vert0.getPosY(), vert0.getPosZ());
            dot = vector.dot(face);
            cross.cross(vec01, vec02);

            volume += cross.length() * dot / 6;
        }

        return volume;
    }

    /**
     * Main method to create a hex patch and a render panel to display it.
     *
     * @param  args
     */
    public static void main(final String... args) {

        HexPatch hex;

        hex = new HexPatch();

        try {
            SwingUtilities.invokeAndWait(hex);
            hex.go();
        } catch (InterruptedException ex) {
            LOG.log(Level.SEVERE, null, ex);
        } catch (InvocationTargetException ex) {
            LOG.log(Level.SEVERE, null, ex);
        }
```

```
    }
}

package com.srbenoit.modeling.mesh;

import com.srbenoit.render.WorldFace;

/**
 * A face that is part of a membrane.
 */
public class MembraneFace extends WorldFace {

    /** a unique ID for the membrane this face belongs to */
    private final int membraneId;

    /**
     * Constructs a new <code>MembraneFace</code>.
     *
     * @param  vert0           the first vertex in the face
     * @param  vert1           the second vertex in the face
     * @param  vert2           the third vertex in the face
     * @param  whichMembrane   the unique ID of the membrane this face belongs to
     */
    public MembraneFace(final MembraneVertex vert0, final MembraneVertex vert1,
        final MembraneVertex vert2, final int whichMembrane) {

        super(vert0, vert1, vert2);
        this.membraneId = whichMembrane;

        vert0.addFace(this, 0);
        vert1.addFace(this, 1);
        vert2.addFace(this, 2);
    }

    /**
     * Gets the unique ID of the membrane this face belongs to.
     *
     * @return  the membrane ID
     */
    public int getMembraneId() {

        return this.membraneId;
    }
}

package com.srbenoit.modeling.mesh;

import com.srbenoit.render.WorldVertex;

/**
 * A vertex that is part of a membrane.
 */
public class MembraneVertex extends WorldVertex {

    /** a unique ID for the membrane this vertex belongs to */
    private final int membraneId;

    /** the radius of the vertex for interactions */
    private final double radius;

    /**
     * Constructs a new <code>MembraneVertex</code>.
     *
     * @param  whichMembrane   the unique ID of the membrane this face belongs to
     * @param  rad             the radius of the vertex for interactions
     */
    public MembraneVertex(final int whichMembrane, final double rad) {

        super();
        this.membraneId = whichMembrane;
        this.radius = rad;
    }

    /**
     * Constructs a new <code>MembraneVertex</code>.
     *
     * @param  xCoord          the X coordinate
     * @param  yCoord          the Y coordinate
     * @param  zCoord          the Z coordinate
     * @param  whichMembrane   the unique ID of the membrane this face belongs to
     * @param  rad             the radius of the vertex for interactions
     */
    public MembraneVertex(final double xCoord, final double yCoord, final double zCoord,
        final int whichMembrane, final double rad) {

        super(xCoord, yCoord, zCoord);
```

```
        this.membraneId = whichMembrane;
        this.radius = rad;
    }

    /**
     * Gets the unique ID of the membrane this vertex belongs to.
     *
     * @return  the membrane ID
     */
    public int getMembraneId() {

        return this.membraneId;
    }

    /**
     * Gets the radius of the vertex for interactions
     *
     * @return  the radius
     */
    public double getRadius() {

        return this.radius;
    }
}
```

E.6.3 Models of Cells and Cell Behavior (com.srbenoit.modeling.cell/co

```
package com.srbenoit.modeling.cell;

import com.srbenoit.geom.Vector2;
import com.srbenoit.modeling.grid.GridMember2Int;
import com.srbenoit.modeling.grid.PointGridMember2;

/**
 * An actin filament. Each filament is characterized by a point position of its pointed end, its
 * length, and a set of crosslinks to other objects (filaments or membrane elements). Each
 * maintains an upstream link to the prior filament (null if this filament is adjacent to the
 * membrane and subject to polymerization, a downstream link to the next filament in the chain
 * (null if this filament is the furthest from the membrane and subject to deploymerization), and a
 * reference to the membrane element this filament ultimately attaches to.
 */
public class ActinFilament extends PointGridMember2 {

    /** the next actin filament toward the membrane (null if attached to membrane) */
    private ActinFilament upstream;

    /** the next actin filament away from the membrane (null if furthest from membrane) */
    private ActinFilament downstream;

    /** the length of the filament */
    private double length;

    /** the radius of the filament */
    private double radius;

    /** the force on the element */
    private final Vector2 force;

    /** working vector */
    private final Vector2 vec;

    /**
     * Constructs a new <code>ActinFilament</code>.
     *
     * @param  xCoord  the X coordinate of the pointed end
     * @param  yCoord  the Y coordinate of the pointed end
     * @param  len     the initial length of the filament
     * @param  rad     the initial radius of the filament
     * @param  prior   the prior (closer to the membrane) filament in the chain
     */
    public ActinFilament(final double xCoord, final double yCoord, final double len,
        final double rad, final ActinFilament prior) {

        super(xCoord, yCoord);

        this.length = len;
        this.radius = rad;

        this.upstream = prior;
        this.downstream = null;
        this.force = new Vector2();
```

```
        if (prior != null) {
            prior.setDownstream(this);
        }

        this.vec = new Vector2();
    }

    /**
     * Gets the radius to use when drawing the grid member.
     *
     * @return  the radius (0 to draw as a point)
     */
    @Override public double getRadius() {

        return this.radius;
    }

    /**
     * Gets this filament's downstream (further from the membrane) attached filament.
     *
     * @return  the downstream filament
     */
    public ActinFilament getDownstream() {

        return this.downstream;
    }

    /**
     * Sets this filament's downstream (further from the membrane) attached filament.
     *
     * @param  filament  the downstream filament
     */
    public void setDownstream(final ActinFilament filament) {

        this.downstream = filament;
    }

    /**
     * Gets this filament's upstream (closer to the membrane) attached filament.
     *
     * @return  the upstream filament
     */
    public ActinFilament getUpstream() {

        return this.upstream;
    }

    /**
     * Sets this filament's upstream (closer to the membrane) attached filament.
     *
     * @param  filament  the upstream filament
     */
    public void setUpstream(final ActinFilament filament) {

        this.upstream = filament;
    }

    /**
     * Gets the length of the filament.
     *
     * @return  the length
     */
    public double getLength() {

        return this.length;
    }

    /**
     * Sets the length of the filament
     *
     * @param  len  the new length
     */
    public void setLength(final double len) {

        this.length = len;
    }

    /**
     * Alters the length of the filament.
     *
     * @param  delta  the change to the length
     */
    public void adjustLength(final double delta) {

        this.length += delta;
        this.radius += delta;
    }
```

```
/**
 * Gets the force vector for the filament.
 *
 * @return  the force vector
 */
public Vector2 getForce() {

    return this.force;
}

/**
 * Computes the force on the element.
 */
public void computeInnerForce() {

    ActinFilament filament;
    double dist;
    double delta;

    // Force from connection to downstream actin, if any
    filament = this.downstream;

    if (filament != null) {
        dist = filament.dist(this);
        delta = dist - filament.getLength();
        this.vec.vectorBetween(this, filament);
        this.vec.normalize();
        this.vec.scaleVec(delta * Simulation.FILAMENT_SPRING);
        this.force.addVec(this.vec);
        filament.getForce().subVec(this.vec); // Don't compute again
    }
}

/**
 * Computes the force on the element due to interactions.
 *
 * @param  maxForce  the maximum force from non-interaction sources
 */
public void computeInteractionForce(final double maxForce) {

    double range;
    int count;
    GridMember2Int nbr;
    double dist;
    double scale;

    count = getNumNeighbors();

    for (int i = 0; i < count; i++) {
        nbr = getNeighbor(i);

        if (nbr instanceof MembraneElement) {

            range = Simulation.getInstance().getMaxSep() + getRadius();
            dist = nbr.dist(this);

            if (dist < range) {
                scale = Simulation.SS_FORCE.forceTimesEqDist(dist / range) / range;

                if (scale > maxForce) {
                    scale = maxForce;
                }

                vec.vectorBetween(nbr, this);
                vec.normalize();
                vec.scaleVec(scale);
                this.force.addVec(vec);
            }
        } else if (nbr instanceof ActinFilament) {
            range = nbr.getRadius() + getRadius();
            dist = nbr.dist(this);

            if (dist < (2.5 * range)) {
                scale = Simulation.ACTIN_FORCE1.forceTimesEqDist(dist / range) / range;

                if (scale > maxForce) {
                    scale = maxForce;
                }

                vec.vectorBetween(nbr, this);
                vec.normalize();
                vec.scaleVec(scale);
                this.force.addVec(vec);

                scale = Simulation.ACTIN_FORCE2.forceTimesEqDist(dist / range) / range;
```

```
                    if (scale > maxForce) {
                        scale = maxForce;
                    }

                    vec.vectorBetween(nbr, this);
                    vec.normalize();
                    vec.scaleVec(scale);
                    this.force.addVec(vec);
                }
            }
        }
    }
}

package com.srbenoit.modeling.cell;

import java.util.Random;
import com.srbenoit.geom.Point2;
import com.srbenoit.geom.Point2Int;
import com.srbenoit.geom.Vector2;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.modeling.grid.Grid2D;
import com.srbenoit.modeling.grid.GridMember2Int;

/**
 * A cell, consisting of a membrane and corresponding actin filaments.
 */
public class Cell extends LoggedObject {

    /** the number of membrane elements currently in the cell */
    private int numMembrane;

    /** an entry to the linked list of membrane elements */
    private MembraneElement membrane;

    /** the total amount of monomer in the cell */
    private final double monomerPool;

    /** the maximum radius of an actin segment */
    private final double maxActinRad;

    /** the length of a step in the actin polymerization process */
    private final double actinStep;

    /** the separation of actin from the membrane */
    private final double separation;

    /** the cell thickness, for volume computation */
    private final double thickness;

    /** the current cell volume */
    private double currVolume;

    /** the equilibrium cell volume */
    private double eqVolume;

    /** random number generator */
    private final Random rnd;

    /** a working vector */
    private final Vector2 vec;

    /** a working point */
    private final Point2 test;

    /**
     * Constructs a new <code>Cell</code>.
     *
     * @param  center           the center about which to build the cell
     * @param  radius           the cell radius
     * @param  numElements      the number of elements from which to build the membrane
     * @param  numActinSeg      the number of actin segments attached to each membrane element
     * @param  actinThickness   the thickness of the initial actin cortex
     * @param  avgSep           the average separation between elements
     * @param  maxMove          the maximum per-step movement of an element
     */
    public Cell(final Point2Int center, final double radius, final int numElements,
        final int numActinSeg, final double actinThickness, final double avgSep,
        final double maxMove) {

        double angle;
        double xPos;
        double yPos;
        MembraneElement elem;

        this.maxActinRad = actinThickness / (2.5 * numActinSeg);
        this.actinStep = maxMove / 2;
```

```
        this.rnd = new Random();
        this.numMembrane = numElements;
        this.separation = Simulation.getInstance().getMaxSep();

        // Empirical: 100 elements = sep/15, 200 = sep/1.5, 400 = sep / 0.15
        // Formula: denom = 66016350 N ^ -3.322
        this.thickness = avgSep / (66016350 * Math.pow(numElements, -3.322));

        // Create the first element in the list
        this.membrane = new MembraneElement(this, center.getPosX() + radius, center.getPosY(), 0,
                numActinSeg, this.maxActinRad);

        for (int i = 1; i < numElements; i++) {
            angle = 2 * Math.PI * i / numElements;
            xPos = center.getPosX() + (radius * Math.cos(angle));
            yPos = center.getPosY() + (radius * Math.sin(angle));

            elem = new MembraneElement(this, xPos, yPos, angle, numActinSeg, this.maxActinRad);
            elem.addBefore(this.membrane);
        }

        this.monomerPool = numElements * actinThickness;
        computeVolume();
        this.eqVolume = this.currVolume;

        this.vec = new Vector2();
        this.test = new Point2();
    }

    /**
     * Gets the head membrane element in the linked list.
     *
     * @return  the membrane element
     */
    public MembraneElement getMembrane() {

        return this.membrane;
    }

    /**
     * Adds all the objects that make up the cell to a grid, and stores the grid so new objects
     * created as the cell evolves can be automatically added to the grid, and objects deleted can
     * be removed.
     *
     * @param  grid  the grid
     */
    public void addToGrid(final Grid2D grid) {

        MembraneElement elem;
        ActinFilament actin;

        elem = this.membrane;

        for (;;) {
            elem.installInGrid(grid);

            for (int i = 0; i < elem.getNumActin(); i++) {
                actin = elem.getActin(i);

                while (actin != null) {
                    actin.installInGrid(grid);
                    actin = actin.getDownstream();
                }
            }

            elem = elem.getNext();

            if (elem == this.membrane) {
                break;
            }
        }
    }

    /**
     * Gets the equilibrium cell volume.
     *
     * @return  the equilibrium cell volume
     */
    public double getEquilibriumVolume() {

        return this.eqVolume;
    }

    /**
     * Gets the current cell volume.
     *
     * @return  the cell volume
```

```
 */
public double getCurrentVolume() {

    return this.currVolume;
}

/**
 * Gets the thickness of the cell.
 *
 * @return  the cell thickness
 */
public double getThickness() {

    return this.thickness;
}

/**
 * Computes the interior volume of the membrane.
 */
public final void computeVolume() {

    MembraneElement current;
    MembraneElement next;
    double vol;

    vol = 0;
    current = this.membrane;
    next = current.getNext();

    do {
        vol += (current.getPosX() * next.getPosY()) - (current.getPosY() * next.getPosX());

        current = next;
        next = current.getNext();

    } while (current != this.membrane);

    this.currVolume = this.thickness * vol * 0.5;
}

/**
 * React to signal levels in the membrane to cause actin polymerization.
 */
public void restructureActin() {

    MembraneElement elem;
    double act;
    double delta;
    double total;

    // Walk the membrane, extending filaments where activation is positive, and shortening
    // where it is negative, keeping track of total actin in use.

    total = 0;
    elem = this.membrane;

    do {
        act = elem.getActivation();

        if (act != 0) {

            if (act > 0) {
                delta = polymerize(elem, act); // returns a positive value
            } else {
                delta = depolymerize(elem, act); // returns a negative value
            }

            elem.adjustActivation(-delta);
            total += delta;
        }

        elem = elem.getNext();
    } while (elem != this.membrane);

    // If there was a net change, choose random membrane elements and have them polymerize
    // or depolymerize to make up the difference
    useUpMonomer(-total);
}

/**
 * Given an amount of monomer level to use up, either polymerizes (of the amount is positive),
 * or depolymerizes (if negative) until the amount is used up.
 *
 * @param  amout  the amount of monomer to use up
 */
private void useUpMonomer(final double amout) {
```

```
        MembraneElement elem;
        double remaining;
        int steps;

        elem = this.membrane;
        remaining = amout;

        while (remaining > 0) {
            steps = this.rnd.nextInt(this.numMembrane);

            for (int i = 0; i < steps; i++) {
                elem = elem.getNext();
            }

            remaining -= polymerize(elem, remaining); // returns a positive value
        }

        while (remaining < 0) {
            steps = this.rnd.nextInt(this.numMembrane);

            for (int i = 0; i < steps; i++) {
                elem = elem.getNext();
            }

            remaining -= depolymerize(elem, remaining); // returns a negative value
        }
    }

    /**
     * Polymerizes the actin below a membrane element.
     *
     * @param   elem        the membrane element at which to polymerize
     * @param   available   the available monomer to use (positive)
     * @return  the amount of monomer actually used (positive)
     */
    private double polymerize(final MembraneElement elem, final double available) {

        ActinFilament actin;
        ActinFilament newActin;
        double xPos;
        double yPos;
        double used;
        double angle;
        double dist;
        int count;
        GridMember2Int nbr;
        boolean good;

        xPos = elem.getPosX();
        yPos = elem.getPosY();

        if (elem.getNumActin() == 0) {

            // There is no actin at the membrane position, so we make a new very small one,
            // positioned on the inside of the membrane at a point that allows it (if such a
            // point exists)
            if (available >= (this.actinStep / 3)) {
                used = this.actinStep / 3;
            } else {
                used = available;
            }

            // Identify an open point at which to insert a new filament.  The best-case is the
            // direction opposite the outward-pointing normal, which is:
            angle = Math.atan2(-elem.getVecY(), -elem.getVecX());
            dist = Simulation.getInstance().getMaxSep();
            count = elem.getNumNeighbors();
            good = false;

            for (double trial = 0; trial < (Math.PI / 2); trial += 0.01) {
                test.setPos(xPos + (dist * Math.cos(angle + trial)),
                    yPos + (dist * Math.sin(angle + trial)));

                good = true;

                for (int i = 0; i < count; i++) {
                    nbr = elem.getNeighbor(i);

                    if (nbr.dist(test) < nbr.getRadius()) {
                        good = false;

                        break;
                    }
                }

                if (good) {
                    actin = new ActinFilament(test.getPosX(), test.getPosY(),
```

```
                        this.separation + used, used, null);
                actin.installInGrid(elem.getGrid());
                elem.addActin(actin);

                break;
            }

            test.setPos(xPos + (dist * Math.cos(angle - trial)),
                yPos + (dist * Math.sin(angle - trial)));

            good = true;

            for (int i = 0; i < count; i++) {
                nbr = elem.getNeighbor(i);

                if (nbr.dist(test) < nbr.getRadius()) {
                    good = false;

                    break;
                }
            }

            if (good) {
                actin = new ActinFilament(test.getPosX(), test.getPosY(),
                        this.separation + used, used, null);
                actin.installInGrid(elem.getGrid());
                elem.addActin(actin);

                break;
            }
        }

        if (!good) {
            // Could not place actin without overlapping something
            used = 0;
        }

    } else {
        actin = elem.getActin(0); // We only polymerize the first actin on a node

        if (actin.getRadius() >= this.maxActinRad) {

            // Need to spawn a new filament from the end of the existing one
            if (available >= (this.actinStep / 3)) {
                used = this.actinStep / 3;
            } else {
                used = available;
            }

            vec.vectorBetween(elem, actin);
            vec.normalize();
            vec.scaleVec(this.separation + used);
            newActin = new ActinFilament(xPos + vec.getVecX(), yPos + vec.getVecY(),
                    this.separation + used, used, null);
            newActin.installInGrid(elem.getGrid());
            actin.setUpstream(newActin);
            actin.setLength(actin.getRadius() + used);
            newActin.setDownstream(actin);
            elem.replaceActin(0, newActin);
        } else {

            // Grow the actin filament that's there one step
            if (available >= this.actinStep) {
                used = this.actinStep;
            } else {
                used = available;
            }

            actin.adjustLength(used);

            if (actin.getDownstream() != null) {
                actin.getDownstream().setLength(actin.getDownstream().getLength() + used);
            }
        }
    }

    return used;
}

/**
 * Depolymerizes the actin below a membrane element.
 *
 * @param   elem        the membrane element at which to depolymerize
 * @param   available   the available monomer to use (negative)
 * @return  the amount of monomer actually used (negative)
 */
private double depolymerize(final MembraneElement elem, final double available) {
```

```
        int num;
        ActinFilament head;
        ActinFilament actin;
        double used;

        // Find the most downstream filament attached to this membrane element

        num = elem.getNumActin();

        if (num == 0) {
            used = 0;
        } else {
            head = elem.getActin(num - 1); // We only depolymerize the last actin on a node

            actin = head;

            while (actin.getDownstream() != null) {
                actin = actin.getDownstream();
            }

            if (available < -this.actinStep) {
                used = -this.actinStep;
            } else {
                used = available;
            }

            // Shorten the actin by one step, delete it if it vanishes
            if ((actin.getRadius() + used) > 0) {
                actin.adjustLength(used);
            } else {
                used = -actin.getRadius();
                actin.removeFromGrid();

                if (actin == head) {
                    elem.removeActin(actin);
                } else {
                    actin.getUpstream().setDownstream(null);
                }
            }
        }

        return used;
    }

    /**
     * Restructures the membrane by adding elements where two elements are too far apart or
     * deleting elements where two are too close together.
     *
     * @param  minSep  the minimum separation between elements
     * @param  maxSep  the maximum separation between elements
     */
    public void restructureMembrane(final double minSep, final double maxSep) {

        MembraneElement start;
        MembraneElement elem;
        MembraneElement next;
        double dist;

        elem = this.membrane;
        start = elem;

        do {
            next = elem.getNext();

            dist = elem.dist(next);

            if (dist > maxSep) {
                subdivideEdge(elem, next);
                start = elem;
            } else if (dist < minSep) {
//                mergeElements(elem, next);
//                start = elem;
            }

            elem = elem.getNext();
        } while (elem != start);
    }

    /**
     * Subdivides the edge between two membrane elements, adding a new element at the midpoint.
     *
     * @param  elem  the first element
     * @param  next  the next element
     */
    private void subdivideEdge(final MembraneElement elem, final MembraneElement next) {
```

```
        MembraneElement newElem;

        // Split the edge in half at the midpoint, adding a new element, whose normal is
        // the average of the normals at the endpoints.
        this.vec.setVec(elem);
        this.vec.addVec(next);
        this.vec.normalize();

        newElem = new MembraneElement(this, (elem.getPosX() + next.getPosX()) / 2,
                (elem.getPosY() + next.getPosY()) / 2, this.vec.getVecX(), this.vec.getVecY(), 0,
                0);
        newElem.installInGrid(elem.getGrid());
        newElem.addAfter(elem);

        this.numMembrane++;
    }

    /**
     * Merges two membrane elements that are too close together.
     *
     * @param   elem  the first element
     * @param   next  the next element
     * @return  the amount of monomer released by the merged element (positive)
     */
    private void mergeElements(final MembraneElement elem, final MembraneElement next) {

        ActinFilament actin;

        // Move any actin filaments under 'next' to 'elem'
        while (next.getNumActin() > 0) {
            actin = next.getActin(0);
            next.removeActin(actin);
            elem.addActin(actin);
        }

        // Move 'elem' to the midpoint
        vec.vectorBetween(elem, next);
        vec.scaleVec(0.5);
        elem.move(vec);

        // Remove 'next' from the membrane.  If 'next' happens to be the 'membrane' member
        // variable, we make 'elem' the new membrane head.
        if (this.membrane == next) {
            this.membrane = elem;
        }

        next.remove();
        next.removeFromGrid();

        this.numMembrane--;
    }

    /**
     * Compute the force on every element in the model.
     */
    public void computeInnerForces() {

        MembraneElement elem;
        int count;
        ActinFilament actin;

        computeVolume();

        // Zero out all forces first
        elem = this.membrane;

        do {
            elem.getForce().setVec(0, 0);

            count = elem.getNumActin();

            for (int i = 0; i < count; i++) {
                actin = elem.getActin(i);

                while (actin != null) {
                    actin.getForce().setVec(0, 0);
                    actin = actin.getDownstream();
                }
            }

            elem = elem.getNext();
        } while (elem != this.membrane);

        // Now compute all forces
        elem = this.membrane;

        do {
```

```
            elem.computeInnerForce();

            count = elem.getNumActin();

            for (int i = 0; i < count; i++) {
                actin = elem.getActin(i);

                while (actin != null) {
                    actin.computeInnerForce();
                    actin = actin.getDownstream();
                }
            }

            elem = elem.getNext();
        } while (elem != this.membrane);
    }

    /**
     * Computes the maximum force on any element in the cell.
     *
     * @return  the maximum force
     */
    public double maxForce() {

        MembraneElement elem;
        int count;
        ActinFilament actin;
        double force;
        double maxForce;

        // Determine the largest force in the system
        maxForce = 0;
        elem = this.membrane;

        do {
            force = elem.getForce().length();

            if (force > maxForce) {
                maxForce = force;
            }

            count = elem.getNumActin();

            for (int i = 0; i < count; i++) {
                actin = elem.getActin(i);

                while (actin != null) {
                    force = actin.getForce().length();

                    if (force > maxForce) {
                        maxForce = force;
                    }

                    actin = actin.getDownstream();
                }
            }

            elem = elem.getNext();
        } while (elem != this.membrane);

        return maxForce;
    }

    /**
     * Compute the force on every element in the model.
     *
     * @param  maxForce  the maximum force from non-interaction sources
     */
    public void computeInteractionForces(final double maxForce) {

        MembraneElement elem;
        int count;
        ActinFilament actin;

        elem = this.membrane;

        do {
            elem.computeInteractionForce(maxForce);

            count = elem.getNumActin();

            for (int i = 0; i < count; i++) {
                actin = elem.getActin(i);

                while (actin != null) {
                    actin.computeInteractionForce(maxForce);
                    actin = actin.getDownstream();
```

```
                }
            }

            elem = elem.getNext();
        } while (elem != this.membrane);
    }

    /**
     * Move elements in the direction of their forces.
     *
     * @param  mobility  the mobility used to convert force into motion
     */
    public void reactToForces(final double mobility) {

        MembraneElement elem;
        int count;
        ActinFilament actin;
        Vector2 force;

        elem = this.membrane;

        do {
            force = elem.getForce();
            elem.move(force.getVecX() * mobility, force.getVecY() * mobility);

            count = elem.getNumActin();

            for (int i = 0; i < count; i++) {
                actin = elem.getActin(i);

                while (actin != null) {
                    force = actin.getForce();
                    actin.move(force.getVecX() * mobility, force.getVecY() * mobility);
                    actin = actin.getDownstream();
                }
            }

            elem = elem.getNext();
        } while (elem != this.membrane);
    }

    /**
     * Computes the normal vector at each membrane element based on its neighbors.
     */
    public void recomputeNormals() {

        MembraneElement elem;

        elem = this.membrane;

        do {
            elem.recomputeNormal();
            elem = elem.getNext();
        } while (elem != this.membrane);
    }
}

package com.srbenoit.modeling.cell;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.event.ComponentEvent;
import java.awt.event.ComponentListener;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Rectangle2D;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;
import java.util.logging.Level;
import javax.imageio.ImageIO;
import javax.imageio.ImageWriter;
import javax.imageio.stream.ImageOutputStream;
import com.srbenoit.log.LoggedPanel;

/**
 * A panel that can render cells.
 */
public class CellPanel extends LoggedPanel implements ComponentListener {

    /** version number for serialization */
```

```
private static final long serialVersionUID = 5666568382362507929L;

/** object on which to synchronize access to member variables */
private final Object synch;

/** the bounding rectangle after adjustment to match window's aspect ratio */
private final Rectangle2D adjBounds;

/** the current width of the offscreen images */
private int width;

/** the current height of the offscreen images */
private int height;

/** the most recently drawn image */
private BufferedImage active;

/** an image to which the next render should be directed */
private BufferedImage passive;

/** temporary storage for an incorrectly sized image rendered while size was changing */
private BufferedImage temp;

/** a color for actin */
private final static Color actinColor = new Color(0, 120, 0);

/**
 * Constructs a new <code>CellPanel</code>.
 *
 * @param  width    the preferred width of the window
 * @param  height   the preferred height of the window
 */
public CellPanel(final int width, final int height) {

    super();

    this.synch = new Object();

    setBackground(Color.WHITE);
    setPreferredSize(new Dimension(width, height));
    setSize(width, height);
    this.adjBounds = new Rectangle2D.Double();

    this.active = null;
    this.passive = null;
    this.temp = null;
    this.width = 0;
    this.height = 0;
    updateOffscreenImages();

    addComponentListener(this);
}

/**
 * Tests the current size of the panel against the offscreen image sizes, and if needed,
 * creates new offscreen images. This is done in a synchronized block so a completing render
 * cannot stomp an image we create, and this method will not stomp the output of a render.
 */
private void updateOffscreenImages() {

    int actWidth;
    int actHeight;
    Graphics grx;

    actWidth = getWidth();
    actHeight = getHeight();

    synchronized (this.synch) {

        if ((actWidth != this.width) || (actHeight != this.height)) {
            this.active = new BufferedImage(actWidth, actHeight, BufferedImage.TYPE_INT_RGB);
            this.passive = new BufferedImage(actWidth, actHeight, BufferedImage.TYPE_INT_RGB);

            this.width = actWidth;
            this.height = actHeight;

            grx = this.active.getGraphics();
            grx.setColor(Color.WHITE);
            grx.fillRect(0, 0, this.width, this.height);

            grx = this.passive.getGraphics();
            grx.setColor(Color.WHITE);
            grx.fillRect(0, 0, this.width, this.height);
        }
    }
}
```

```
/**
 * Draws the panel contents using the most up-to-date rendered image available.
 *
 * @param  grx  the <code>graphics</code> to which to draw.
 */
@Override public void paintComponent(final Graphics grx) {

    int actWidth;
    int actHeight;
    double xScale;
    double yScale;
    double scale;
    int scaledWidth;
    int scaledHeight;

    synchronized (this.synch) {

        if (this.temp == null) {

            if (this.active != null) {
                grx.drawImage(this.active, 0, 0, null);
            }

        } else {
            actWidth = getWidth();
            actHeight = getHeight();
            xScale = actWidth / this.temp.getWidth();
            yScale = actHeight / this.temp.getHeight();
            scale = (xScale > yScale) ? xScale : yScale;
            scaledWidth = (int) ((this.temp.getWidth() * scale) + 0.5); // at least actWidth
            scaledHeight = (int) ((this.temp.getHeight() * scale) + 0.5); // at least actHeight
            grx.drawImage(this.temp, (actWidth - scaledWidth) / 2,
                (actHeight - scaledHeight) / 2, scaledWidth, scaledHeight, null);
        }
    }
}

/**
 * Handles panel resize events, which require allocation of new offscreen images.
 *
 * @param  evt  the component event
 */
public void componentResized(final ComponentEvent evt) {

    updateOffscreenImages();
    repaint();
}

/**
 * Handles panel move events, which we ignore.
 *
 * @param  evt  the component event
 */
public void componentMoved(final ComponentEvent evt) {
    // No action
}

/**
 * Handles panel shown events, which causes a test for actual size against current size of the
 * offscreen images, and may result in allocation of new offscreen images.
 *
 * <p>NOTE: Rendering is done outside the AWT event loop, but writes to the offscreen images.
 * At the end of a render process, if the rendered image is not the size specified in <code>
 * width</code> and <code>height</code>, we store the image in the <code>temp</code> member
 * variable. Otherwise, the end of the render cycle stores the result as the active image, and
 * clears the <code>temp</code> member. When a repaint is requested, if the <code>temp</code>
 * object has an image, that image is drawn, but scaled to fit the window.
 *
 * @param  evt  the component event
 */
public void componentShown(final ComponentEvent evt) {

    updateOffscreenImages();
}

/**
 * Handles panel hidden events, which we ignore.
 *
 * @param  evt  the component event
 */
public void componentHidden(final ComponentEvent evt) {
    // No action
}

/**
 * Updates the offscreen bitmap.
 *
```

```
     * @param  bounds     the bounds of model space to render
     * @param  cells      the list of cells to draw
     * @param  emitters   the list of emitters to draw
     * @param  walls      the list of walls to draw
     */
    public void render(final Rectangle2D bounds, final List<Cell> cells,
        final List<Emitter> emitters, final List<Wall> walls) {

        BufferedImage target;

        synchronized (this.synch) {
            target = this.passive;
        }

        if (target != null) {

            renderScene(bounds, target, cells, emitters, walls);

            synchronized (this.synch) {

                if (this.passive == target) {

                    // Images have not been reallocated, so just flip buffers
                    this.passive = this.active;
                    this.active = target;
                    this.temp = null;
                } else {

                    // A resize has resulted in new images, so place our results in "temp"
                    this.temp = target;
                }
            }
        }

        repaint();
    }

    /**
     * Draws the cells on the target image.
     *
     * @param  bounds     the bounds of model space to render
     * @param  target     the <code>BufferedImage</code> on which to draw.
     * @param  cells      the list of cells to render
     * @param  emitters   the list of emitters to render
     * @param  walls      the list of walls to render
     */
    private void renderScene(final Rectangle2D bounds, final BufferedImage target,
        final List<Cell> cells, final List<Emitter> emitters, final List<Wall> walls) {

        Graphics2D grx;
        int imgWidth;
        int imgHeight;
        double pixelScale;

        // Get the target size and adjust the bounds for the correct aspect ratio
        imgWidth = target.getWidth();
        imgHeight = target.getHeight();
        adjustBounds(imgWidth, imgHeight, bounds);
        pixelScale = imgWidth / adjBounds.getWidth();

        // Clear the image for rendering
        grx = (Graphics2D) target.getGraphics();
        grx.setColor(Color.WHITE);
        grx.fillRect(0, 0, imgWidth, imgHeight);
        grx.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
        grx.setStroke(new BasicStroke(0.4f));

        renderCells(imgWidth, imgHeight, grx, pixelScale, cells);
        renderEmitters(imgWidth, imgHeight, grx, pixelScale, emitters);
        renderWalls(imgWidth, imgHeight, grx, pixelScale, walls);
    }

    /**
     * Adjust the bound rectangle to have the proper target aspect ratio. The result is stored in
     * <code>adjBounds</code>.
     *
     * @param  imgWidth   the image width
     * @param  imgHeight  the image height
     * @param  bounds     the bounding rectangle
     */
    private void adjustBounds(final int imgWidth, final int imgHeight, final Rectangle2D bounds) {

        double aspect;
        double rectAspect;
        double newSize;
        double delta;
```

```
        aspect = (double) imgWidth / (double) imgHeight;
        rectAspect = bounds.getWidth() / bounds.getHeight();

        if (aspect > rectAspect) {

            // Make bounds wider to match aspect ratio
            newSize = bounds.getWidth() * (aspect / rectAspect);
            delta = (newSize - bounds.getWidth()) / 2;
            this.adjBounds.setFrame(bounds.getX() - delta, bounds.getY(), newSize,
                bounds.getHeight());
        } else {

            // Make bounds taller to match aspect ratio
            newSize = bounds.getHeight() * (rectAspect / aspect);
            delta = (newSize - bounds.getHeight()) / 2;
            this.adjBounds.setFrame(bounds.getX(), bounds.getY() - delta, bounds.getWidth(),
                newSize);
        }
    }

    /**
     * Renders the cells.
     *
     * @param  imgWidth     the width of the image to which to render
     * @param  imgHeight    the height of the image to which to render
     * @param  grx          the <code>Graphics2D</code> to which to draw
     * @param  pixelScale   the pixel scale (pixels per unit in model space)
     * @param  cells        the list of cells to render
     */
    private void renderCells(final int imgWidth, final int imgHeight, final Graphics2D grx,
        final double pixelScale, final List<Cell> cells) {

        MembraneElement mem;
        MembraneElement next;
        double xPix;
        double yPix;
        double xPix2;
        double yPix2;
        double xPos;
        double yPos;
        Ellipse2D ellipse;
        Line2D line;
        int count;
        ActinFilament actin;
        double rad;

        ellipse = new Ellipse2D.Double();
        line = new Line2D.Double();

        // Render the cells
        for (Cell cell : cells) {

            mem = cell.getMembrane();
            next = mem.getNext();

            for (;;) {
                xPos = mem.getPosX();
                yPos = mem.getPosY();

                xPix = (xPos - adjBounds.getX()) * pixelScale;
                yPix = imgHeight - ((yPos - adjBounds.getY()) * pixelScale);
                xPix2 = (next.getPosX() - adjBounds.getX()) * pixelScale;
                yPix2 = imgHeight - ((next.getPosY() - adjBounds.getY()) * pixelScale);

                // Draw the membrane element point
                grx.setColor(Color.BLUE);
                ellipse.setFrame(xPix - 1, yPix - 1, 2, 2);
                grx.fill(ellipse);
                line.setLine(xPix, yPix, xPix2, yPix2);
                grx.draw(line);

                // Draw the actin filament
                count = mem.getNumActin();

                for (int i = 0; i < count; i++) {
                    actin = mem.getActin(i);

                    while (actin != null) {

                        xPos = actin.getPosX();
                        yPos = actin.getPosY();
                        xPix2 = (xPos - adjBounds.getX()) * pixelScale;
                        yPix2 = imgHeight - ((yPos - adjBounds.getY()) * pixelScale);
                        rad = actin.getRadius() * pixelScale;

                        grx.setColor(actinColor);
                        ellipse.setFrame(xPix2 - rad, yPix2 - rad, 2 * rad, 2 * rad);
```

```
                    grx.draw(ellipse);
                    line.setLine(xPix, yPix, xPix2, yPix2);
                    grx.draw(line);

                    xPix = xPix2;
                    yPix = yPix2;

                    actin = actin.getDownstream();
                }
            }

            mem = mem.getNext();
            next = mem.getNext();

            if (mem == cell.getMembrane()) {
                break;
            }
        }
    }
}

/**
 * Renders the emitters.
 *
 * @param  imgWidth     the width of the image to which to render
 * @param  imgHeight    the height of the image to which to render
 * @param  grx          the <code>Graphics2D</code> to which to draw
 * @param  pixelScale   the pixel scale (pixels per unit in model space)
 * @param  emitters     the list of emitters to render
 */
private void renderEmitters(final int imgWidth, final int imgHeight, final Graphics2D grx,
    final double pixelScale, final List<Emitter> emitters) {

    double xPix;
    double yPix;
    double xPos;
    double yPos;
    double xPix2;
    double yPix2;
    double xPos2;
    double yPos2;
    Ellipse2D ellipse;
    Line2D line;
    int count;
    Signal sig;
    FixedElement fixed;
    FixedElement next;

    ellipse = new Ellipse2D.Double();
    line = new Line2D.Double();

    // Render the emitters
    for (Emitter emitter : emitters) {

        xPos = emitter.getPosX();
        yPos = emitter.getPosY();

        xPix = (xPos - adjBounds.getX()) * pixelScale;
        yPix = imgHeight - ((yPos - adjBounds.getY()) * pixelScale);

        // Draw the emitter point
        grx.setColor(Color.MAGENTA);
        ellipse.setFrame(xPix - 2, yPix - 2, 4, 4);
        grx.fill(ellipse);

        // Draw the fixed ring
        grx.setColor(Color.GRAY);
        count = emitter.getNumFixed();

        for (int i = 0; i < count; i++) {
            fixed = emitter.getFixed(i);
            next = emitter.getFixed((i + 1) % count);

            xPos = fixed.getPosX();
            yPos = fixed.getPosY();

            xPix = (xPos - adjBounds.getX()) * pixelScale;
            yPix = imgHeight - ((yPos - adjBounds.getY()) * pixelScale);

            xPos2 = next.getPosX();
            yPos2 = next.getPosY();

            xPix2 = (xPos2 - adjBounds.getX()) * pixelScale;
            yPix2 = imgHeight - ((yPos2 - adjBounds.getY()) * pixelScale);

            ellipse.setFrame(xPix - 1, yPix - 1, 2, 2);
            grx.fill(ellipse);
```

```
                line.setLine(xPix, yPix, xPix2, yPix2);
                grx.draw(line);
            }

            // Draw the signals
            grx.setColor(Color.GRAY);
            count = emitter.getNumSignals();

            for (int i = 0; i < count; i++) {
                sig = emitter.getSignal(i);

                xPos = sig.getPosX();
                yPos = sig.getPosY();

                xPix = (xPos - adjBounds.getX()) * pixelScale;
                yPix = imgHeight - ((yPos - adjBounds.getY()) * pixelScale);

                ellipse.setFrame(xPix, yPix, 1, 1);
                grx.fill(ellipse);
            }
        }
    }

    /**
     * Renders the emitters.
     *
     * @param  imgWidth     the width of the image to which to render
     * @param  imgHeight    the height of the image to which to render
     * @param  grx          the <code>Graphics2D</code> to which to draw
     * @param  pixelScale   the pixel scale (pixels per unit in model space)
     * @param  walls        the list of walls to render
     */
    private void renderWalls(final int imgWidth, final int imgHeight, final Graphics2D grx,
        final double pixelScale, final List<Wall> walls) {

        double xPix;
        double yPix;
        double xPos;
        double yPos;
        double xPix2;
        double yPix2;
        double xPos2;
        double yPos2;
        Ellipse2D ellipse;
        Line2D line;
        int count;
        Signal sig;
        FixedElement fixed;
        FixedElement next;

        ellipse = new Ellipse2D.Double();
        line = new Line2D.Double();

        for (Wall wall : walls) {

            grx.setColor(Color.GRAY);
            count = wall.getNumFixed();

            for (int i = 0; i < count; i++) {
                fixed = wall.getFixed(i);

                xPos = fixed.getPosX();
                yPos = fixed.getPosY();
                xPix = (xPos - adjBounds.getX()) * pixelScale;
                yPix = imgHeight - ((yPos - adjBounds.getY()) * pixelScale);

                ellipse.setFrame(xPix - 1, yPix - 1, 2, 2);
                grx.fill(ellipse);

                if ((i + 1) < count) {
                    next = wall.getFixed(i + 1);
                    xPos2 = next.getPosX();
                    yPos2 = next.getPosY();
                    xPix2 = (xPos2 - adjBounds.getX()) * pixelScale;
                    yPix2 = imgHeight - ((yPos2 - adjBounds.getY()) * pixelScale);

                    line.setLine(xPix, yPix, xPix2, yPix2);
                    grx.draw(line);
                }
            }
        }
    }

    /**
     * Exports a single frame of animation to a JPEG file.
     *
     * @param  frameNum  The integer frame number
```

```
     */
    public void exportFrame(final int frameNum) {

        File file;
        ImageWriter writer = null;
        Iterator<?> writers;
        ImageOutputStream ios;

        file = new File("/imp/frames/frame_" + Integer.toString(frameNum / 1000)
                + Integer.toString((frameNum / 100) % 10) + Integer.toString((frameNum / 10) % 10)
                + Integer.toString(frameNum % 10) + ".png");

        synchronized (this.synch) {
            writers = ImageIO.getImageWritersByFormatName("PNG");

            if (!writers.hasNext()) {
                LOG.warning("No PNG Writers Available");

                return;
            }

            writer = (ImageWriter) writers.next();

            if (file.exists()) {
                file.delete();
            }

            try {
                ios = ImageIO.createImageOutputStream(file);
                writer.setOutput(ios);
                writer.write(this.active);
                ios.flush();
                writer.dispose();
                ios.close();
            } catch (IOException e) {
                LOG.log(Level.WARNING, "Exception writing frame", e);
            }
        }
    }
}
```

```
package com.srbenoit.modeling.cell;

import java.awt.geom.Rectangle2D;
import java.util.Random;
import com.srbenoit.geom.Vector2;
import com.srbenoit.modeling.grid.Grid2D;
import com.srbenoit.modeling.grid.PointGridMember2;

/**
 * An emitter of diffusing signals.
 */
public class Emitter extends PointGridMember2 {

    /** the fixed elements */
    private FixedElement[] fixed;

    /** the activation level of signals this emitter generates */
    private double activation;

    /** the lifetime of signals this emitter generates */
    private int lifetime;

    /** the diffusion step length */
    private final double stepLength;

    /** the number of signals from this emitter that are currently active */
    private int numSignals;

    /** the active signals that have been emitted from this emitter. */
    private Signal[] signals;

    /** random generator for diffusion */
    private final Random rnd;

    /** the force on the element */
    private final Vector2 force;

    /** a constant external force on the element */
    private final Vector2 handOfGodForce;

    /**
     * Constructs a new <code>Emitter</code>.
     *
     * @param  xCoord        the X coordinate of the emitter
     * @param  yCoord        the Y coordinate of the emitter
     * @param  actLevel      the activation level of signals this emitter will generate
```

```
     * @param   life          the lifetime of the emitted signals
     * @param   diffStep      the diffusion step length (for random walk)
     * @param   radius        the ring radius
     * @param   numElements   the number of elements from which to build the ring
     */
    public Emitter(final double xCoord, final double yCoord, final double actLevel, final int life,
        final double diffStep, final double radius, final int numElements) {

        super(xCoord, yCoord);

        double angle;
        double xPos;
        double yPos;

        this.activation = actLevel;
        this.lifetime = life;
        this.stepLength = diffStep;

        this.signals = new Signal[100];
        this.numSignals = 0;

        this.rnd = new Random();
        this.force = new Vector2();
        this.handOfGodForce = new Vector2();

        this.fixed = new FixedElement[numElements];

        for (int i = 0; i < numElements; i++) {
            angle = 2 * Math.PI * i / numElements;
            xPos = xCoord + (radius * Math.cos(angle));
            yPos = yCoord + (radius * Math.sin(angle));

            this.fixed[i] = new FixedElement(xPos, yPos);
        }
    }

    /**
     * Gets the force vector for the filament.
     *
     * @return  the force vector
     */
    public Vector2 getForce() {

        return this.force;
    }

    /**
     * Gets the external "hand of God" force vector for the filament.
     *
     * @return  the hand of God force vector
     */
    public Vector2 getHandOfGod() {

        return this.handOfGodForce;
    }

    /**
     * Moves the emitter and its associated fixed elements. Active signals are not affected.
     *
     * @param   deltaX  the X component by which to move the emitter
     * @param   deltaY  the Y component by which to move the emitter
     */
    @Override public void move(final double deltaX, final double deltaY) {

        super.move(deltaX, deltaY);

        for (int i = 0; i < this.fixed.length; i++) {
            this.fixed[i].move(deltaX, deltaY);
        }
    }

    /**
     * Gets the number of active signals emitted by the emitter.
     *
     * @return  the number of signals
     */
    public int getNumSignals() {

        return this.numSignals;
    }

    /**
     * Gets a particular signal.
     *
     * @param   index  the index of the signal
     * @return  the signal
     */
```

```
public Signal getSignal(final int index) {

    return this.signals[index];
}

/**
 * Gets the number of fixed elements.
 *
 * @return  the number of fixed elements
 */
public int getNumFixed() {

    return this.fixed.length;
}

/**
 * Gets a particular fixed element.
 *
 * @param   index  the index of the fixed element
 * @return  the fixed element
 */
public FixedElement getFixed(final int index) {

    return this.fixed[index];
}

/**
 * Emits new signals.
 *
 * @param  numToEmit  the number of signals to emit.
 */
public void emit(final int numToEmit) {

    Signal[] newArray;
    Signal sig;

    if (this.signals.length < (this.numSignals + numToEmit)) {
        newArray = new Signal[this.signals.length + 100];
        System.arraycopy(this.signals, 0, newArray, 0, this.numSignals);
        this.signals = newArray;
    }

    for (int i = 0; i < numToEmit; i++) {
        sig = new Signal(getPosX(), getPosY(), this.lifetime, this.activation);

        if (getGrid() != null) {
            sig.installInGrid(getGrid());
        }

        this.signals[this.numSignals] = sig;
        this.numSignals++;
    }
}

/**
 * Ages all active signals and culls those that have expired.
 */
public void age() {

    Signal sig;
    int remain;

    for (int i = 0; i < this.numSignals;) {

        sig = this.signals[i];
        remain = sig.age();

        if (remain <= 0) {

            if (sig.getGrid() != null) {
                sig.removeFromGrid();
            }

            this.signals[i] = this.signals[this.numSignals - 1];
            this.numSignals--;
        } else {
            i++;
        }
    }
}

/**
 * Allows all active signals to diffuse (this does not age them), and culls any that leave a
 * bounding rectangle.
 *
 * @param  bounds  the bounding rectangle
 */
```

```
    public void diffuse(final Rectangle2D bounds) {

        Signal sig;
        double angle;

        for (int i = 0; i < this.numSignals;) {

            angle = this.rnd.nextDouble() * 2 * Math.PI;

            sig = this.signals[i];
            sig.move(this.stepLength * Math.cos(angle), this.stepLength * Math.sin(angle));

            if (bounds.contains(sig.getPosX(), sig.getPosY())) {
                i++;
            } else {
                this.signals[i] = this.signals[this.numSignals - 1];
                this.numSignals--;
            }
        }
    }

    /**
     * Adds the emitter and any active signals to a grid, and stores the grid so emitted signals
     * can be automatically added to the grid and expiring signals can be removed.
     *
     * @param  grid  the grid
     */
    public void addToGrid(final Grid2D grid) {

        installInGrid(grid);

        for (int i = 0; i < this.numSignals;) {
            this.signals[i].installInGrid(grid);
        }

        for (FixedElement elem : this.fixed) {
            elem.installInGrid(grid);
        }
    }

    /**
     * Compute the force on every element in the model.
     */
    public void computeInnerForces() {

        for (FixedElement elem : this.fixed) {
            elem.getForce().setVec(this.handOfGodForce);
        }
    }

    /**
     * Compute the force on every element in the model.
     *
     * @param  maxForce the maximum force from non-interaction sources
     */
    public void computeInteractionForces(final double maxForce) {

        for (FixedElement elem : this.fixed) {
            elem.computeInteractionForce(maxForce);
        }

        // Sum the forces and distribute the average force to all fixed elements
        this.force.setVec(0,0);
        for (FixedElement elem : this.fixed) {
            this.force.addVec(elem.getForce());
        }

        this.force.scaleVec(1.0 / this.fixed.length);

        for (FixedElement elem : this.fixed) {
            elem.getForce().setVec(this.force);
        }
    }

    /**
     * Move elements in the direction of their forces.
     *
     * @param  mobility  the mobility used to convert force into motion
     */
    public void reactToForces(final double mobility) {

        move(this.force.getVecX() * mobility, this.force.getVecY() * mobility);
    }
}

package com.srbenoit.modeling.cell;
```

```
import com.srbenoit.geom.Vector2;
import com.srbenoit.modeling.grid.GridMember2Int;
import com.srbenoit.modeling.grid.PointGridMember2;

/**
 * A fixed element, which does not move, but which can exert soft-sphere forces on other elements.
 */
public class FixedElement extends PointGridMember2 {

    /** the force on the element */
    private final Vector2 force;

    /** working tuple */
    private final Vector2 vec;

    /**
     * Constructs a new <code>FixedElement</code>.
     *
     * @param  xCoord  the X coordinate of the element
     * @param  yCoord  the Y coordinate of the element
     */
    public FixedElement(final double xCoord, final double yCoord) {

        super(xCoord, yCoord);
        this.force = new Vector2();
        this.vec = new Vector2();
    }

    /**
     * Gets the force vector for the filament.
     *
     * @return  the force vector
     */
    public Vector2 getForce() {

        return this.force;
    }

    /**
     * Computes the force on the element.
     *
     * @param  maxForce  the maximum force from non-interaction sources
     */
    public void computeInteractionForce(final double maxForce) {

        GridMember2Int nbr;
        double scale;
        double dist;
        int count;
        double range;

        // Interaction force with membrane or fixed elements
        range = 1.5 * Simulation.getInstance().getMaxSep();
        count = getNumNeighbors();

        for (int i = 0; i < count; i++) {
            nbr = getNeighbor(i);

            if ((nbr instanceof MembraneElement) || (nbr instanceof FixedElement)) {
                dist = nbr.dist(this);

                if (dist < range) {
                    scale = Simulation.SS_FORCE.forceTimesEqDist(dist / range) / range;

                    if (scale > maxForce) {
                        scale = maxForce;
                    }

                    vec.vectorBetween(nbr, this);
                    vec.normalize();
                    vec.scaleVec(scale);
                    this.force.addVec(vec);
                }
            }
        }
    }
}

package com.srbenoit.modeling.cell;

import java.util.ArrayList;
import java.util.List;
import com.srbenoit.geom.Vector2;
import com.srbenoit.modeling.grid.BasedVectorGridMember2;
import com.srbenoit.modeling.grid.GridMember2Int;

/**
```

```
 * An element of a membrane, with the corresponding actin filament. Elements are maintained in a
 * doubly linked list.
 *
 * <p>The equilibrium length of the link to the next element is maintained here, and can adjust
 * slowly based on tension. If this value dips below a minimum, the element will be merged with its
 * neighbor, and it it exceeds a maximum, a new element is inserted between this element and its
 * neighbor. This equilibrium length also applies to the separation between the actin filament of
 * this element and that of the next element, although with a different spring constant.
 */
public class MembraneElement extends BasedVectorGridMember2 {

    /** the maximum activation level */
    private static final int MAX_ACTIVATION = 4;

    /** elastic modulus (microgram micron^2 / microsecond^2) */
    private static final double ELASTICMOD = 7e-11;

    /** tension (microgram / microsecond^2) */
    private static final double TENSION = 3e-8;

    /** cytoplasm bulk modulus (microgram / (micron microsecond^2)) */
    private static final double BULKMOD = 1e-4;

    /** the cell to which this membrane belongs */
    private Cell cell;

    /** the next membrane unit (counterclockwise) */
    private MembraneElement flink;

    /** the prior membrane unit (clockwise) */
    private MembraneElement blink;

    /** the actin filaments attached to this element */
    private final List<ActinFilament> actin;

    /** the activation level of the membrane element */
    private double activation;

    /** the force on the element */
    private final Vector2 force;

    /** working tuple */
    private final Vector2 e0;

    /** working tuple */
    private final Vector2 ep1;

    /** working tuple */
    private final Vector2 em1;

    /** working tuple */
    private final Vector2 em2;

    /** working tuple */
    private final Vector2 em1p1;

    /** working tuple */
    private final Vector2 term1;

    /** working tuple */
    private final Vector2 term2;

    /** working tuple */
    private final Vector2 term3;

    /**
     * Constructs a new <code>MembraneElement</code>.
     *
     * @param  theCell       the cell to which this membrane belongs
     * @param  xCoord        the X coordinate of the element
     * @param  yCoord        the Y coordinate of the element
     * @param  angle         the direction of the outward-pointing normal vector, used to construct
     *                       the actin filament attached to the membrane element
     * @param  numActinSeg   the number of actin segments to attach to the element
     * @param  actinRad      the radius of each actin segment
     */
    public MembraneElement(final Cell theCell, final double xCoord, final double yCoord,
        final double angle, final int numActinSeg, final double actinRad) {

        this(theCell, xCoord, yCoord, Math.cos(angle), Math.sin(angle), numActinSeg, actinRad);
    }

    /**
     * Constructs a new <code>MembraneElement</code>.
     *
     * @param  theCell       the cell to which this membrane belongs
     * @param  xCoord        the X coordinate of the element
```

```
     * @param  yCoord        the Y coordinate of the element
     * @param  normalX       the X component of the outward-pointing normal vector
     * @param  normalY       the Y component of the outward-pointing normal vector
     * @param  numActinSeg   the number of actin segments to attach to the element
     * @param  actinRad      the radius of each actin segment
     */
    public MembraneElement(final Cell theCell, final double xCoord, final double yCoord,
        final double normalX, final double normalY, final int numActinSeg, final double actinRad) {

        super(xCoord, yCoord, normalX, normalY);

        double len;
        double deltaX;
        double deltaY;
        double xPos;
        double yPos;
        ActinFilament prior;
        ActinFilament next;

        this.cell = theCell;
        this.flink = this;
        this.blink = this;
        this.activation = 0;
        this.force = new Vector2();

        this.actin = new ArrayList<ActinFilament>(2);
        len = Simulation.getInstance().getMaxSep() + actinRad;

        if (numActinSeg > 0) {
            xPos = xCoord - len * normalX;
            yPos = yCoord - len * normalY;
            deltaX = -2 * actinRad * normalX;
            deltaY = -2 * actinRad * normalY;

            prior = new ActinFilament(xPos, yPos, len, actinRad, null);
            this.actin.add(prior);

            for (int i = 1; i < numActinSeg; i++) {
                xPos += deltaX;
                yPos += deltaY;

                next = new ActinFilament(xPos, yPos, 2 * actinRad, actinRad, prior);
                prior = next;
            }
        }

        this.e0 = new Vector2();
        this.ep1 = new Vector2();
        this.em1 = new Vector2();
        this.em2 = new Vector2();
        this.em1p1 = new Vector2();
        this.term1 = new Vector2();
        this.term2 = new Vector2();
        this.term3 = new Vector2();
    }

    /**
     * Adds this member in the linked list before a given element.
     *
     * @param  elem  the element before which to add this element
     */
    public void addBefore(final MembraneElement elem) {

        this.setPrior(elem.getPrior());
        this.setNext(elem);
        elem.getPrior().setNext(this);
        elem.setPrior(this);
    }

    /**
     * Adds this member in the linked list after a given element.
     *
     * @param  elem  the element after which to add this element
     */
    public void addAfter(final MembraneElement elem) {

        setNext(elem.getNext());
        setPrior(elem);
        elem.getNext().setPrior(this);
        elem.setNext(this);
    }

    /**
     * Removes this element from the list in which it is installed. NOTE: the Cell object keeps a
     * reference to an element in the membrane list to allow iteration. If you remove that element,
     * that reference must be set to a valid remaining element.
     */
```

```
public void remove() {

    this.getNext().setPrior(getPrior());
    this.getPrior().setNext(getNext());
    setNext(this);
    setPrior(this);
}

/**
 * Gets the next item in the linked list.
 *
 * @return  the next item
 */
public MembraneElement getNext() {

    return this.flink;
}

/**
 * Gets the prior item in the linked list.
 *
 * @return  the prior item
 */
public MembraneElement getPrior() {

    return this.blink;
}

/**
 * Sets the next item in the linked list.
 *
 * @param  next  the next item
 */
public void setNext(final MembraneElement next) {

    this.flink = next;
}

/**
 * Sets the prior item in the linked list.
 *
 * @param  prior  the prior item
 */
public void setPrior(final MembraneElement prior) {

    this.blink = prior;
}

/**
 * Gets the force vector for the element.
 *
 * @return  the force vector
 */
public Vector2 getForce() {

    return this.force;
}

/**
 * Adds an actin filament associated with this element.
 *
 * @param  filament  the actin filament
 */
public void addActin(final ActinFilament filament) {

    this.actin.add(filament);
}

/**
 * Removes an actin filament associated with this element.
 *
 * @param  filament  the actin filament
 */
public void removeActin(final ActinFilament filament) {

    this.actin.remove(filament);
}

/**
 * Replaces an actin filament in the list with a new filament.
 *
 * @param  index     the index of the filament to replace
 * @param  filament  the new actin filament
 */
public void replaceActin(final int index, final ActinFilament filament) {

    this.actin.set(index, filament);
```

```
}

/**
 * Gets the number of actin filaments associated with this element.
 *
 * @return  the number of actin filaments
 */
public int getNumActin() {

    return this.actin.size();
}

/**
 * Gets an actin filament associated with this element.
 *
 * @param   index  the index of the actin filament to get
 * @return  the actin filament
 */
public ActinFilament getActin(final int index) {

    return this.actin.get(index);
}

/**
 * Process activation by a signal.
 *
 * @param  actLevel  the activation level carried by the signal
 */
public void activate(final double actLevel) {

    this.activation += actLevel;

    if (this.activation > MAX_ACTIVATION) {
        this.activation = MAX_ACTIVATION;
    } else if (this.activation < -MAX_ACTIVATION) {
        this.activation = -MAX_ACTIVATION;
    }
}

/**
 * Gets the current activation level of the element.
 *
 * @return  the current activation level
 */
public double getActivation() {

    return this.activation;
}

/**
 * Adjusts the current activation level;
 *
 * @param  delta  the amount by which to adjust the current activation level
 */
public void adjustActivation(final double delta) {

    this.activation += delta;
}

/**
 * Computes the force on the element.
 */
public void computeInnerForce() {

    double eqVolume;
    double volume;
    MembraneElement prior1;
    MembraneElement prior2;
    MembraneElement next1;
    MembraneElement next2;
    double normE0;
    double normEm1;
    double dot;
    double scale;
    double dist;
    double delta;

    eqVolume = this.cell.getEquilibriumVolume();
    volume = this.cell.getCurrentVolume();

    // Get the surrounding elements in the membrane
    prior1 = this.getPrior();
    prior2 = prior1.getPrior();
    next1 = this.getNext();
    next2 = next1.getNext();

    // Build relative vectors (cache these?)
```

```
        this.e0.vectorBetween(this, next1);
        this.ep1.vectorBetween(next1, next2);
        this.em1.vectorBetween(prior1, this);
        this.em2.vectorBetween(prior2, prior1);
        this.em1p1.vectorBetween(prior1, next1);
        normE0 = this.e0.length();
        normEm1 = this.em1.length();
        this.e0.normalize();
        this.ep1.normalize();
        this.em1.normalize();
        this.em2.normalize();

        // Pressure force
        scale = BULKMOD * this.cell.getThickness() * (eqVolume - volume) / (2 * eqVolume);
        this.force.addVec(scale * this.em1p1.getVecY(), -scale * this.em1p1.getVecX());

        // Tension force
        this.force.addVec(-TENSION * this.cell.getThickness()
            * ((this.em1.getVecX() * normEm1) - (this.e0.getVecX() * normE0)),
            -TENSION * this.cell.getThickness()
            * ((this.em1.getVecY() * normEm1) - (this.e0.getVecY() * normE0)));

        // Curvature force
        if ((normE0 > 0) && (normEm1 > 0)) {

            dot = this.e0.dot(this.ep1);
            this.term1.setVec(this.e0);
            this.term1.scaleVec(dot);
            this.term1.subVec(this.ep1);
            scale = 1 / (normE0 * (1 + dot) * (1 + dot));
            this.term1.scaleVec(scale);

            dot = this.em2.dot(this.em1);
            this.term2.setVec(this.em1);
            this.term2.scaleVec(-dot);
            this.term2.addVec(this.em2);
            scale = 1 / (normEm1 * (1 + dot) * (1 + dot));
            this.term2.scaleVec(scale);

            dot = this.em1.dot(this.e0);
            this.term3.setVec(this.e0);
            this.term3.scaleVec(normE0);
            this.term3.addVecScaled(-normEm1, this.em1);
            this.term3.addVecScaled(dot * normEm1, this.e0);
            this.term3.addVecScaled(-dot * normE0, this.em1);
            scale = 1 / (normEm1 * normE0 * (1 + dot) * (1 + dot));
            this.term3.scaleVec(scale);

            this.force.addVec((this.term1.getVecX() + this.term2.getVecX() + this.term3.getVecX())
                * 8 * ELASTICMOD,
                (this.term1.getVecY() + this.term2.getVecY() + this.term3.getVecY()) * 8
                * ELASTICMOD);
        }

        // Connection to actin filament, if any
        for (ActinFilament fil : this.actin) {

            dist = fil.dist(this);
            delta = dist - fil.getLength();
            this.term1.vectorBetween(this, fil);
            this.term1.normalize();
            this.term1.scaleVec(delta * Simulation.FILAMENT_SPRING);
            this.force.addVec(this.term1);
            fil.getForce().subVec(this.term1); // Don't compute again
        }
    }

    /**
     * Computes the force on the element due to interactions.
     *
     * @param  maxForce  the maximum force from non-interaction sources
     */
    public void computeInteractionForce(final double maxForce) {

        MembraneElement prior1;
        MembraneElement next1;
        double range;
        int count;
        GridMember2Int nbr;
        double dist;
        double scale;

        // Get the surrounding elements in the membrane
        prior1 = this.getPrior();
        next1 = this.getNext();

        // Interaction force with membrane or fixed elements
```

```
        count = getNumNeighbors();

        for (int i = 0; i < count; i++) {
            nbr = getNeighbor(i);

            if (nbr instanceof MembraneElement) {

                if ((nbr != next1) && (nbr != prior1)) {
                    range = Simulation.getInstance().getMaxSep();
                    dist = nbr.dist(this);

                    if (dist < range) {
                        scale = Simulation.SS_FORCE.forceTimesEqDist(dist / range) / range;

                        if (scale > maxForce) {
                            scale = maxForce;
                        }

                        this.term1.vectorBetween(nbr, this);
                        this.term1.normalize();
                        this.term1.scaleVec(scale);
                        this.force.addVec(term1);
                    }
                }
            } else if (nbr instanceof FixedElement) {
                range = 1.5 * Simulation.getInstance().getMaxSep();
                dist = nbr.dist(this);

                if (dist < range) {
                    scale = Simulation.SS_FORCE.forceTimesEqDist(dist / range) / range;

                    if (scale > maxForce) {
                        scale = maxForce;
                    }

                    this.term1.vectorBetween(nbr, this);
                    this.term1.normalize();
                    this.term1.scaleVec(scale);
                    this.force.addVec(term1);
                }
            } else if (nbr instanceof ActinFilament) {
                dist = nbr.dist(this);
                range = Simulation.getInstance().getMaxSep() + nbr.getRadius();

                if (dist < range) {
                    scale = Simulation.SS_FORCE.forceTimesEqDist(dist / range) / range;

                    if (scale > maxForce) {
                        scale = maxForce;
                    }

                    this.term1.vectorBetween(nbr, this);
                    this.term1.normalize();
                    this.term1.scaleVec(scale);
                    this.force.addVec(term1);
                }
            }
        }
    }

    /**
     * Computes the normal vector at the element based on its neighbors.
     */
    public void recomputeNormal() {

        MembraneElement prior1;
        MembraneElement next1;

        prior1 = this.getPrior();
        next1 = this.getNext();

        this.em1p1.vectorBetween(prior1, next1);
        this.em1p1.normalize();
        this.setVec(this.em1p1.getVecY(), -this.em1p1.getVecX());
    }
}

package com.srbenoit.modeling.cell;

import com.srbenoit.modeling.grid.PointGridMember2;

/**
 * A signal emitted by an emitter;
 */
public class Signal extends PointGridMember2 {

    /** the remaining life of this signal */
```

```
    private int life;

    /** the activation level this signal carries */
    private final double activation;

    /**
     * Constructs a new <code>Signal</code>.
     *
     * @param  xCoord     the X coordinate of the signal location
     * @param  yCoord     the Y coordinate of the signal location
     * @param  lifetime   the lifetime of the signal
     * @param  actLevel   the activation level this signal carries
     */
    public Signal(final double xCoord, final double yCoord, final int lifetime,
        final double actLevel) {

        super(xCoord, yCoord);

        this.life = lifetime;
        this.activation = actLevel;
    }

    /**
     * Ages the signal, decrementing its remaining life.
     *
     * @return  the remaining life after aging
     */
    public int age() {

        this.life--;

        return this.life;
    }

    /**
     * Kills the signal (sets its life to zero). Called when a signal is absorbed. The signal is
     * not removed from the grid.
     */
    public void die() {

        this.life = 0;
    }

    /**
     * Test whether a signal is still alive.
     *
     * @return  <code>true</code> if the signal is still alive
     */
    public boolean isLiving() {

        return this.life > 0;
    }

    /**
     * Gets the activation level this signal carries.
     *
     * @return  the activation level
     */
    public double getActivation() {

        return this.activation;
    }
}

package com.srbenoit.modeling.cell;

import java.awt.geom.Rectangle2D;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import com.srbenoit.geom.Point2;
import com.srbenoit.geom.Vector2;
import com.srbenoit.log.LoggedObject;
import com.srbenoit.modeling.grid.FastLennardJones;
import com.srbenoit.modeling.grid.FastSoftSphere;
import com.srbenoit.modeling.grid.Grid2D;
import com.srbenoit.modeling.grid.GridMember2Int;
import com.srbenoit.ui.UIUtilities;

/**
 * A cell simulation.
 */
public class Simulation extends LoggedObject implements Runnable {

    /** the singleton instance */
```

```
    private static final Simulation INSTANCE;

    /** percentage of border to add around cells when rendering */
    private static final double BORDER = 0.05;

    /** the cell radius, in microns */
    private static final double CELL_RADIUS = 5;

    /** the number of membrane elements per cell */
    private static final int MEMBRANE_PER_CELL = 200;

    /** the number of actin layers in each cell */
    private static final int ACTIN_LAYERS = 4;

    /** the number of frames to process between renderings */
    private static final double FRAMES_PER_RENDER = 100;

    /** the number of frames to process between renderings (negative to not output anything) */
    private static final double FRAMES_PER_OUTPUT = 10000;

    /** a soft-sphere force calculator for use within the simulation */
    public static final FastSoftSphere SS_FORCE;

    /** a soft-sphere force calculator for use within the simulation */
    public static final FastSoftSphere ACTIN_FORCE1;

    /** a soft-sphere force calculator for use within the simulation */
    public static final FastLennardJones ACTIN_FORCE2;

    /** a Lennard-Jones force calculator for use within the simulation */
    public static final FastLennardJones LJ_FORCE;

    /** hand of God force to add to emitters */
    public static final Vector2 HAND_OF_GOD;

    /** flag to control whether signals are emitted */
    public static final boolean EMIT_SIGNALS = true;

    /** Hooke's law spring constant for filaments */
    public static final double FILAMENT_SPRING = 1e-8;

    /** the panel that will render the cells */
    private CellPanel panel;

    /** the list of cells being simulated */
    private final List<Cell> cells;

    /** the list of signal emitters */
    private final List<Emitter> emitters;

    /** the list of fixed walls */
    private final List<Wall> walls;

    /** the bounds of the simulation region */
    private final Rectangle2D bounds;

    /** the minimum permitted separation between model elements */
    private final double avgSep;

    /** the minimum permitted separation between model elements */
    private final double minSep;

    /** the maximum permitted separation between model elements */
    private final double maxSep;

    /** the maximum permitted separation between model elements */
    private final double maxMove;

    /** the grid to do neighbor testing */
    private Grid2D grid;

    static {
        INSTANCE = new Simulation();
        INSTANCE.buildScene();

        SS_FORCE = new FastSoftSphere(1e-1);
        ACTIN_FORCE1 = new FastSoftSphere(1e-5);
        ACTIN_FORCE2 = new FastLennardJones(1e-12);
        LJ_FORCE = new FastLennardJones(1e-11);

// HAND_OF_GOD = new Vector2(-1e-11, 0); // Left-moving pusher
        HAND_OF_GOD = new Vector2(0, 6e-12); // upward-moving "chase me"
    }

    /**
     * Gets the singleton simulation instance.
     *
```

```
     * @return  the instance
     */
    public static Simulation getInstance() {

        return INSTANCE;
    }

    /**
     * Constructs a new <code>Simulation</code>.
     */
    private Simulation() {

        super();

        this.avgSep = 2 * Math.PI * CELL_RADIUS / MEMBRANE_PER_CELL;
        this.maxSep = 4 * this.avgSep / 3;
        this.minSep = this.maxSep / 2;
        this.maxMove = this.minSep / 50;

        this.cells = new ArrayList<Cell>(10);
        this.emitters = new ArrayList<Emitter>(10);
        this.walls = new ArrayList<Wall>(10);
        this.bounds = new Rectangle2D.Double();
    }

    /**
     * Builds the scene.
     */
    private void buildScene() {

        double neighborhood;
        int width;
        int height;

        // We set the neighborhood to twice the maximum separation size, for the grid
        neighborhood = 2 * this.maxSep;

        this.cells.add(new Cell(new Point2(-2, 0), CELL_RADIUS, MEMBRANE_PER_CELL, ACTIN_LAYERS,
                ACTIN_LAYERS * this.avgSep, this.avgSep, this.maxMove));
        this.emitters.add(new Emitter(7, 0, this.maxMove * 0.1,
                EMIT_SIGNALS ? (10 * MEMBRANE_PER_CELL) : 0, this.minSep, 1,
                (int) ((2 * Math.PI / this.minSep) + 0.5)));
        this.walls.add(new Wall(2 * CELL_RADIUS, -1.5 * CELL_RADIUS, -2 * CELL_RADIUS,
                -1.5 * CELL_RADIUS, this.avgSep, true));
        this.walls.add(new Wall(-2 * CELL_RADIUS, -1.5 * CELL_RADIUS, -2 * CELL_RADIUS,
                3 * CELL_RADIUS, this.avgSep, true));
        this.walls.add(new Wall(-2 * CELL_RADIUS, 3 * CELL_RADIUS, 2 * CELL_RADIUS,
                3 * CELL_RADIUS, this.avgSep, true));
        this.walls.add(new Wall(2 * CELL_RADIUS, -1.5 * CELL_RADIUS, 2 * CELL_RADIUS,
                3 * CELL_RADIUS, this.avgSep, true));

        computeBounds(this.bounds, this.cells, this.emitters);

        width = (int) (this.bounds.getWidth() / neighborhood) + 1;
        height = (int) (this.bounds.getHeight() / neighborhood) + 1;
        this.grid = new Grid2D(this.bounds.getX(), this.bounds.getY(), neighborhood, width,
                height);

        for (Cell cell : this.cells) {
            cell.addToGrid(this.grid);
        }

        for (Emitter emitter : this.emitters) {
            emitter.addToGrid(this.grid);
        }

        for (Wall wall : this.walls) {
            wall.addToGrid(this.grid);
        }
    }

    /**
     * Get the maximum separation between model elements.
     *
     * @return  the maximum separation
     */
    public double getMaxSep() {

        return this.maxSep;
    }

    /**
     * Computes the bounding rectangle that contains all cell elements and emitters, with a border.
     * We assume actin is contained in the cell, and if the rectangle contains the cell, it
     * automatically contains the actin.
     *
     * @param  bounds     the rectangle to populate with bounds
```

```
     * @param   cells      the list of cells
     * @param   emitters   the list of emitters
     */
    private void computeBounds(final Rectangle2D bounds, final List<Cell> cells,
        final List<Emitter> emitters) {

        MembraneElement mem;

        if (emitters.size() > 0) {
            bounds.setFrame(emitters.get(0).getPosX(), emitters.get(0).getPosY(), 0, 0);
        } else if (cells.size() > 0) {
            bounds.setFrame(cells.get(0).getMembrane().getPosX(),
                cells.get(0).getMembrane().getPosY(), 0, 0);
        }

        for (Cell cell : cells) {
            mem = cell.getMembrane();

            for (;;) {
                bounds.add(mem.getPosX(), mem.getPosY());
                mem = mem.getNext();

                if (mem == cell.getMembrane()) {
                    break;
                }
            }
        }

        for (Emitter emitter : emitters) {
            bounds.add(emitter.getPosX(), emitter.getPosY());

            for (int i = 0; i < emitter.getNumFixed(); i++) {
                bounds.add(emitter.getFixed(i).getPosX(), emitter.getFixed(i).getPosY());
            }
        }

        for (Wall wall : walls) {

            for (int i = 0; i < wall.getNumFixed(); i++) {
                bounds.add(wall.getFixed(i).getPosX(), wall.getFixed(i).getPosY());
            }
        }

        bounds.setFrame(bounds.getX() - (bounds.getWidth() * BORDER),
            bounds.getY() - (bounds.getHeight() * BORDER),
            bounds.getWidth() + (bounds.getWidth() * 2 * BORDER),
            bounds.getHeight() + (bounds.getHeight() * 2 * BORDER));
    }

    /**
     * Constructs the user interface in the AWT thread.
     */
    public void run() {

        JFrame frame;

        frame = new JFrame("Cell Simulation");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.panel = new CellPanel(500, 550);
        frame.setContentPane(this.panel);
        frame.pack();
        UIUtilities.positionFrame(frame, 0.5, 0.5);
        frame.setVisible(true);
    }

    /**
     * Runs the simulation after the user interface is created.
     */
    public void go() {

        MembraneElement elem;
        int count;
        ActinFilament actin;
        int frameNum;
        double max;
        int outputFrame;

        // Set an external force on emitters if we want them to move
        for (Emitter emitter : this.emitters) {
            emitter.getHandOfGod().setVec(HAND_OF_GOD);
        }

        outputFrame = 0;

        for (frameNum = 0; frameNum < Integer.MAX_VALUE; frameNum++) {
```

```
            // Do things that change the positions of objects in the grid...
            evolveSignals();
            restructureActin();
            restructureMembrane();

            // Recompute neighbor relationships since nodes may have been added/removed
            for (Cell cell : this.cells) {
                elem = cell.getMembrane();

                do {
                    this.grid.getNeighborsOf(elem);

                    count = elem.getNumActin();

                    for (int i = 0; i < count; i++) {
                        actin = elem.getActin(i);

                        while (actin != null) {
                            this.grid.getNeighborsOf(actin);
                            actin = actin.getDownstream();
                        }
                    }

                    elem = elem.getNext();
                } while (elem != cell.getMembrane());
            }

            // Do force-based forces and motions
            computeInnerForces();
            max = maxForce();
// LOG.fine("Max: " + max);
            computeInteractionForces(max);
            reactToForces(max);

            // Recompute membrane element normal vectors
            recomputeNormals();

            // Act on adjacencies in the grid, but don't move anything
            detectSignalsAtMembrane();

            if ((frameNum % FRAMES_PER_RENDER) == 0) {
                this.panel.render(this.bounds, this.cells, this.emitters, this.walls);

                if ((FRAMES_PER_OUTPUT > 0) && ((frameNum % FRAMES_PER_OUTPUT) == 0)) {
                    outputFrame++;
                    this.panel.exportFrame(outputFrame);
                }
            }
        }
    }

    /**
     * Emits new signals, ages and deletes expired signals, and allows them to diffuse.
     */
    private void evolveSignals() {

        for (Emitter emitter : this.emitters) {
            emitter.emit(5);
            emitter.age();
            emitter.diffuse(this.bounds);
        }
    }

    /**
     * Scan for signals in contact with membrane, and activate the corresponding membrane,
     * consuming the signal in the process.
     *
     * @param  iter  an iterator that can be used to walk the grid
     */
    private void detectSignalsAtMembrane() {

        MembraneElement elem;
        GridMember2Int nbr;
        Signal sig;
        int count;
        double distSq;
        double rangeSq;

        rangeSq = this.maxSep * this.maxSep;

        for (Cell cell : this.cells) {
            elem = cell.getMembrane();

            do {

                count = elem.getNumNeighbors();
```

```
                for (int i = 0; i < count; i++) {
                    nbr = elem.getNeighbor(i);

                    if (nbr instanceof Signal) {
                        sig = (Signal) nbr;

                        if (sig.isLiving()) {
                            distSq = sig.distSquared(elem);

                            if (distSq < rangeSq) {
                                elem.activate(sig.getActivation());
                                sig.die();
                            }
                        }
                    }
                }

                elem = elem.getNext();
            } while (elem != cell.getMembrane());
        }
    }

    /**
     * React to signal levels in the membrane to cause changes in actin polymerization.
     */
    private void restructureActin() {

        for (Cell cell : this.cells) {
            cell.restructureActin();
        }
    }

    /**
     * Adjust the membrane as needed to ensure adjacent elements are neither too close together nor
     * too far apart.
     */
    private void restructureMembrane() {

        for (Cell cell : this.cells) {
            cell.restructureMembrane(this.minSep, this.maxSep);
        }
    }

    /**
     * Compute the force on every element in the model due to internal factors.
     */
    private void computeInnerForces() {

        for (Cell cell : this.cells) {
            cell.computeInnerForces();
        }

        for (Emitter emitter : this.emitters) {
            emitter.computeInnerForces();
        }
    }

    /**
     * Computes the maximum force anywhere in the system.
     *
     * @return  the maximum force
     */
    private double maxForce() {

        double max;
        double force;

        max = 0;

        for (Cell cell : this.cells) {
            force = cell.maxForce();

            if (force > max) {
                max = force;
            }
        }

        for (Emitter emitter : this.emitters) {
            force = emitter.getForce().length();

            if (force > max) {
                max = force;
            }
        }

        return max;
    }
```

```
    /**
     * Compute the force on every element in the model due to interactions.
     *
     * @param  maxForce  the maximum force from non-interaction sources
     */
    private void computeInteractionForces(final double maxForce) {

        for (Cell cell : this.cells) {
            cell.computeInteractionForces(maxForce);
        }

        for (Emitter emitter : this.emitters) {
            emitter.computeInteractionForces(maxForce);
        }
    }

    /**
     * Move elements in the direction of their forces.
     *
     * @param  maxForce  the maximum force in the system
     */
    private void reactToForces(final double maxForce) {

        double mobility;

        if (maxForce > Double.MIN_NORMAL) {
            mobility = this.maxMove / maxForce;

            for (Cell cell : this.cells) {
                cell.reactToForces(mobility);
            }

            for (Emitter emitter : this.emitters) {
                emitter.reactToForces(mobility);
            }
        }
    }

    /**
     * Computes the normal vector at each membrane element based on its neighbors.
     */
    private void recomputeNormals() {

        for (Cell cell : this.cells) {
            cell.recomputeNormals();
        }
    }

    /**
     * Main method to run the simulation.
     *
     * @param  args  command-line arguments
     */
    public static void main(final String... args) {

        Simulation sim;

        sim = Simulation.getInstance();

        try {
            SwingUtilities.invokeAndWait(sim);
            sim.go();
        } catch (Exception ex) {
            LOG.log(Level.SEVERE, "Exception in simulation", ex);
        }
    }
}

package com.srbenoit.modeling.cell;

import com.srbenoit.modeling.grid.Grid2D;

/**
 * A wall of fixed elements
 */
public class Wall {

    /** the fixed elements */
    private FixedElement[] fixed;

    /**
     * Constructs a new <code>Wall</code>.
     *
     * @param  startX        the starting X coordinate of the emitter
     * @param  startY        the starting Y coordinate of the emitter
     * @param  endX          the ending X coordinate of the emitter
```

```
     * @param  endY          the ending Y coordinate of the emitter
     * @param  maxSep        the maximum separation between elements
     * @param  includeLast   true to include the last element, false not to include it
     */
    public Wall(final double startX, final double startY, final double endX, final double endY,
        final double maxSep, final boolean includeLast) {

        double distX;
        double distY;
        double dist;
        int numSegs;
        double deltaX;
        double deltaY;
        double xPos;
        double yPos;
        int numElements;
        int total;

        distX = endX - startX;
        distY = endY - startY;
        dist = Math.sqrt((distX * distX) + (distY * distY));
        numSegs = (int) ((dist / maxSep) + 0.999);
        deltaX = distX / numSegs;
        deltaY = distY / numSegs;

        numElements = includeLast ? (numSegs + 1) : numSegs;

        this.fixed = new FixedElement[numElements];

        for (int i = 0; i < numElements; i++) {
            xPos = startX + (i * deltaX);
            yPos = startY + (i * deltaY);

            this.fixed[i] = new FixedElement(xPos, yPos);
        }
    }

    /**
     * Gets the number of fixed elements.
     *
     * @return  the number of fixed elements
     */
    public int getNumFixed() {

        return this.fixed.length;
    }

    /**
     * Gets a particular fixed element.
     *
     * @param   index  the index of the fixed element
     * @return  the fixed element
     */
    public FixedElement getFixed(final int index) {

        return this.fixed[index];
    }

    /**
     * Adds the emitter and any active signals to a grid, and stores the grid so emitted signals
     * can be automatically added to the grid and expiring signals can be removed.
     *
     * @param  grid  the grid
     */
    public void addToGrid(final Grid2D grid) {

        for (FixedElement elem : this.fixed) {
            elem.installInGrid(grid);
        }
    }
}
```

REFERENCES

[1] M. Botsch and L. P. Kobbelt. A robust procedure to eliminate degenerate faces from triangle meshes. *Proceedings of the Sixth International Fall Workshop Vision, Modeling, and Visualization*, pages 402–410, 2001.

[2] D. D. Holm. *Geometric Mechanics II: Rotation, Translation and Rolling*. Imperial College Press, 2009.

[3] I. Mezic. On the dynamics of molecular conformation. *Proc. Natl. Acad. Sci.*, 103:7542–7547, 2006.