Dissertation

Homotopy Continuation Methods, Intrinsic Localized Modes, and Cooperative Robotic Workspaces

Submitted by

Daniel Abram Brake

Department of Mathematics

In partial fulfullment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2012

Abstract

Homotopy Continuation Methods, Intrinsic Localized Modes, and Cooperative Robotic Workspaces

This dissertation considers three topics that are united by the theme of application of geometric and nonlinear mechanics to practical problems.

Firstly we consider the parallel implementation of numerical solution of nonlinear polynomial systems depending on parameters. The program written to do this is called *Paramotopy*, and uses the Message Passing Interface to distribute homotopy continuation solves in another program called *Bertini* across a supercomputer. Paramotopy manages writing of Bertini input files, allows automatic re-solution of the system at points at which paths failed, and makes data management easy. Furthermore, parameter homotopy nets huge performance gains over fresh homotopy continuation runs. Superlinear speedup was achieved, up to hard drive throughput capacity. Various internal settings are demonstrated and explored, and the User's Manual is included.

Second, we apply nonlinear theory and simulation to nanomechanical sensor arrays. Using vibrating GaAs pillars, we model Intrinsic Localized Modes (ILMs), and investigate ILM-defect pinning, formation, lifetime, travel and movement, and parameter dependence. Intrinsic Localized Modes have been analyzed on arrays of nonlinear oscillators. So far, these oscillators have had a single direction of vibration. In current experiments for single molecule detection, arrays made of Gallium Arsenide will be innately bidirectional, forced, dissipative. We expand previous full models to bidirectionality, and simulate using ODE solvers. We show that small regions of a very large parameter space permit strong ILM formation. Additionally, we use Hamiltonian mechanics to derive new simplified models for the monodirectional ILM travel on an infinite array. This monodirectional ILMs of constant amplitude have unrealistic behavior. Permitting the amplitude of the ILM to vary in time produces much more realistic behavior, including wandering and intermittent pinning.

The final set of problems concerns the application of numerical algebraic geometric methods to untangle the phase space of cooperating robots, and optimize configuration for fault tolerance. Given two robots in proximity to each other, if one experiences joint failure, the other may be able to assist, restoring lost workspace. We define a new multiplicity-weighted workspace measure, and use it to solve the optimization problem of finding the best location for an assistance socket and separation distance for the two robots, showing that the solution depends on robot geometry, which link is being grasped, and the choice of objective function.

Table of Contents

List of Figures

Chapter 1

Paramotopy: Parallel Parameter Homotopy through Bertini

Algebraic Geometry is a mathematical field of study which has blossomed vigorously over the last few decades. It has grown from the study of geometry arising from coupling algebraic equations, to include and relate to such fields as Gromov-Witten invariants, string theory, intersection theory, and many others.

Recently, the application of algebraic geometric methods to concrete, real-world problems has received much attention. Because algebraic constraints arise in many contexts, it is only natural to try to exploit this nature. Fields such as robotic workspace analysis [1, 2], robot design [3], kinematics [4], and optimal control [5], all use some form of geometric methods to solve problems.

Algebraic Geometry could be said to be the study of systems of algebraic equations; *e.g.*, the study of the zero-sets of coupled sets of polynomials, of invariants of such zero-sets, of their topology. A brief treatment of foundational material is merited.

Let $\mathbb{F}$ be our favorite field, usually $\mathbb{C}$, and adjoin to it n variables, creating the ring of polynomials $\mathbb{F}[x_1, x_2, \ldots, x_n]$. Then we can form an ideal I by taking generating polynomials, and writing $I = \langle f_1(x_1, x_2, \ldots, x_n), \ldots, f_k(x_1, x_2, \ldots, x_n)\rangle$. That is, I is the set of all elements of the form $\cup_{i=1}^{k} p_i(x_1, \ldots, x_n) f_i(x_1, \ldots, x_n)$, where the p_i are arbitrary polynomials coming from the polynomial ring. From here on we frequently make implicit the dependence of polynomials f_i on any variables.

Definition 1. *The set of points on which all polynomials f_i in I are simultaneously zero is the* ***variety*** *associated with I, and is written $V(I)$.*

Varieties may have one or more *components*, which may or may not be all of the same dimension; the dimension of a component of $V(I)$ is defined in the usual sense. There may be multiple components of the same dimension as well. The dimension of a component is defined to be the dimension of the tangent space at a nonsingular point of the component. For example, we call *points* zero-dimensional, *curves* one-dimensional, *surfaces* two-dimensional, etc.

One of the fundamental problems in algebraic geometry is, given some set of polynomials corresponding to I, what is a generating set for the ideal? That is, it is not obvious from looking at a set of polynomials whether another polynomial could be in the ideal. We want a membership test, or a way to compute the variety. The desire to quickly, accurately, and automatically answer the ideal membership question gives rise to the field of Computational Algebraic Geometry.

Various tools have emerged to answer the ideal membership test, among other questions. The two main categories of methods are *symbolic* and *numeric*. The former computes directly with the generators the user provides, arriving at exact equations. For example, if a user desires a unique generating set for an ideal, they could call a Gröbner basis method, perhaps as implemented in Maple, GAP, Singular, Macaulay2, or others. The inputs are symbolic expressions, and the output is also a set of symbolic expressions. From the computed basis, we can determine whether a given polynomial lies in the ideal. A more thorough discussion of symbolic methods, namely Gröbner Bases and Buchberger's Algorithm, may be found in Appendix B.

In contrast to the exact world of symbolic computation, numeric methods work with floating point approximations of numbers. There is a moderate number of well-developed software packages which find varieties associated to ideals. The one we will focus on here will be the Homotopy Continuation solver, *Bertini* [6], which can find both zero and positive dimensional varieties. In addition, our focus will be not directly on the ideal membership test, but rather on finding the zero-dimensional components of V.

1. Homotopy Continuation

In contrast to symbolic methods, which can give exact equations for varieties, numerical methods give approximations to the solutions of systems of equations. The only numeric method we will discuss here is Homotopy Continuation [7].

A *homotopy*, shortly, is a continuous deformation. To *continue* perhaps is to follow. The combination of these two ideas gives *homotopy continuation*, which is the following of solutions as one system of equations is continuously deformed into another.

The simplest example of a homotopy is perhaps the linear case. Take two functions f, g, along with a 'time' variable $t \in \mathbb{C}$, and write

$$h(t) = (1-t) \cdot f + t \cdot g \,. \tag{1}$$

If we start at $t = 1$ and continuously vary to $t = 0$, then h will be a continuous deformation of g into f. Note that there is nothing in Equation 1 which prohibits the use of $t \in \mathbb{C}$; as long as our functions can be defined over $\mathbb{C}$, and we ourselves are willing to work over $\mathbb{C}$, complex time is just fine.

Numerically, we track solutions to (1) as t deforms. Hence, to perform a homotopy continuation solve, we must obtain the solutions to the start system g, and the number of solutions to g ought to be greater than the number of solutions to f. The generic method for homotopy continuation is presented in Algorithm 8, and is presented in Figure 1. Obtaining the start solutions tends to be easy – we construct a system with known solutions, and simply write them down.

There is a well known upper bound on the number of solutions to a system, called the Bézout bound [8, 9]. The number of isolated solutions of a polynomial system is no more than the product of the degrees of the polynomials.

Consider the roots of unity example. If we have a system for which the Bézout bound is N, and we wish to use the "total degree" start system, factor N as $N = \prod_i d_i$. Then we can write,

$$g(z) = \begin{cases} g_1(z_1) &= z_1^{d_1} - 1 \\ \quad\vdots & \\ g_n(z_n) &= z_n^{d_n} - 1 \end{cases} \tag{2}$$

The function g has as its zeros the roots of unity specified by the decomposition, and we require no computation to find them. In contrast, it is the finding of solutions to f that is hard. Generally, f will be nonlinear, and may have many monomials. Also, if g has more solutions than the target system f, the excess solutions will diverge to ∞ as $t \to 0$. Any 'excess paths' waste CPU time – thus, reduction of the upper bound on the number of solutions to f is greatly desired.

Sometimes a system to be solved using numerical methods will have parameters $\bar{p}$ in it. Perhaps they represent a point in space, an orientation, or rates of chemical reactions. Regardless of the meaning of the parameters, homotopy continuation lends itself to rapid computation of a system for different values of parameters with what is called a *parameter homotopy*. This is a two step process. First, we perform a primary solve of the system, using a constructed start system which will, more than likely, overestimate the number of paths necessary, using randomly chosen complex values $\bar{p}^*$ of the parameters. The second step uses the result of the first solve as its start system, tracking now only the *exact* number of solutions from $\bar{p}^*$ to $\bar{p}_i$, the particular point at which we wish to solve. For a graphical representation of this, see Figure 2. This was originally called the 'Cheater's Homotopy' [10], though it is referred to differently now.

The benefit of a parameter homotopy is reduction of computation time. Suppose we wish to solve the system for a set $\{\bar{p}_i\}$ of parameter values. Performing a fresh solve of

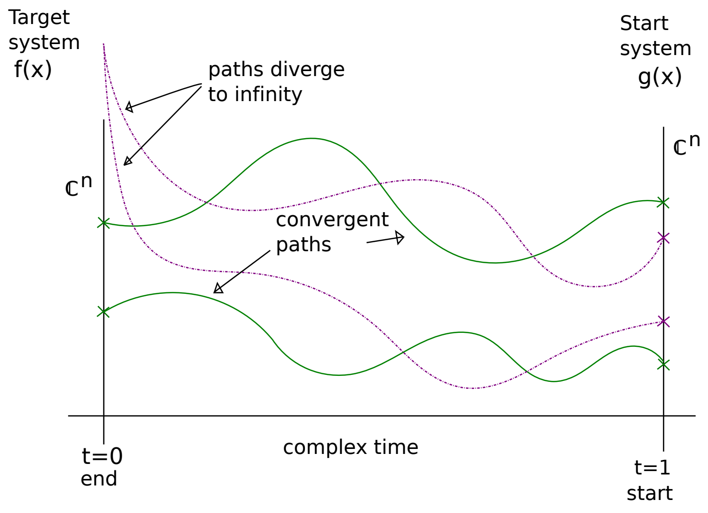


Figure 1. An illustration of generic homotopy continuation. The magenta paths from the constructed start system diverge to ∞, while green paths converge to solutions to the desired system f. Theory guarantees that off a set of measure zero in $\mathbb{C}^n$, we will have a uniform number of (complex) solutions to f. The divergent paths are present because the constructed start system has a surplus of solutions.

the system for each distinct parameter value would repeatedly follow paths diverging to ∞. We would prefer to track only *exactly* the correct number of paths each time. Since theory guarantees that if $\bar{p}^*$ is a randomly chosen point (perhaps using a random number generator, and off a set of measure zero) in parameter space, it will have the exact same number of solutions as any other generic point. Hence, we repeatedly track from only the solutions at $\bar{p}^*$, instead of starting fresh from a constructed system, if we wish to solve at particular points in parameter space. This reduces dramatically the number of paths to track when solving a system repeatedly.

Specific points in the parameter space of a system my have a different number of solutions than a generic point. This number is always a decrease from the generic number. This guarantees that no solutions will ever be missed when tracking from the generic point to the specific.

Algorithm 1 Generic Homotopy Continuation

```
1: procedure HOMOTOPY CONTINUATION($f$)
2:     Find upper bound $b$ on number of solutions to $f$
3:     Form start system $g$ having (# solutions) = $b$
4:     Write $h = (1 - t)f + t \cdot g$
5:     for each solution to $g$ do
6:         Track solution as $t$ is deformed through $\mathbb{C}$ from 1 to 0
7:     end for
8: end procedure
```

1.1. Bertini. We turn now to a description of the homotopy continuation solver of which Paramotopy will make use. Bertini is a freely available software package, using predictor-corrector methods to track from a start system, and end at a target system. Paramotopy is designed to repeatedly call Bertini to perform parameter homotopies in parallel, as well as provide data management and pre- and post-processing capabilities.

Beginning from the pre-obtained solutions of the start system, Bertini uses successive predictor and corrector solves, combined with 'endgame' methods at the tail end of the solve, to arrive at the solutions to a desired system. The predictor steps are computed using the Jacobian to the system. Since solution of a matrix system induces magnification of error, the condition number K of the matrix is monitored, and the precision of the computation can be dynamically adjusted using Adaptive Multiple Precision methods. At present, there are at least eight options available to the user for choice of predictor method. Following a predictor step, which advances in time ($t \in \mathbb{C}$), a Newton step is called to move the approximation to the path closer to the actual path. If the Newton step takes too long to converge, Bertini returns to the previous good step, and retries with a smaller step size in t. The predictor-corrector pattern continues until the solve is almost complete, and t is almost 0. Finally, Bertini turns to endgame methods [7] to complete the solve, ensuring that paths remain separate.

If two paths pass close to a singularity, they may come close to each other, numerical path-crossing may occur. This means that the predictor-corrector might accidentally switch from one path to another. Bertini monitors for this, and checks at the end of a solve to see whether any paths have crossed. If paths cross, Bertini informs the user.

Paths may also fail in a way more critically than jumping, as Bertini tracks solutions, such that Bertini gives up on tracking the path. As a genuinely superfluous path approaches $t = 0$, it will necessarily diverge to infinity. However, sometimes paths which would lead to actual solutions to a system may *appear* to diverge, as its norm becomes large. Bertini checks for this at the end of a solve, and writes to file `failed_paths` which paths failed.

Bertini, while it tracks only points through a solve, is capable of determining the dimensions, degrees, and numbers of components of a variety. Hence, solves are not limited to square systems, where the number of equations is *equal* to the number of variables. In general, the dimension of a variety is equal to the difference in the number of variables n, and the number of equations m, as $n-m$. An underconstrained system where $n > m$ will tend to have positive dimensional components, and possibly some of lesser dimensions. Bertini can be instructed to sample these; the samples will almost certainly lie in $\mathbb{C} - \mathbb{R}$, of appropriate dimension, and it is currently very difficult to perform *real* sampling of positive dimensional components. Additionally, this is merely an intuitive description, and there can in fact be multiple solution components in various dimensions.

Bertini has many more modes and capabilities, which are enumerated and described in the Bertini User's Manual, as well as an upcoming book.

2. Paramotopy

The Paramotopy software is an extension of Bertini. Because Bertini has built-in support for custom and parameter homotopies, it can be used to solve a system for myriad parameter values. However, the `userhomotopy` mode in Bertini has only support for one parameter point at a time, so if a user wishes to solve a system for 10 million parameter values (not that many, really), they must run Bertini 10 million separate times. Paramotopy takes care of running the Bertini solves for an entire parameter set automatically, in parallel, and in such a fashion as is portable and easy to use.

2.1. Program Algorithms.

A very coarse outline of Paramotopy is:

(1) Perform 'Step1' solve; run Bertini for random complex values of the parameters appearing in the system to be solved,
(2) For each point in the parameter space sample, perform the 'Step2' solve.

Step1 is sometimes referred to as the offline step, and Step2 as the online. A more thorough description of Paramotopy is in Algorithm 2. Essentially, the user-guided program initializes, runs Step1 and Step2, and allows for re-solution at parameter points at which paths failed. The program is provided as two separate compiled executables named (unsurprisingly) `Paramotopy` and `Step2`. `Paramotopy` as an executable is a serial program, which manages the parsing of input files, management of Bertini settings, and calling of `Bertini` and `Step2`. A simplified model of the process is given in Figure 2.

The majority of the CPU time in Paramotopy is spent in `Step2`. Using the common Message Passing Interface (MPI) method for parallelism in software, `Step2` is implemented with the master-slave control scheme. It is described in Algorithm 3 and 4. The master node possesses a buffer containing the parameter points at which we will solve; each slave is responsible for writing its own input files for Bertini, and writing its own data files.

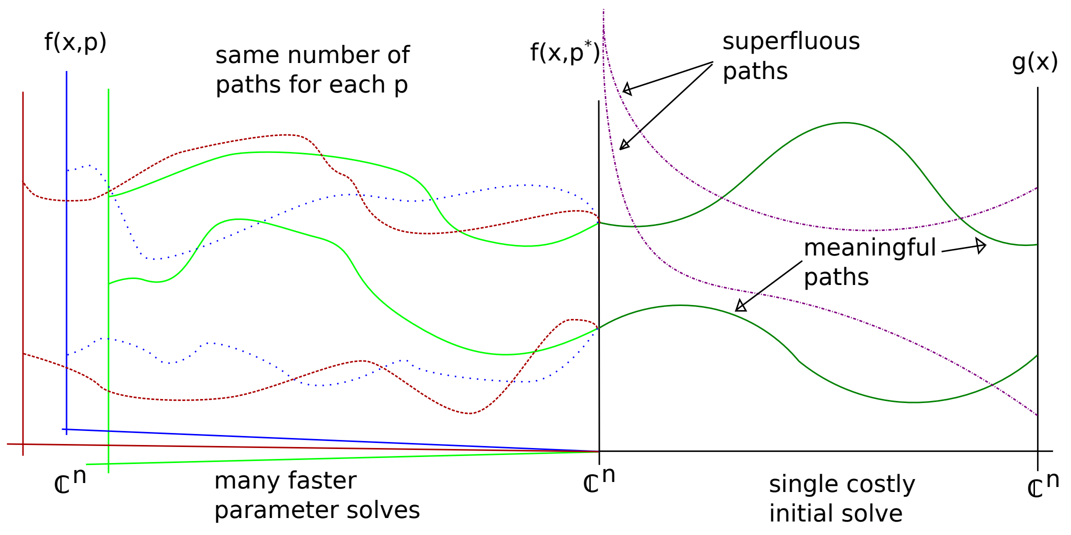

f(x,p)
same number of paths for each p
f(x,p*)
superfluous paths
g(x)
meaningful paths
ℂn
many faster parameter solves
ℂn
single costly initial solve
ℂn

Figure 2. An illustration of the Paramotopy method. Starting at the right from a constructed system in Bertini, we continue to the left, to solutions of the desired system for random complex values. From there, we continue once more (many times more) to any desired set of parameter values.

The master process (Algorithm 3) remains largely idle, especially for smaller runs; its main purpose is to achieve equal distribution of points to solve. Initially, it sends each worker a set of m points, passed in the form of an array of type double. Following the initial seeding of workers, the master enters a wild-card receive, waiting for workers to finish. When a worker tells the master it has completed the solves for each of its data points, the master sends it new work, and enters receive mode once more. This continues until the list of parameter points is exhausted.

The slave function is described in Algorithm 4. Regarding the Bertini solve, the slave has a two-step process. To improve performance, and due to issues related to the Bertini library calls (namely, to forego parsing a nearly identical input file for each and every solve), each worker performs an initial parameter solve on arbitrary nonzero complex parameter values, so as to initialize certain memory structures. Once completed, the worker is ready to solve en masse. It enters a while loop, and receives double arrays containing parameter points at which to solve from the headnode. After receiving, the worker calls Bertini, gathers

Algorithm 2 Paramotopy Algorithm. A C++ compiled wrapper around Bertini.

```
 1: procedure Paramotopy(Input File)
 2:     Load input file
 3:     Load previous preferences, or create new if not found
 4:     Get correct folder for run
 5:     User modifies settings for run
 6:     Write config and input sections of Bertini input file          ▷ Step1
 7:     system() call to Bertini, parallel or not                       ▷ Step1
 8:     User Ensures quality of Step1 results
 9:     if parallel then                                                ▷ Begin Step2
10:         Paramotopy calls system('mpilauncher step2')
11:         Process command string – files to save, and other parameters
12:         Load preferences
13:         if MPIid==0 then
14:             master()                                                ▷ Algorithm 3
15:         else
16:             slave()                                                 ▷ Algorithm 4
17:         end if
18:         Return to Paramotopy
19:     else
20:         Run Step2 internally, single process performing both master and slave
21:     end if                                                          ▷ End Step2
22:     FailedPathAnalysis()
23: end procedure
```

the data from the output files, and stores the data into a file unique to the worker. The receive-solve-write routine is iterated until the work is complete. The exit condition of this while loop is a special integer message tag sent from the master.

Finally, after a Step2 run has been completed, the user may elect to engage in Failed Path Analysis. Paramotopy scans the output files for parameter points at which one or more paths failed during the Bertini run, and reports the number of such points. User may modify settings for Bertini, such as choosing higher accuracy throughout the solve, and elevating the security level. The user may also set a number of automatic resolves, and automatic tolerance tightening.

Currently, analysis of generated data must be done through a separate program. A general set of data-gathering and plotting functions has been written in Matlab by the author. However, specialized or custom analysis must be done by the user.

Algorithm 3 Paramotopy master function.

```
1: function MASTER
2:     Create input file in memory
3:     Get start points from completed Step1
4:     MPI_Bcast Bertini input file and Start file to each worker
5:     Get first set of points, from mesh or user-provided file
6:     Distribute first round of points
7:     while points left to run do
8:         MPI_Receive from any source,
9:         Create next set of parameters
10:        MPI_Send new parameters to slave
11:    end while
12:    for each worker do
13:        MPI_Send kill tag
14:    end for
15:    Write timing data to disk
16:    delete temp files
17: end function
```

Demonstrations of use of Paramotopy have been performed, using a variety of systems and machines. The next section describes the features of Paramotopy, and subsequently we will demonstrate program performance.

Algorithm 4 Paramotopy slave function.

```
1: function SLAVE
2:     MPI_Bcast to receive Start and Input files from master
3:     Change directory to initial temp file location
4:     Call initial Bertini using Start and Input, to seed memory structures
5:     Collect all .out files from initial Bertini call
6:     Inform master ready for work
7:     Receive initial round of work
8:     Change directory to working temp location
9:     Write .out files
10:    while Have not received kill tag from master do
11:        while Have points to solve do
12:            Write num.out file for straight line program
13:            Call Bertini library
14:            Read selected output files into buffer
15:            if bufferthreshold exceeded then
16:                Write corresponding file buffer to disk
17:                Clear buffer
18:                Initialize buffer
19:            end if
20:        end while
21:        MPI_Send to master that work is completed
22:        MPI_Receive from master, work or kill tag
23:    end while
24:    Write timing data to disk
25: end function
```

2.2. Program Features. Paramotopy has been implemented in the C++ programming language, relying on the MPICH2 libraries for intercomputer communication, and Boost for filesystem operations. At this time, the program uses a text interface using menus and numeric input for control. Here we will briefly describe some features and usage, and then the technical details of the program. See Appendix C for the User's Manual, and details on compiling, running, and troubleshooting Paramotopy.

Real-world problems may encompass *many* parameters; this could be problematic for discretization of parameter space into a uniform sample. Hence, Paramotopy contains support for both linear uniform meshes of parameters, as well as user-defined sets of parameter values stored in a text file. Systems with few parameters can make excellent use of computer-generated discretizations. In contrast, systems with many parameters could use the Monte Carlo sampling method to obtain useful desired information. To use this functionality, the user must generate a text file containing whitespace-separated real and imaginary pairs for each parameter, with distinct parameter points being on separate lines.

As a piece of scientific software, efficiency and performance are important. To this end, Paramotopy is written with the most accurate timing statements possible built in – those of the OpenMP package are used, specifically `omp_get_wtime()`, having up to nanosecond resolution, depending on the operating system. This enables analysis of performance by process type, which appears below in the demonstration sections (3).

Paramotopy has both serial and parallel capabilities. The user may choose in the preferences whether to use a launcher to execute `step2`, or whether to perform Step2 entirely internally. Since Step2 exists as a separate program from Paramotopy, it is called from Paramotopy via a `system(c_str(command))` instruction. Therefore, we use whatever parallel process launcher exists on the computer. Paramotopy comes equipped to use `mpiexec` and `aprun`, but allows the user to select their own launcher program name. Unfortunately, submission of Step2 to a queue via a script is not supported at this time.

To aid in the creation and management of data, Paramotopy makes individual folders for containing data for multiple runs of the same input file. These run folders may be reloaded in the future, for further work such as Failed Path Analysis. Correspondingly, the user may select any of the Bertini output files to save for each parameter point, including both the computer-suited and human-readable files.

Paramotopy collects the Bertini output file `failed_paths` automatically; the user may not opt out of this. After a Step2 run is completed, the user may enter Failed Path Analysis mode via the main menu. Paramotopy scans the failed path files collected during the run, and marks the points at which failures occurred. Subsequently the user may have Paramotopy re-solve at those points. Prior to the rerun, the user may move to a new random start point. Additionally, the Failed Path portion has its own set of Bertini settings, independent of Step1 and Step2, and the tolerances may be *automatically* tightened by one decimal order of magnitude with each re-solve iteration. Paramotopy will attempt a user-set number of iterations for each round of re-solve, terminating after that number, or when all points have been successfully run with zero path failures.

Bertini settings through Paramotopy are persistent, and user-set. Using an `xml` file located near the user's home directory, Paramotopy maintains preferences from run to run, and session to session. If a critical setting is missing, the program will either provide a default value, or prompt the user for their choice. Should the preferences file become corrupt, Paramotopy falls back to defaults and informs the user. Each of Step1, Step2, and Failed Path Analysis have their own groups of Bertini settings, with a minimal default set. The groups are fully dynamic and mutable; the user can add and delete Bertini settings, as well as change existing settings. This is advantageous, as Bertini has numerous configuration possibilities, and as new modes and options are added to the underlying Bertini program, they may be added with little difficulty directly to Paramotopy.

Paramotopy has been successfully compiled on a CentOS linux cluster named Sam; several Apple OSX machines; and a Cray XT6M 1048-core tier 3 supercomputer. Performance has been proven on Sam, as the laptop Macintoshes available do not have enough processors. The Cray has been tested, but only minimally after inclusion of several major features – namely the use of shared memory ramdisks – which the Cray currently does not appear to support.

The random complex start point used in Step1 has a strong affect on the runtime for subsequent solves. Hence, we have incorporated the ability to save and load initial random complex values. Additionally, the user may generate new values internally, should the need arise. The new point may be in a standard range (the unit cube), or in a custom region of parameter space.

User-specified preferences within Paramotopy contribute greatly to its performance. Discussion and performance analysis of several major parameters appears below in Section 3.2.3, and a User's Manual in Appendix C.

3. Demonstrations and Performance

In the following sections, we demonstrate Paramotopy's capabilities, by showing input files, performance scaling, potential uses, and plots of solution sets.

3.1. Monks Equations.

As derived by Ken Monks in an unpublished work, we have the 'Monks Equations,' which describe the amplitude of interacting waves on an annulus, and are related to growth patterns in cacti and other plants. They are computationally interesting because the number of real solutions varies in parameter space, and some equilibrium solutions are stable and others unstable, as well as being visually appealing. Cutting directly to the differential equations, we have the following square coupled four-complex-dimensional system:

$$\begin{aligned}
\dot{z_0} &= \mu_0 z_0 + \bar{z}_1 z_2 - \gamma z_0(S - |z_0|^2) \\
\dot{z_1} &= \mu_1 z_1 + \bar{z}_0 z_2 + \bar{z}_2 z_3 - \gamma z_1(S - |z_1|^2) \\
\dot{z_1} &= \mu_1 z_2 + \bar{z}_0 z_1 + \bar{z}_1 z_3 - \gamma z_2(S - |z_2|^2) \\
\dot{z_0} &= \mu_0 z_3 + z_1 z_2 - \gamma z_3(S - |z_3|^2),
\end{aligned} \tag{3}$$

where $S = 2\sum_{j=0}^{4} |z_j|^2$. Monks notes that the three parameters μ_0, μ_1, γ must be real, and that $\gamma < 0$. Note that the complex conjugate operator is is nondifferentiable and nonalgebraic. To get around this difficulty, we seek pure real solutions, and drop the bar from each equation.

Paramotopy is useful in describing the dependence on the number of real equilibria in the system in relation to the values of the parameters, as seen in Figure 4. For generic parameter values, the Monks Equations have 81 distinct isolated complex solutions, with no positive dimensional components to the variety corresponding to equilibria solution, as verified using a positive dimensional run in *Bertini*. Rarely enough, the single variable group total degree start system constructed by Bertini happens to have 81 paths to track as well. In the figure, we see four slices along γ values in the three dimensional parameter space, with $(\mu_0 , \mu_1) \in [0 , 10] \times [0 , 10]$; there a clear dependence on parameter values.

The solution set of the Monks equations is also interesting, seeming to perhaps have multiple components, as seen in Figure 5. The projections onto different coordinates coupled with coloring dependent on another variable are nice to look at.

```
4 1 3 0
mu0*z0+z1*z2-gamma*z0*(2*(z0^2+z1^2+z2^2+z3^2)-z0^2)
mu1*z1+z0*z2+z2*z3-gamma*z1*(2*(z0^2+z1^2+z2^2+z3^2)-z1^2)
mu1*z2+z0*z1+z1*z3-gamma*z2*(2*(z0^2+z1^2+z2^2+z3^2)-z2^2)
mu0*z3+z1*z2-gamma*z3*(2*(z0^2+z1^2+z2^2+z3^2)-z3^2)
z0, z1, z2, z3
0
mu0 0 0 10 0 40
mu1 0 0 10 0 40
gamma 0 0 10 0 1
```

File 1.1. Monks system Paramotopy input file.

Using a mesh with 8000 points in it, as 20 points in each of three parameters, this set of equations was used for timing on the Dell 72 CPU cluster, Sam. Results of the tests are in Figure 3. We use the one-worker test timings as the base time for scaling numbers. The main highlight of this figure is the flatlining of the speedup factor at about 30x over a single-worker run. This is far from optimal, as we would prefer the speedup to scale linearly with the number of processors used in the solve. Based on timing analysis, the hangup is due an imbalance in the Paramotopy setting which controls distribution of parameter points to workers. A mere 8000 points distributed across 72 workers is only about 111 points per worker. For the timing runs shown here, the parameter for distribution was set to 112, so each worker received only a single round of work. Furthermore, one of the axes in parameter space has a singularity. Nevertheless, a speedup of 30x over a single worker is excellent, with a terminal efficiency of almost 40%.

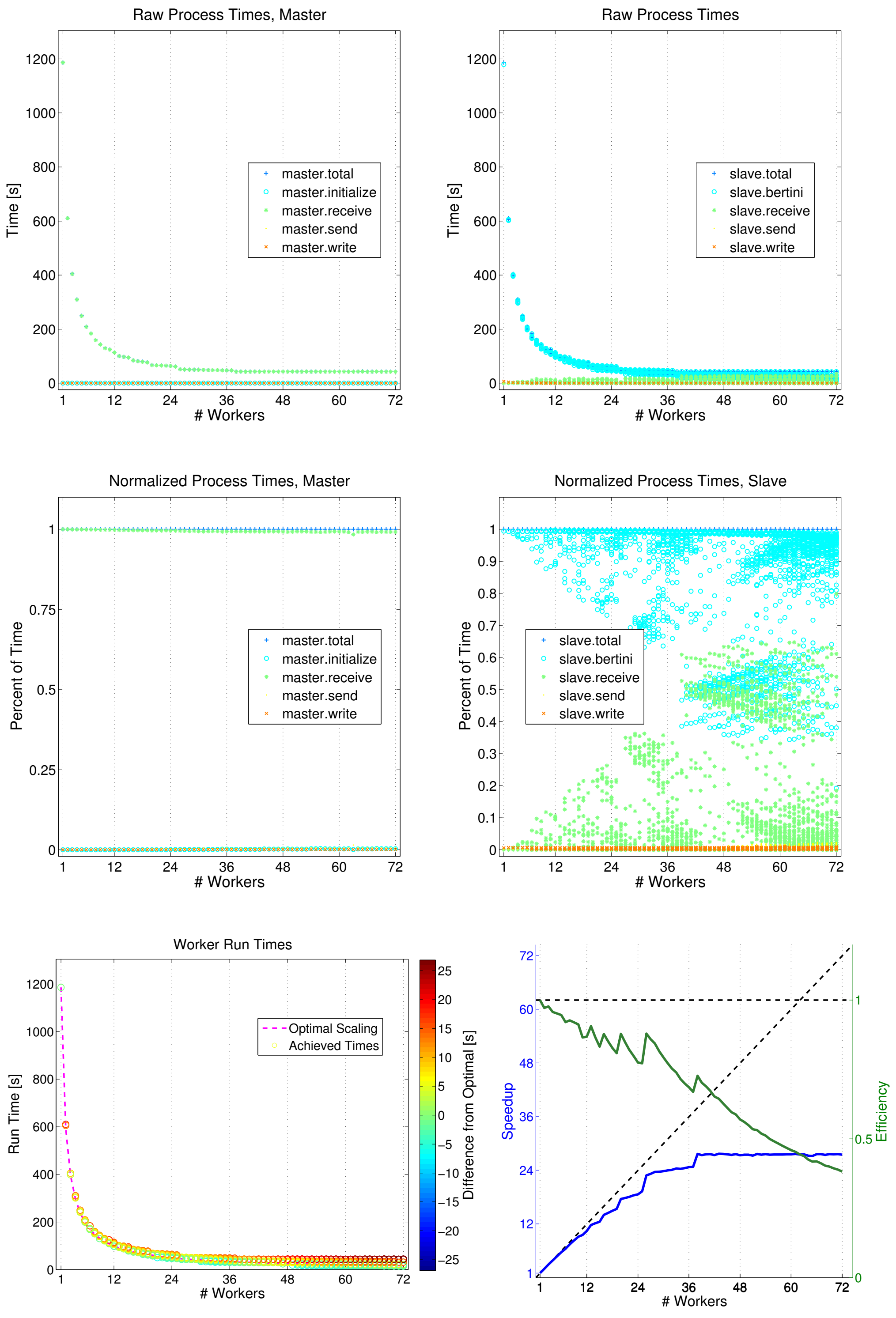
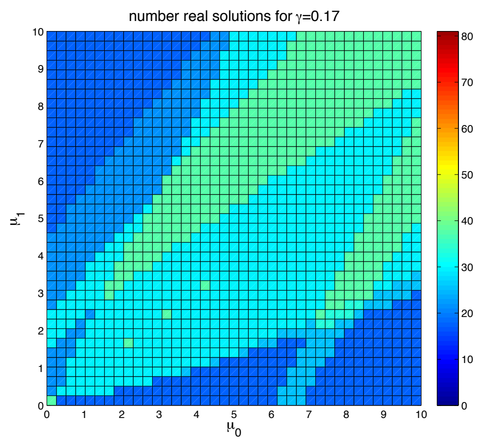
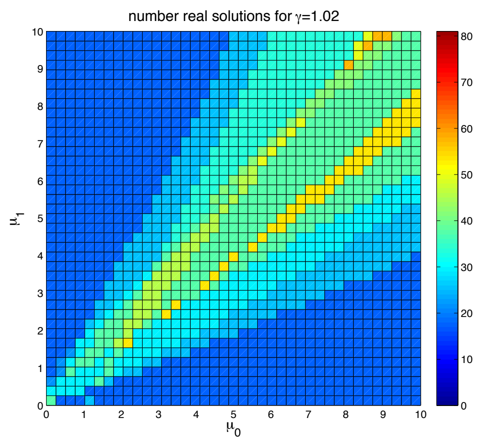


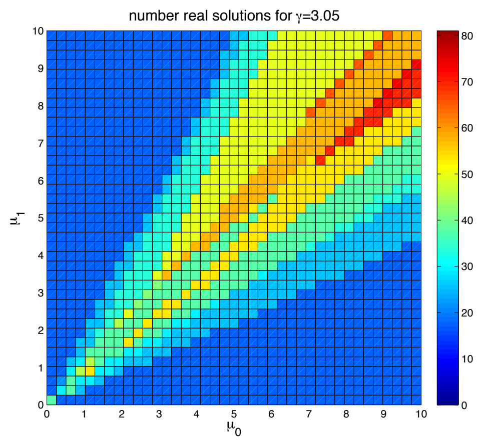
FIGURE 3. Monks equations timing, 8000 parameter points, scaling from 1 to 72 workers. Maximum speedup of ≈30x achieved. The relatively small parameter point sample hinders effective distribution of work.



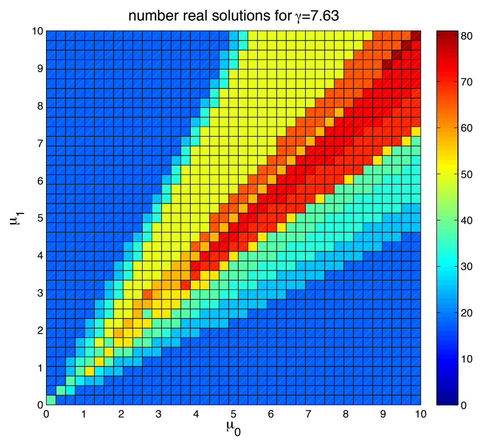
(A)



(B)



(C)



(D)

FIGURE 4. Regions with various numbers of real solutions to the Monks equations in parameter space (μ_0, μ_1, γ). The displayed region is for $(\mu_0\,, \mu_1) \in [0\,, 10] \times [0\,, 10]$. At $\gamma = 0$ there is one solution everywhere, namely the 0 solution; this plot omitted. For low γ, there are few solutions near the parameter-origin, as in (a). As γ increases, regions of higher numbers of solutions appear, and these regions move toward the origin.

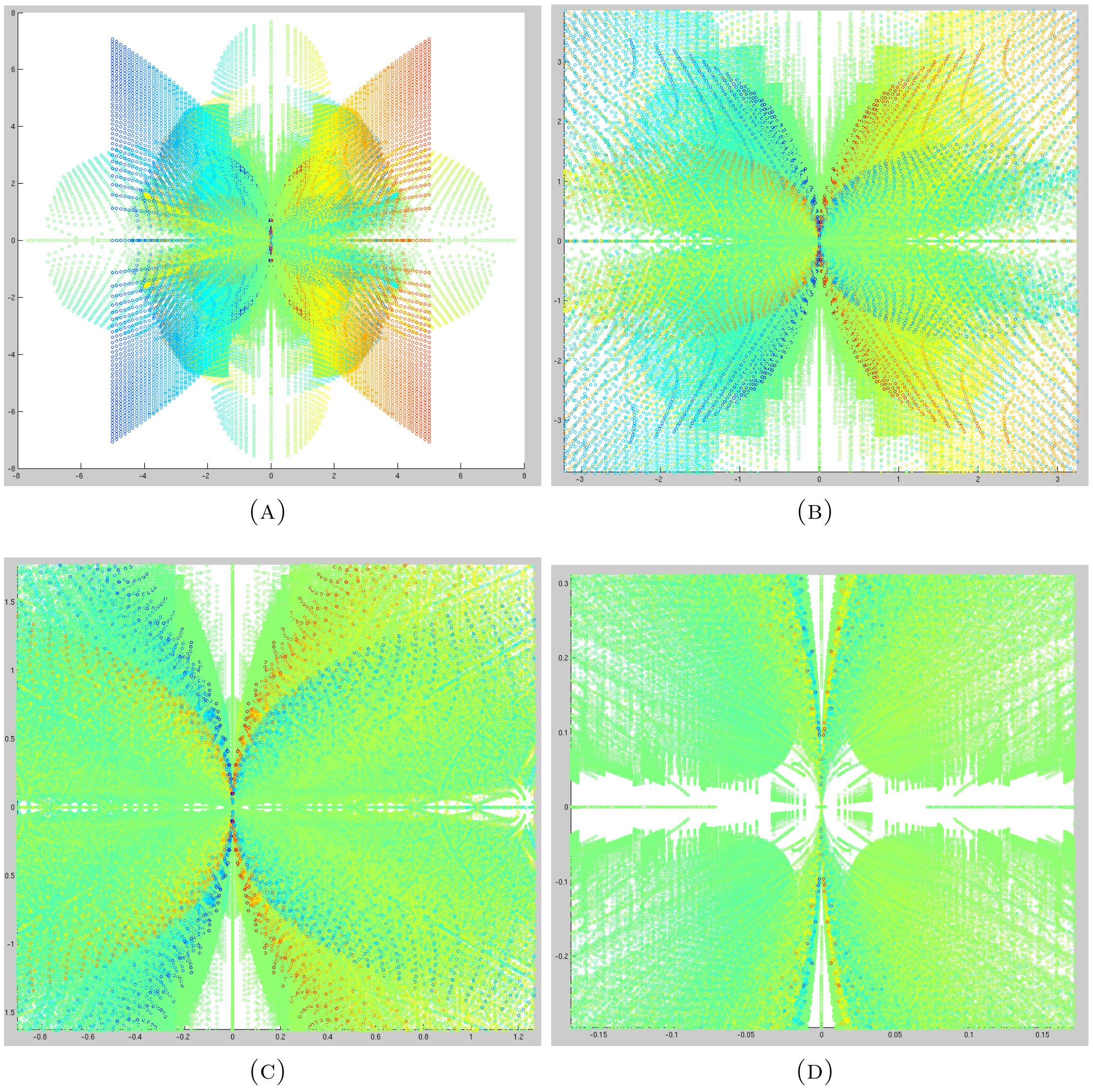


FIGURE 5. Successive zooms into the origin of solution space. Points represent projection onto z_0, z_1, with color determined by z_3. All solutions for the entire scan through parameter space plotted.

3.2. Damped Duffing Oscillators. The following systems are derived from a receding horizon optimization problem, concerned with driving a nonlinear Duffing oscillator to rest [11]. The problem depends on parameters, and can be formulated as a polynomial system, for which we desire the roots. Hence, Paramotopy is a perfect program for solving the system over and over. We will use three versions of the system, corresponding to looking at one, two, and three steps.

3.2.1. 9×9 *Duffing System.*

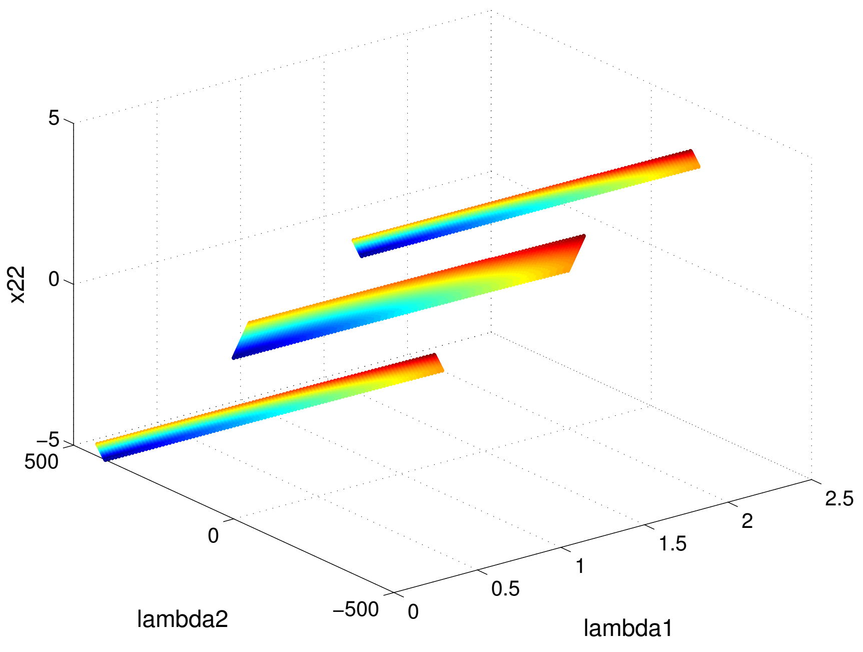
```
9 2 2 0
0.2*u1+.5e-1*lambda2
2*x12-lambda1+mu1-mu2
2*x22-lambda2+mu3-mu4
x11+.5e-1*x21-x12
-.5e-1*x11+.970*x21+.5e-1*u1-.5e-1*x11^3-x22
mu1*(x12-5)
mu2*(-1.0*x12-5)
mu3*(x22-5)
mu4*(-1.0*x22-5)
u1, lambda1, lambda2, mu1, mu2, mu3, mu4
x12, x22
0
x11   0 0 1 0 100
x21   0 0 1 0 100
```

File 1.2. 9×9 Duffing oscillator system Paramotopy input file.

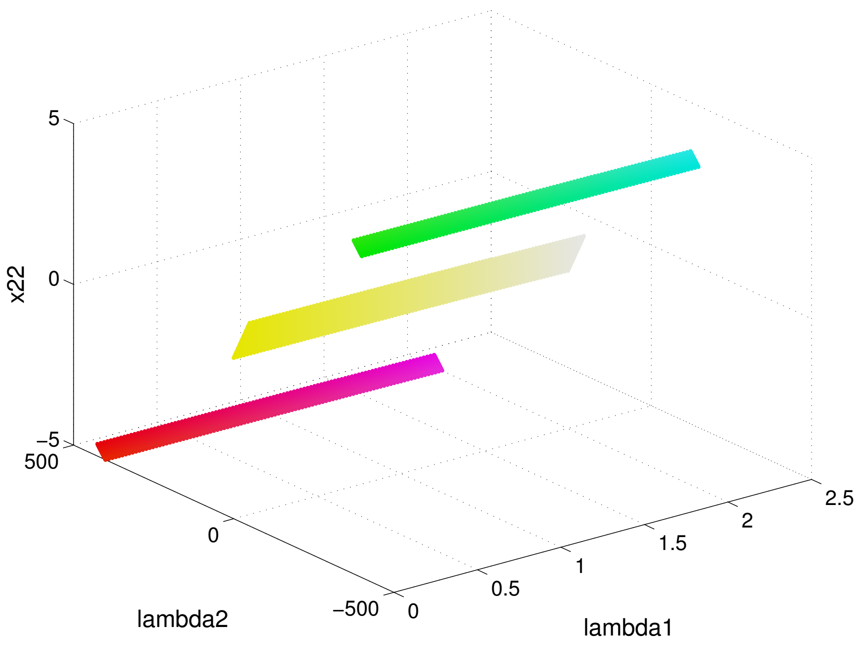
The 9×9 Duffing system is a great test for scaling experiments. Step1 follows seven paths, to three finite nonsingular solutions, and four infinite singular solutions. Step2 follows these three solutions to each point in parameter space, which is two-dimensional.

Timings here are based off a 1000×1000 discretization, in $[0, 1] \times [0, 1]$; see Figure 7. Base case with one worker took about ≈ 7837 seconds, yielding 380 paths per second, or 0.0026 seconds per path. The system is 9×9, but steps of the solve do not appear to be very complex. At peak, this scaling test achieved $72\times$ speedup, and greater than unity efficiency, using 72 workers, and a single master process. Minimum completion time was 109.5 seconds, with 27397 paths per second.

Solutions for the Duffing 9×9 appear to lie on several distinct sheets. See Figure 6. These two plots are projections of solutions to a set of 10000 parameter points. The left hand of the figure demonstrates projections onto (`lambda1, lambda2, x22`), with colors determined by the corresponding parameter point's distance from the origin. On the right, we have the same data projection, but color is instead determined by a RGB triple, composed of normalized values of the three variables (`mu3, mu4, x12`).



(A)



(B)

FIGURE 6. Solution plots for the Duffing 9×9 system. (a) Projections onto variables `lambda1, lambda2, x22`, with points colored according to the corresponding value of $\sqrt{\texttt{x11}^2 + \texttt{x21}^2}$. (b) The same coordinate projections, with coloring coming by letting RGB values be set according to normalized values of `mu3, mu4, x12`.

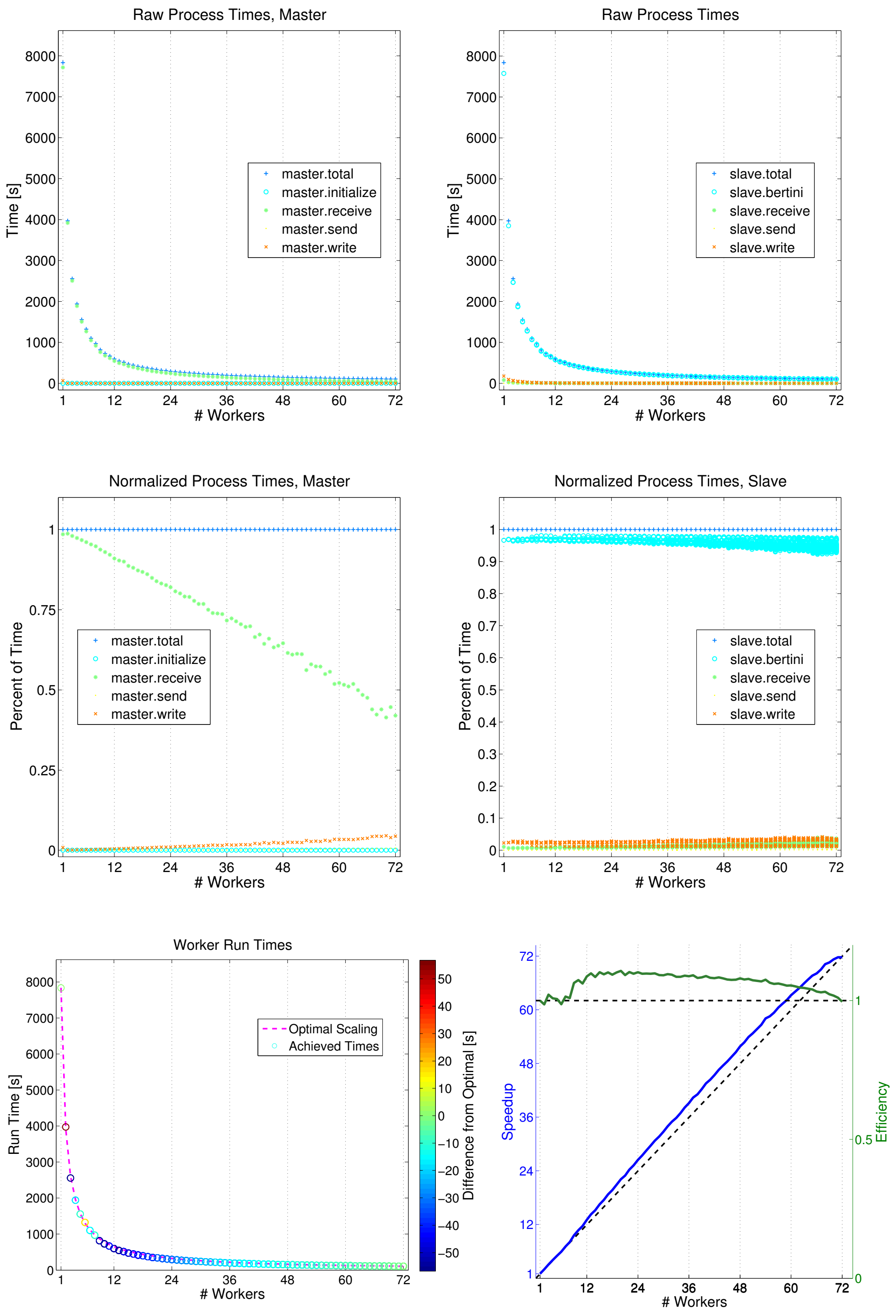
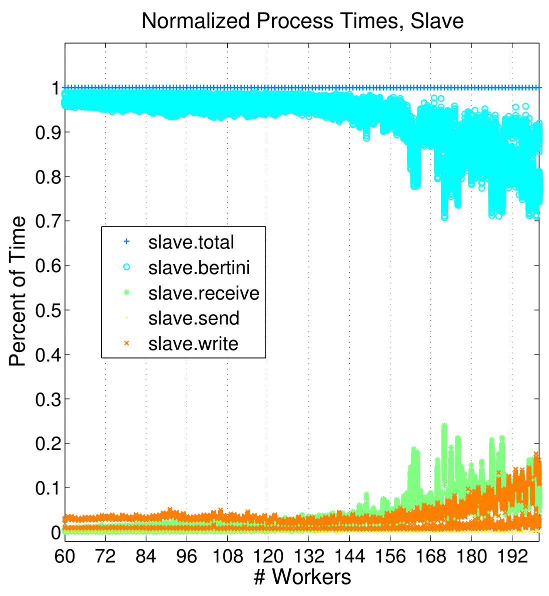


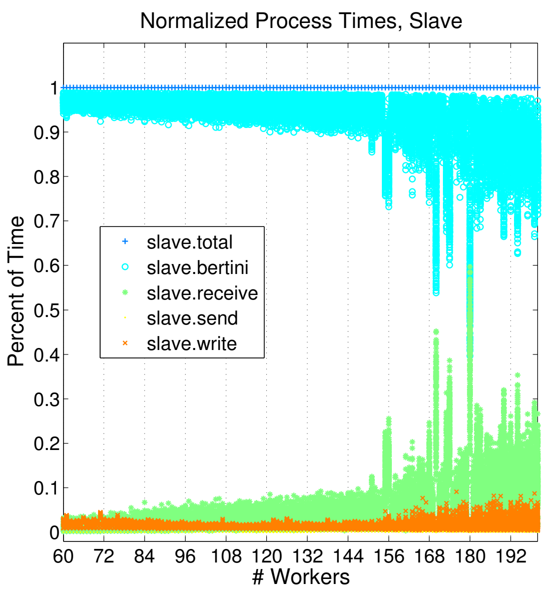
FIGURE 7. Duffing 9×9 timing data. Peak speedup of about $72\times$, with 72 processors. Greater than unity efficiency achieved. Peak rate of almost 27400 paths per second.

3.2.2. *Process Overdistribution.* The computer Sam has at its disposal one twelve-core head node, and six twelve-core worker nodes. To test full capabilities of the computer, we run Paramotopy well over its capacity. These tests revealed that the computer has multithreading support beyond the raw number of CPU's it has. See Figure 8

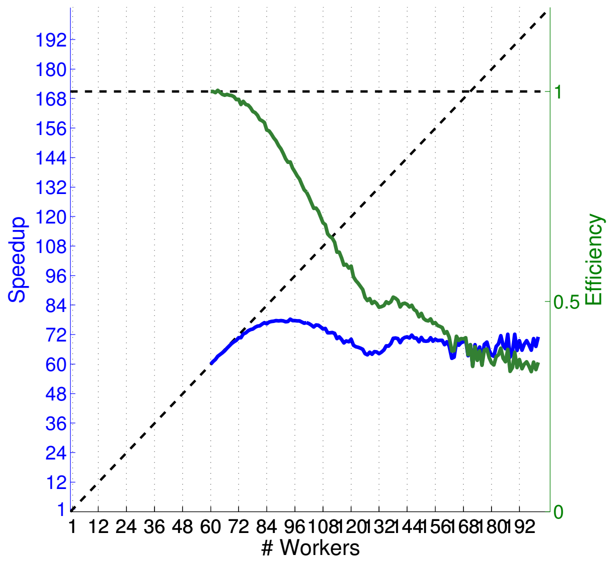
We use the 9x9 Duffing system, with a 5×10^5 point discretization. Beginning from a run with 60 workers, and using its timing as the base for scaling numbers, we scale all the way to 200 workers. Peak speedup is about 78x, at 80 workers, but beyond this, performance drops off. Analysis of process type reveals that beyond the two-fold multithreading capacity of the computer, disk access becomes a barrier, and network throughput becomes limited. Two sets of program parameters were used as well; increasing the data buffer to 1MB from 64KB, and increasing the number of files passed out at a time from 100 to 500, did not noticeably affect the performance of the program.



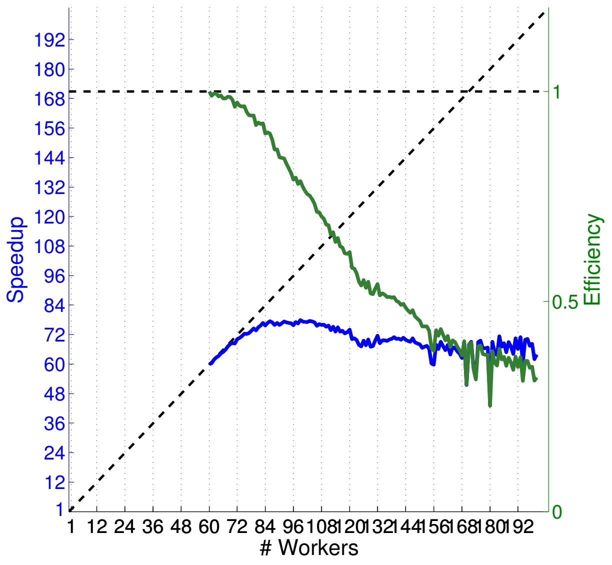
(A) 100 points at a time, 64KB buffer



(B) 500 points at a time, 1MB buffer



(C) 100 points at a time, 64KB buffer



(D) 500 points at a time, 1MB buffer

FIGURE 8. Overdistribution of processes to Sam using the 9×9 Duffing system.

3.2.3. *Buffer Size and Parameter Distribution.* Paramotopy has many user-configurable settings available, of which two have ranges of values rather than simply on-off choices, and greatly influence the performance of the program.

As Step2 runs, its many processes on many computers call Bertini from a library *ad nauseum*, generating a plethora of output files. These output files are consolidated into a set of files, with each worker writing its own data, so as to minimize network traffic. If the output from each call to Bertini is written immediately to disk, the hard drive head cannot handle the near-constant access. Hence, to free the hard drive, the workers read Bertini output into memory immediately upon production, and hold it until the buffer of user-determined size is filled. Upon filling the buffer, the worker writes that entire group of output to disk, purges memory, and again begins creating and reading data.

Early versions of Paramotopy wrote output to disk immediately, without buffering, and performance was poor. This corresponds to the red corner in Figure 9, where the buffer stores only 1024 characters before dumping to disk. The current Paramotopy allows the user to specify a number of KB for the buffer size. Note that the buffer size is a *minimum threshold*, so that the amount of data written at once is larger than the preference, except at the end of a run when there is no more data to be added to the stream. The hardcoded upper limit on buffer size is 64MB.

The second setting available to the user under investigation here concerns the distribution of parameter points to the workers from the master node. To minimize network traffic and achieve load balancing, the master parcels out work in even chunks, of user-specified size. Consider this example: The user specified to have chunk size be $c = 100$, and there were a total of 3000 points, with 12 workers. Initially the master seeds each worker with 100 points, with 1800 remaining. Perhaps worker 2 reports back first, so master passes it 100 more points. Then worker 5 completes its first round, and master sends it 100 more points. This continues, in first-come-first-served fashion, until all 3000 points have been handed out.

Once all work has been distributed, the master sends each worker a message containing a tag indicating that the solve is over, and each worker finalizes its data files, writes timing data if suitably compiled, and exits.

This `numfilesatatime` parameter influences the solve time, but not near so much as the data buffer. Setting the parameter to `1` can slow the program down, especially if the number of workers is large and path track time is low. In this case, workers report back almost immediately, and workers spend their time idly waiting for more work. In contrast, if the `numfilesatatime` setting is too large, all the work will be passed out immediately; in the extreme case only several or even a single worker will receive points to solve, and the remaining workers do nothing but exit.

The Figure 9 demonstrates run time versus a set of parameters. The two horizontal axes in this figure are logarithmic. A total of 299 Paramotopy runs were executed for this plot, with run times ranging from a peak of 128 seconds, to a minimum of 26 seconds. Not surprisingly, the maximum occurred with a buffer size of 1KB, and only a single parameter point being passed out to each worker at a time. The ridge of slightly higher times appearing at `buffer`=4096KB is unexplained, but the minimum was achieved for the maximum value of both parameters.

Optimal settings are easy to achieve. Simply attempt to divide work so that the distribution of points happens faster than solving a group of points, and keep the buffer size moderately large, and runtimes will be low. As can be seen in Figure 9, the region of slow runtimes is bounded near the axes for both parameters.

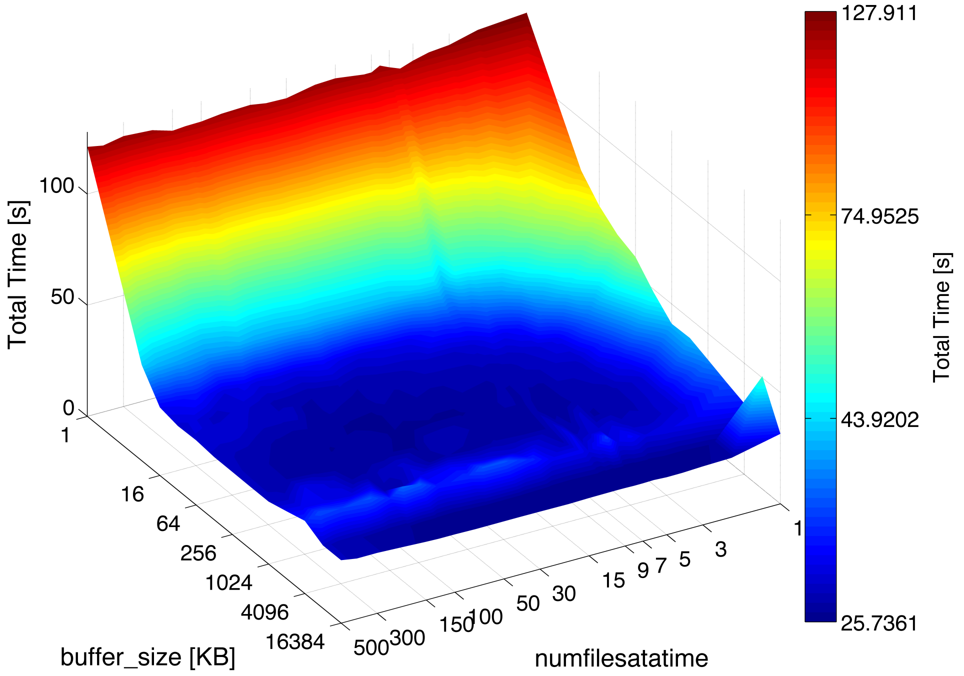


FIGURE 9. Testing Paramotopy run times versus internal parameters, using a 500000 parameter point sample, in the 9×9 Duffing system. The horizontal axes are logarithmic, with linear vertical axis corresponding to run time. Coloration is logarithmic in runtime. Optimal settings are bounded away from the origin.

3.2.4. *Duffing 18×18.* The Duffing 18×18 system is the same as the damped controlled 9×9 system, but is sees one step further toward the horizon. There are 18 equations in 18 unknowns.

```
18 2 2 0
0.2*u1+0.05*lambda2
2.0*x12−lambda1+lambda3+lambda4*(−0.05−0.15*x12^2)+mu1−mu2
2.0*x22−lambda2+0.05*lambda3+0.970*lambda4+mu3−mu4
0.2*u2+0.05*lambda4
2.0*x13−lambda3+mu5−mu6
2.0*x23−lambda4+mu7−mu8
x11+0.05*x21−x12
−0.05*x11+0.970*x21+0.05*u1−0.05*x11^3−x22
x12+0.05*x22−x13
−0.05*x12+0.970*x22+0.05*u2−0.05*x12^3−x23
mu1*(x12−5.0)
mu2*(−1.0*x12−5.0)
mu3*(x22−5.0)
mu4*(−1.0*x22−5.0)
mu5*(x13−5.0)
mu6*(−1.0*x13−5.0)
mu7*(x23−5.0)
mu8*(−1.0*x23−5.0)
u1, u2, lambda1, lambda2, lambda3, lambda4, mu1, mu2, mu3, mu4,
    mu5, mu6, mu7, mu8
x12, x13, x22, x23
0
x11   −1  0  1  0  50
x21   −1  0  1  0  50
```

FILE 1.3. Paramotopy input file for 18×18 Duffing system.

Timing runs using a mesh of 1×10^6 are demonstrated in Figure 10. Again, linear speedup was achieved with efficiency ≈ 1 throughout the range of processors. There is an unexplained dip in timing near 13 and 14 processors, and breakdown of timing by process type reveals an increase in writing time – we therefore conclude that something else was using the disk at that time. The master spends most of its time waiting for the slaves to ask for more work, and the trends in the timing indicate that Paramotopy in this case would have scaled well even beyond 72 workers. Unfortunately, the Cray computer was incapable of exploiting shared memory at the time of this writing, and timing tests were not performed.

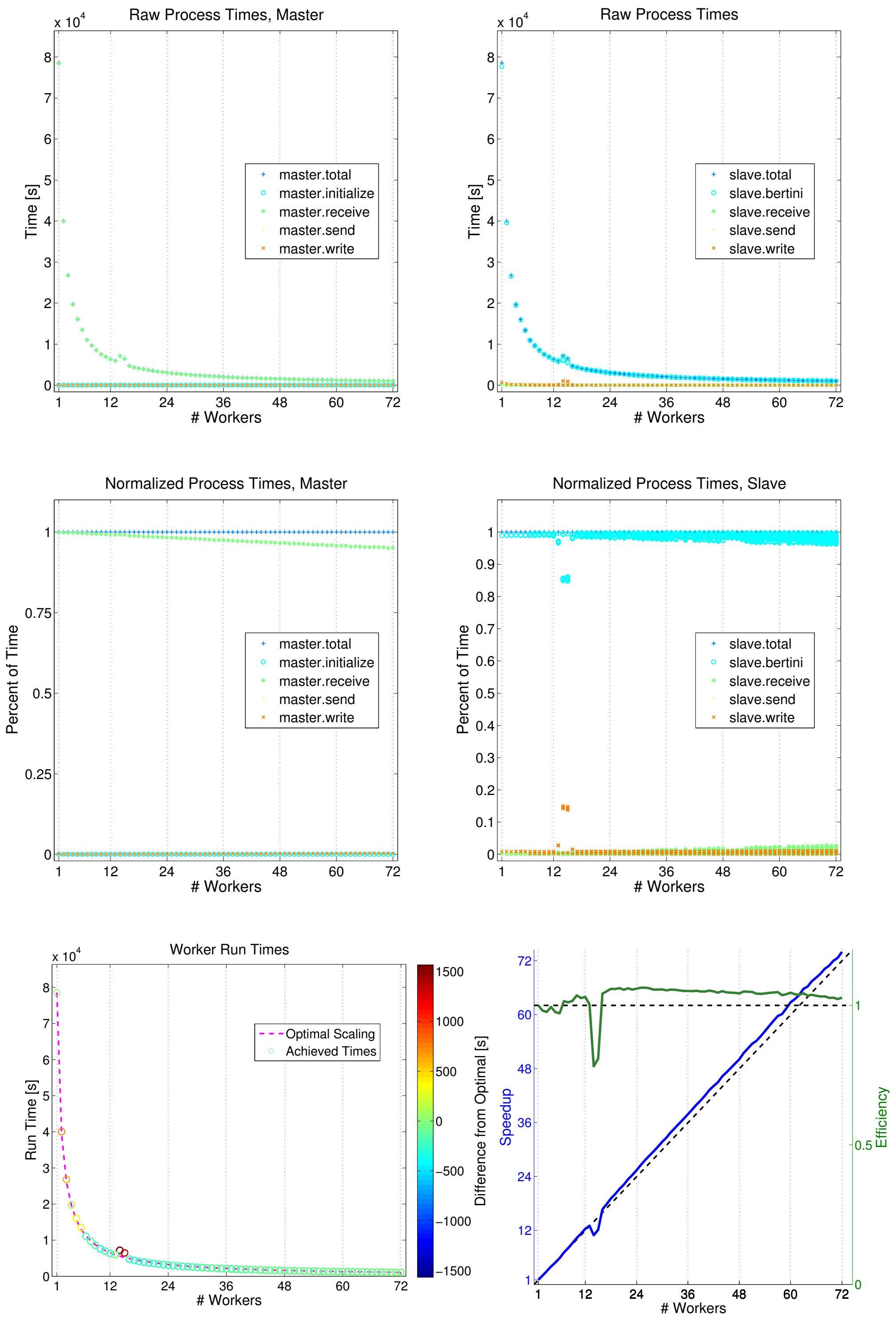
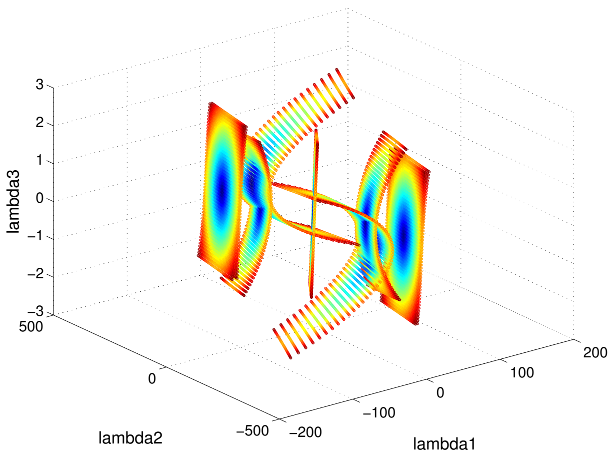
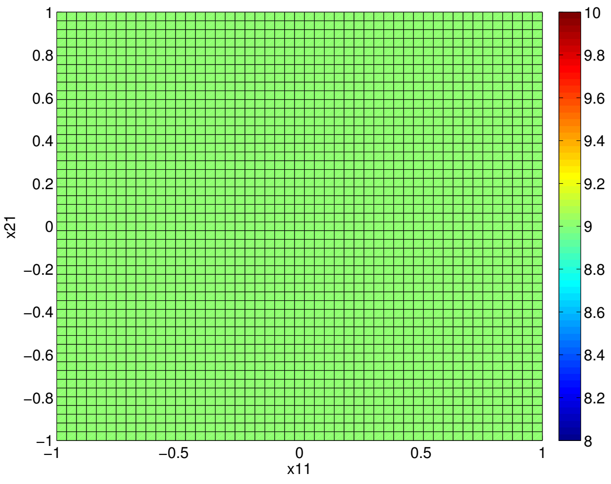


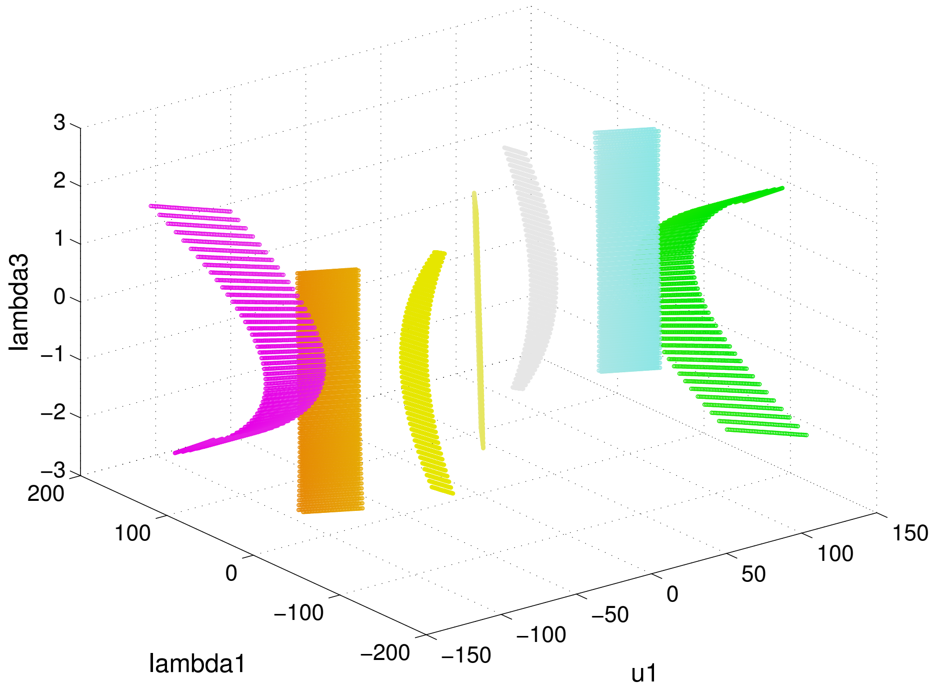
FIGURE 10. Duffing 18×18 timing results. Linear speedup achieved throughout the range of processors used. The small blip at 13 processors indicates another process was using the hard drive, inhibiting Paramotopy performance.



(A)



(B)



(C)

FIGURE 11. Duffing 18×18 solution plots. (a) Projection of solutions onto coordinates (`lambda1`, `lambda2`, `lambda3`), with color determined by corresponding parameter distance to the origin. (b) The parameter space uniformly has nine real solutions. (c) Projection onto (`lambda1`, `u1`, `lambda3`), with RBG value determined by a triple composed of normalized values of (`mu3`, `mu4` , `x23`)

3.2.5. *Duffing* 27×27. The Duffing 27×27 system is akin to the previous two Duffing systems, in that it is a receding horizon optimization problem. This adds one step beyond the 18×18 system.

```
27 2 2 0
1/5*u1+.5e-1*lambda2
2*x12-lambda1+lambda3+lambda4*(-.5e-1-.15*x12^2)+mu1-mu2
2*x22-lambda2+.5e-1*lambda3+.970*lambda4+mu3-mu4
1/5*u2+.5e-1*lambda4
2*x13-lambda3+lambda5+lambda6*(-.5e-1-.15*x13^2)+mu5-mu6
2*x23-lambda4+.5e-1*lambda5+.970*lambda6+mu7-mu8
1/5*u3+.5e-1*lambda6
2*x14-lambda5+mu9-mu10
2*x24-lambda6+mu11-mu12
x11+.5e-1*x21-x12
-.5e-1*x11+.970*x21+.5e-1*u1-.5e-1*x11^3-x22
x12+.5e-1*x22-x13
-.5e-1*x12+.970*x22+.5e-1*u2-.5e-1*x12^3-x23
x13+.5e-1*x23-x14
-.5e-1*x13+.970*x23+.5e-1*u3-.5e-1*x13^3-x24
mu1*(x12-5)
mu2*(-1.0*x12-5)
mu3*(x22-5)
mu4*(-1.0*x22-5)
mu5*(x13-5)
mu6*(-1.0*x13-5)
mu7*(x23-5)
mu8*(-1.0*x23-5)
mu9*(x14-5)
mu10*(-1.0*x14-5)
mu11*(x24-5)
mu12*(-1.0*x24-5)
u1, u2, u3, lambda1, lambda2, lambda3, lambda4, lambda5, lambda6
    , mu1, mu2, mu3, mu4, mu5, mu6, mu7, mu8, mu9, mu10, mu11,
    mu12
x12, x13, x14, x22, x23, x24
0
x11 -1 0 1 0 50
x21 -1 0 1 0 50
```

File 1.4. Duffing 27×27 Paramotopy input file.

Duffing27 has an initial path count of 2729, tracking to 32 solutions, of which 27 are nonsingular and finite, and 5 are singular and finite. Hence, the best file to choose for our start for Step2 is `nonsingular_solutions`.

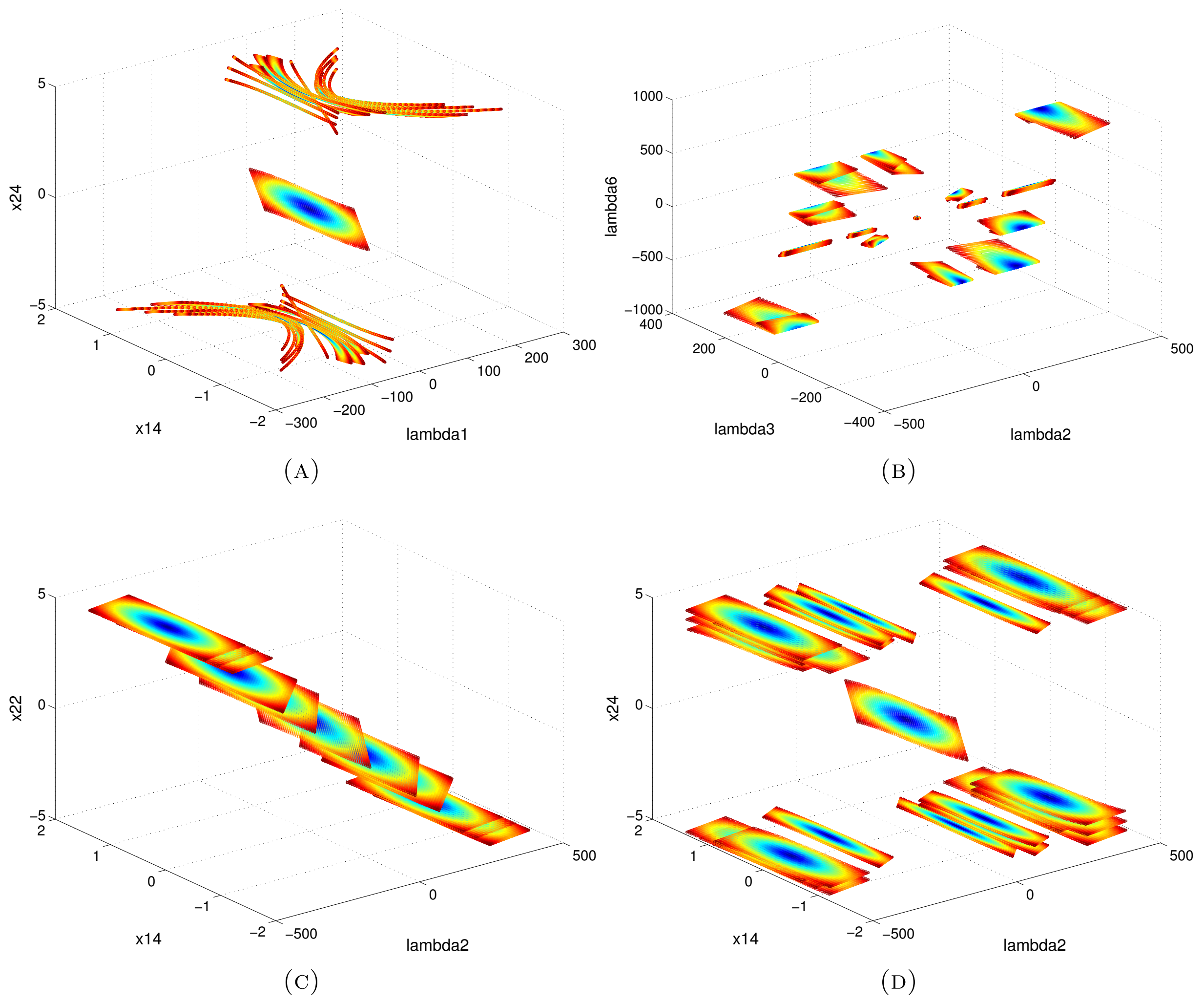


Figure 12. Projections of the Duffing 27×27 system onto various combinations of variables. In all four plots shown here, colors of points correspond to the distance to the parameter point from the origin. The system generically has 27 nonsingular finite solutions at each point in parameter space.

Plotting solutions based on a scan in the parameters (`x11`, `x21`) reveal a complicated structure; see Figure 12. There appears to be a component near the origin for each combination of variables on which we project. The origin in parameter space is mapped to the 'center' of each component in variable space.

3.3. Chemical Network Equilibria. As a final demonstration of the capabilities of Paramotopy, we verify the results of the application of a theorem to dynamical systems in [12]. The application of a modified 'DetSign' condition to a chemical reaction system, that is to say the strict positiveness or negativeness of the determinant, allows the authors to prove the uniqueness of a positive equilibrium in the domain of validity.

The following examples are numbered as they appear in [12].

3.3.1. *Example 2.2.* Example 2.2 from Craciun et al, originally from [13], given in equations (2.3) - (2.5) is:

$$\begin{aligned}
\dot{c}_1 &= \frac{p_1 c_0}{p_2 + c_3} - p_3 c_1 \,, \\
\dot{c}_2 &= p_3 c_1 - p_4 c_2 \,, \\
\dot{c}_3 &= p_4 c_2 - \frac{p_5 c_3}{p_6 + c_3} \,.
\end{aligned}$$

The claim is that for all parameter values in the unit cube, the dynamical system has a unique equilibrium. Note that these are not strictly polynomials – they are more of a Laurent style polynomial. We will turn them into polynomials, by setting the left hand side to be 0, and clearing denominators.

$$\begin{aligned}
0 &= p_1 c_0 - (p_2 + c_3) p_3 c_1 \,, \\
0 &= p_3 c_1 - p_4 c_2 \,, \\
0 &= (p_6 + c_3) p_4 c_2 - p_5 c_3 \,.
\end{aligned}$$

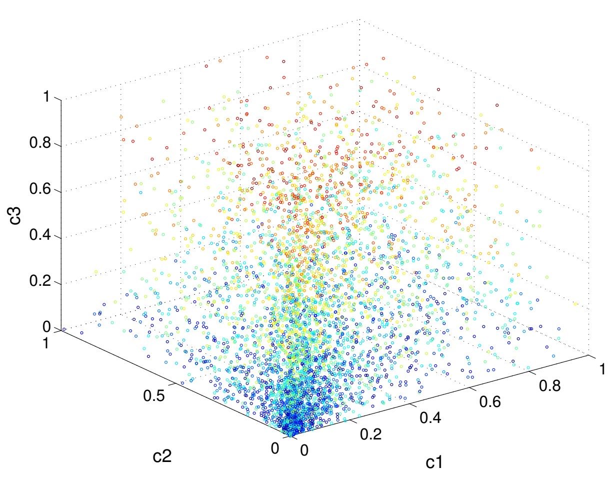
```
3 1 7 0
p1*c0 - (p2 + c3)*p3*c1
p3*c1-p4*c2
(p6 + c3)*p4*c2 - p5*c3
c1,c2,c3
1
network_monte
c0
p1
p2
p3
p4
p5
p6
```

FILE 1.5. Input file for Example 2.2, Craciun, Helton, Williams, for the random sampling of parameter space.

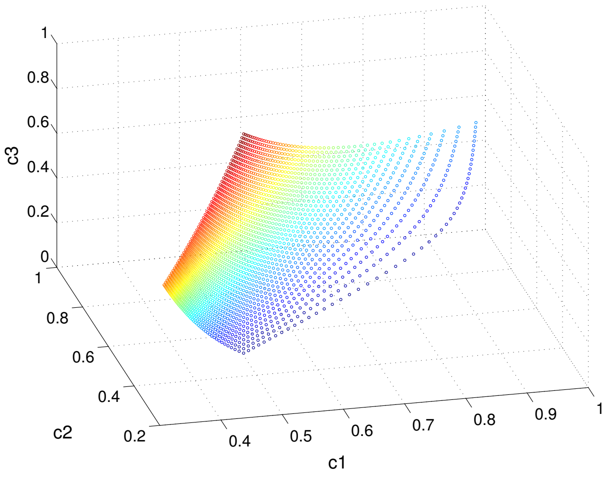
For this chemical reaction network, Bertini tracks 4 initial paths total, to 2 finite nonsingular solutions, with multiplicity 1, and 2 infinite nonsingular solutions. As an aside, for this Step2 we will follow the `finite_solutions` file from Step1 since the `nonsingular_solutions` file contains two infinite solutions, which cause path failures during Step2.

To test the claims of Craciun, Helton, and Williams, we sample the seven dimensional parameter space 10,000 times randomly. Because Bertini finds *all* solutions, we perform a post-run data sort to eliminate all equilibria not in the domain of validity. Sure enough, each and every one of our random samples has exactly one equilibria in the unit cube. Figure 13 shows a pair of scatter plots of the equilibria in variable space. One is the solution set corresponding to the random sampling, and the other is generated by a 50x50 mesh in parameters `p2` and `p6`. In both the shown plots, the colors of the points correspond to the value of `(p6+c3)`.

3.3.2. *Example 2.3.* A similar claim is made regarding Example 2.3, which is a model for "mitogen activated protein kinase (MAPK) cascades with inhibitory feedback". The system is originally from [14, 15, 16]. Equations (2.6) - (2.8) from [12] are:



(A)



(B)

FIGURE 13. Craciun, Helton, Williams, Example 2.2. Each parameter combination yields a unique equilibrium in the unit cube. (a) Random sampling solution set. (b) Mesh solution set. Coloring is determined by the value of (`p6` + `c3`).

$$\dot{c}_1 = -\frac{b_1 c_1}{c_1 + a_1} + \frac{d_1(1-c_1)}{e_1 + (1-c_1)}\frac{\mu}{1+kc_3}\,,$$

$$\dot{c}_2 = -\frac{b_2 c_2}{c_2 + a_2} + \frac{d_2(1-c_2)}{e_2 + (1-c_2)}c_1\,,$$

$$\dot{c}_3 = -\frac{b_3 c_3}{c_3 + a_3} + \frac{d_3(1-c_3)}{e_3 + (1-c_3)}c_2\,.$$

Clearing denominators, we get,

$$0 = -b_1c_1(e_1 + (1-c_1))(1+kc_3) + (c_1+a_1)d_1(1-c_1)\mu\,,$$

$$0 = -b_2c_2(e_2 + (1-c_2)) + (c_2+a_2)d_2(1-c_2)c_1\,,$$

$$0 = -b_3c_3(e_3 + (1-c_3)) + (c_3+a_3)d_3(1-c_3)c_2\,.$$

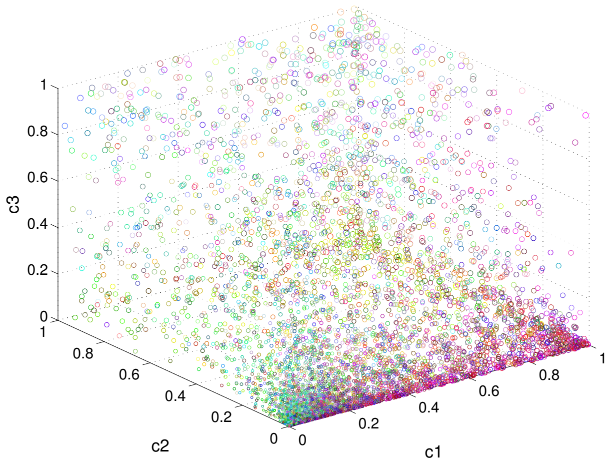
```
3 1 14 0
-b1*c1*(e1 + (1-c1))*(1 + k*c3)  + (c1 + a1)*d1*(1 - c1)*mu
-b2*c2*(e2 + (1 - c2)) +  (c2 + a2)*d2*( 1- c2)*c1
-b3*c3*(e3 + (1 - c3)) +  (c3 + a3)*d3*(1 - c3)*c2
c1,c2,c3
1
network_monte10000
a1
a2
a3
b1
d1
e1
b2
d2
e2
b3
d3
e3
mu
k
```

File 1.6. Input file for Example 2.3, Craciun, Helton, Williams, for the computer-generated mesh.

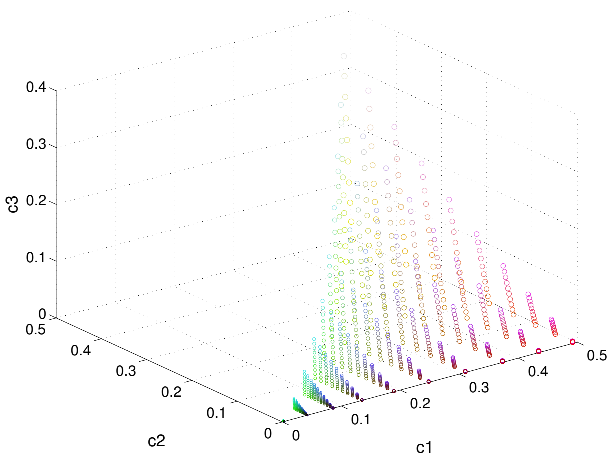
The Bertini Step1 run tracks 27 paths to 9 finite nonsingular solutions, and 3 infinite singular solutions each with multiplicity 6. Hence, each parameter point has 9 paths to track.

We sample the parameter space 10,000 times in the strictly positive domain, and test the claims given in the paper; namely, that each parameter value has exactly one equilibrium value in the unit cube, $c_i \in [0, 1]$. Performing post-solve sorts, we verify the claim – there is indeed a unique equilibrium in the unit cube. Figure 14 demonstrates both the equilibria for the each parameter in the random sampling, as well as a 10x10x10 three-parameter unit-cube mesh in `d1`, `d2`, `d3`.

3.3.3. *Example 6.1.* As a final demonstration of using Paramotopy to investigate claims regarding equilibria of dynamical systems, we check the claim in Example 6.1 from Craciun



(A)



(B)

FIGURE 14. Chemical equilibria solution plots for Craciun Helton Williams Example 2.3. Colors of points are determined by letting the RGB value be determined by normalized values of (`d1`, `d2`, `d3`). Hence, the red colored points are heavy in `d1`, whereas green are heavy in `d2`, etc.

Helton Williams.

$$\dot{c}_A = k_{0\to A} - k_{A\to 0}c_A - k_{A+B\to P}c_Ac_B + 2k_{C\to 2A}c_C$$

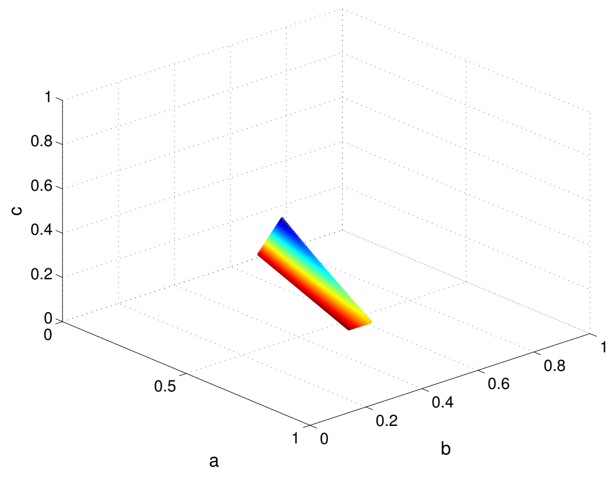
$$\dot{c}_B = k_{0\to B} - k_{B\to 0}c_B - k_{A+B\to P}c_Ac_B - k_{B+C\to Q}c_Bc_C$$

$$\dot{c}_C = k_{0\to C} - k_{C\to 0}c_C - k_{B+C\to Q}c_Bc_C - k_{C\to 2A}c_C$$

$$\dot{c}_P = k_{0\to P} - k_{P\to 0}c_P + k_{A+B\to P}c_Ac_B$$

$$\dot{c}_Q = k_{0\to Q} - k_{Q\to 0}c_Q + k_{B+C\to Q}c_BAc_C$$

Bertini tracks 32 initial paths to 3 finite nonsingular solutions, and 20 infinite singular solutions with multiplicity 1, and 3 infinite singular solutions with multiplicity three. A random sampling of 1000 parameters values verified that the system does in fact have a positive equilibrium for each parameter value. However, in contrast to a note by Craciun et al, we found no parameter values with multiple positive equilibria. Perhaps those parameters lie outside the unit cube.



(A)

FIGURE 15. Solution Plot for Craciun, Helton, Williams Example 6.1. This five-variable, 13-parameter system has a unique positive equilibrium for each parameter point. Here, we have projected onto coordinates $(\mathtt{a}, \mathtt{b}, \mathtt{c})$, with coloring coming from $\sqrt{\mathtt{p}^2 + \mathtt{q}^2}$.

Figure 15 shows a 100x100 two-parameter scan in `kbctoq`, `kctoa`. The solutions seem to form a sheet in solution space. Coloration is determined by $\sqrt{\mathtt{p}^2 + \mathtt{q}^2}$.

```
5 1 13 0
kzeroa − kazero∗a − kabtop∗a∗b + 2∗kctoa∗c
kzerob − kbzero∗b − kabtop∗a∗b − kbctoq∗b∗c
kzeroc − kczero∗c − kbctoq∗b∗c − kctoa∗c
kzerop − kpzero∗p + kabtop∗a∗b
kzeroq − kqzero∗q + kbctoq∗b∗c
a,b,c,p,q
0
kzeroa 0.5 0 0.5 0 1
kazero 1 0 1 0 1
kzerob 0.5 0 0.5 0 1
kbzero 1 0 1 0 1
kzeroc 0.5 0 0.5 0 1
kczero 1 0 1 0 1
kzerop 0.5 0 0.5 0 1
kpzero 1 0 1 0 1
kzeroq 0.5 0 0.5 0 1
kqzero 1 0 1 0 1
kabtop 0.5 0 0.5 0 1
kbctoq 0.01 0 0.99 0 100
kctoa 0.01 0 0.99 0 100
```

FILE 1.7. Input file for Craciun, Helton, Williams example 6.1

4. Conclusion

Paramotopy, the parallel wrapper around Bertini, has many uses and is proven in performance. We have demonstrated the use of Paramotopy for investigating the number of isolated equilibria in dynamical polynomial systems, counting the numbers of solutions to systems depending on parameters, and examining the structure of solution sets.

We have achieved speedup of up to $80\times$ over running a single worker with one master node, exploiting shared memory for temporary files, and buffers for reading data upon generation. Equidistribution of tasks across a computer has been demonstrated as well, and has been achieved for even moderately sized parameter samplings – and certainly for large samplings.

Path failure re-solves can be run by the user, with automatic tolerance tightening, and the ability to move around in parameter space with new Step1 runs before each re-solve attempt.

The author has provided material for support of anyone who desires to use Paramotopy, including a User's Manual, and generic Matlab functions. The Matlab functions include code for collection of data from solutions files, and display of projections of solution sets colored in various ways, and numbers of solutions plotted against parameter values.

Chapter 2

Single Molecule Detection through Intrinsic Localized Modes

The goal of the analysis presented here is to support an experimental team in performing 'basic research' to develop the technology for single molecule detectors. The hypothesis of the detector is as follows. An array of nano-scale crystalline pillars [17], imaged via Fourier holography [18], and vibrated via some arbitrary mechanism, could be used as a sensor. If a suspected contaminant or material, which is known to attach to the pillar material, is passed over the array while being driven, and does in fact attach to one or more pillars, the exhibited wave form will shift. Furthermore, if the array acts as a system of coupled nonlinear oscillators, and the parameters of the array are suitable, a phenomenon known as Intrinsic Localized Mode (ILM) will form and pin to the attachment site. This ILM affects the entire array. By monitoring the amplitude of vibration for a set of the pillars, molecule detection is achieved with amplitude shift.

To excite the crystal array, various forcing methods have been suggested, and some are more immediately realistic than others. Mechanical driving could be achieved by attaching the base of an array to a piezo electric crystal, which would provide oscillatory energy at a moderately controllable frequency. However, piezo devices tend to vibrate in a random fashion, and devices operating in the megahertz regime are not all that common, so they might not be the best choice.

Magnetic or electric forcing could be an alternative to piezo driving. If the oscillators were able to be permanently magnetized, then a varying electromagnetic field could be used. Fluctuating fields of the desired frequency range could easily be generated, and the direction of forcing more tightly controlled. The difficulty with this is whether a permanent field could be established on the oscillators; fundamentally this is a materials problem, and it appears that a solution might be realized as in [19].

Localized modes appear when oscillator arrays have defects, as seen in early research and noted in Section 1. To exploit this property, consider this: ILMs on an array are a global phenomenon - they act to concentrate the energy of the full array at a particular location. This is why they are called 'localized' – they localize the energy, rather than leaving it distributed. Hence, if a molecule or particle were to attach to an element in an array, creating a defect at that element, the global vibrational properties of the entire array will change. Additionally, this means that we can monitor the global properties of the array for change, rather than focusing solely on individual pillars.

One of the basic properties of oscillators is that when the scale of the vibrating object is shrunk, the frequency goes up. Miniaturization of arrays to the level which would permit single molecule detection will therefore necessitate the development of new imaging techniques. Traditional lasing using visible light frequencies do not shoot packets of photons with a short enough burst – the 'flash' of the photograph is too long. Hence, this study employs the NSF Engineering Research Center for Extreme Ultraviolet (EUV) Science and Technology at Colorado State University. The particular laser in use is as reported in [20, 21]. The 46.9 nm wavelength corresponds to a frequency of 6.4×10^{15} Hz, and a cycle time of 1.5644×10^{-16} seconds. The spatial resolution for visualization of this laser system as of 2005 was 120-150 nm [21], and operates by shooting the laser over the object, through a zone plate, and onto a CCD. The EUV laser can also be used to pattern materials such as PMMA plastic, down to a resolution of about $\lambda/2 = 28$ nm [22].

Previous studies employing micromechanical cantilevers for use in sensing have monitored frequency shift upon attachment of particles or molecules [23]. In contrast, we propose to use *amplitude modulation* of a 1D array of bidirectional oscillators as an ILM single molecule detection method; this is practically a consequence of the EUV Holography visualization scheme.

1. Background

A physical or electronic structure subjected to periodic or quasi-periodic forcing tends to vibrate with one or several pronounced frequencies; this has been well known for centuries, and these are the *resonant* frequencies. As technology has become more refined, we have been able to develop systems of oscillators, coupled or uncoupled, damped or undamped, linear or nonlinear, with various forcing schemes. With these advancements, the early model for oscillation – the simple harmonic oscillator – has long since ceased to be sufficient for analysis of more complicated systems.

Various special or interesting waveforms emerge in different types of oscillating media. Strings, wires and the like form standing, traveling, and transverse waves; continuous bulk media such as water or other liquids can form waves called 'solitons' which propagate without the typical additive interference properties; and crystals exhibit behaviour described as 'phonons'. The waveform with which we are concerned is called Intrinsic Localized Mode (ILM), and is characterized by:

(1) Temporal *periodicity*,
(2) Spatial *localization* of energy.

The existence of ILMs has been demonstrated in many coupled systems of oscillators, but were first discovered in anharmonic pure crystals, as reported in the seminal papers [24, 25]. Since their initial discovery, ILMs have been found in a wide variety of physical and simulated structures, having a variety of nonlinearities; *e.g* linear oscillator lattices with cubic or quartic terms in the Hamiltonian [26]. Many different forms of ILM have been developed and investigated, such as 'twisted un/staggered modes' [27] and 'bright compact breathers' [28]. Furthermore, there has been substantial interest in the theoretical analysis and practical applications of these structures, both at the microscopic and macroscopic scales.

Across all the following cited works, two themes emerge:

- Nonlinearity,
- Discreteness.

It is these two properties which seem to allow the emergence of a sort of wave form which is called a localized mode.

The arrays used to investigate ILMs have included:

- Micromechanical coupled oscillator arrays [29, 30, 31, 32, 33, 34],
- Biaxial antiferromagnets [35],
- 2D coupled oscillator arrays [36],
- Coupled arrays of ring oscillators [37],
- Both simulated [29, 30, 36] and experimental [33, 34, 35] investigations.

By far the most common apparatus for the study of ILM has been the micromechanical cantilever array [29, 30, 31, 32, 33, 34]. Topics of study have included pinning and movement, manipulation and control by means such as varying nonlinearity, lifetime of ILMs, and frequency characterization. Many studies use the beam approximation of a cantilever array; [38] describes the use of molecular dynamics to extend the investigation of ILM from the 1D lattices common to the field, to 2D and 3D lattices of oscillators.

One-dimensional oscillator arrays made from a few hundred silicon nitride (SiN_x) crystals are the subject of the experiments in [29, 30]. These crystals are imaged by a CCD camera, using a HeNe laser; higher amplitude of vibration is indicated by darker pixels in an image. The fundamental frequency of the pillars at $f_0 = 147.0$ kHz is within the range which permits use of typical lasers and visible light for imaging. This is in contrast to the vibrational frequencies typical of the arrays under our study later, which are more in the range of tens of megahertz, and necessitate such technologies as the Extreme Ultra Violet (EUV) laser at Colorado State University.

In the microscopic world, [35] details an automatic ILM detection scheme for low temperature microwave driving of biaxial antiferromagnets made from $(C_2H_5NH_3)_2CuCl_4$. The

ILMs were detected using a ‘nonlinear energy magnetometer,’ and were spin waves rather than typical vibrations.

Yet another style of array used to generate ILMs exploits geometry of ring structures to create a nonlinearity from linear elastic springs [37], generating two particular ILM forms, called ‘rotobreathers’ and ’kinks’. One of the proposed advantages of the ring setup is that it would be easily realized in macroscopic experiments.

In general, the above works have concentrated on the analysis of the oscillations being described by a single variable that is perhaps allowed to be complex. In the particular case of experimental investigation of ILMs in cantilever arrays, the one-dimensionality of vibrations was enforced by designing the apparatus with the cantilever thickness in the direction of vibration to be much smaller than in the other direction.

2. Monodirectional System

Let u_n be the deflection of the n-th crystal pillar from its neutral position, which is taken to be $u_n = 0$. Then the Hamiltonian of the system is

$$\mathcal{H}_{\text{mono}} = \mathcal{K}_{\text{mono}} + \mathcal{P}_{\text{mono}} \tag{4}$$

$$\mathcal{K}_{\text{mono}} = \frac{1}{2}\sum_n \dot{u}_n^2 \tag{5}$$

$$\mathcal{P}_{\text{mono}} = \frac{1}{2}\sum_n \big(\alpha_1 u_n^2 + \alpha_2 (u_n - u_{n-1})^2\big) + \frac{1}{4}\sum_n \big(\beta_1 u_n^4 + \beta_2 (u_n - u_{n-1})^4\big) \tag{6}$$

Therefore, the typical equation describing the evolution of one-direction deflection in a one-dimensional array is given by,

$$\begin{aligned} \ddot{u_n} = &- \alpha_1 u_n - \alpha_2\big((u_n - u_{n-1}) + (u_n - u_{n+1})\big) \\ &- \beta_1 u_n^3 - \beta_2\big((u_n - u_{n-1})^3 + (u_n - u_{n+1})^3\big)\,. \end{aligned} \tag{7}$$

Here, $\alpha_1, \alpha_2, \beta_1, \beta_2$ are constants that are determined by the material properties of the pillars, as well as the geometry.

This system has been investigated previously and fairly extensively in the above cited works.

3. Full Bidirectional System

Previous work in ILM analysis has used systems with purely monodirectional oscillation. The topic of this section is the expansion of this previous model, namely equations (4-7), to account for oscillators with *two directions of deflection*. More specifically, we will numerically investigate the appearance and durability of ILMs in oscillator arrays with up to two degrees of vibrational freedom, having nonlinear equations of motion derived from a Hamiltonian with quadratic and quartic terms.

Modern application of ILMs to areas such as sensing require miniaturization of the pillar arrays to sub-micron levels [39], and cantilever arrays of this size are fabricated using techniques like nanoheteropitaxy [40]. This typically produces pillars with similar transversal sizes, which thus do not possess a preferred direction for vibration. Driving these cantilever arrays at their resonant frequencies typically excites vibration modes with deflection in both directions.

The two-dimensional motion of the pillars is similar to having two coupled oscillator arrays, or an array of oscillators with an internal degree of freedom. While there has been considerable amount of work in this area already, see for example [41, 42, 43], the difference between this work and previous studies is in the structure of elastic coupling between two directions of oscillation; the goal here is to analyze the detectability of ILMs in the case when oscillations in two orthogonal directions are allowed.

An illustration of the setup for the problem of interest is shown on Figure 16. For simplicity, we assume that each crystalline pillar in the array has square cross-section, with the flat sides being parallel to the axis of the crystal, and the material is such that crystalline axes are parallel to the pillars' sides and perpendicular to each other. These cantilever arrays, when driven by an external vibration source, function as coupled oscillators. Ultimately, it is the material and design properties that determine the Hamiltonian $\mathcal{H}$ and hence the governing differential equations, which are a generalization of the nonlinear equations for

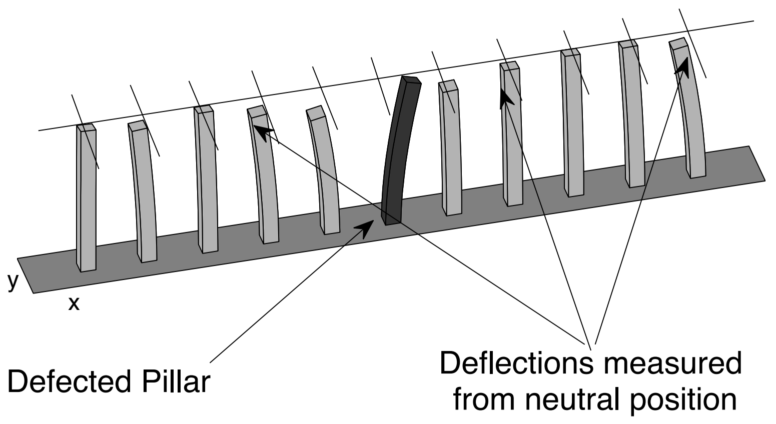


FIGURE 16. An illustration of one dimensional cantilever array that is able to undergo vibrations in two directions. Also drawn is a defected pillar that is used to pin the ILM appearing from random initial condition to that pillar.

pillar vibrations derived earlier. In order to test ILM formation and emulate pinning to molecule attachment to the array, we introduce a small defect in that pillar's properties, and query whether an ILM is attractive to the defect.

3.1. EQUATIONS OF MOTION. The equations of motion for the pillars are formulated via classical mechanics as follows. In order to describe the two-dimensional deflection of the pillars, we introduce the two components of pillar deflection as $\mathbf{u}_n = (u_{n,x}, u_{n,y})$. Since our bidirectional model is a direct expansion of the monodirectional, we write new kinetic and potential energies $\mathcal{K}$, $\mathcal{P}$, which are analogous. The total *potential* energy is,

$$\mathcal{P} = \mathcal{E}_2 + \mathcal{E}_4. \tag{8}$$

Suppose for the given deflection, the linear part of deformation energy is given by the quadratic form $\mathcal{E}_2 = \frac{1}{2}\mathbf{u}_n^T Q_2 \mathbf{u}_n$. We choose a parameterization of the symmetric matrix Q_2 with three parameters $\alpha_{1,x}$, $\alpha_{1,y}$ and α_3 so as to make $\mathcal{E}_2$ be:

$$\mathcal{E}_2 = \frac{1}{2}\sum_n \alpha_{1,x}u_{n,x}^2 + \alpha_{1,y}u_{n,y}^2 + \alpha_2\left((u_{n,x} - u_{n-1,x})^2 + (u_{n,y} - u_{n-1,y})^2\right) + \alpha_3\left(u_{n,x} - u_{n,y}\right)^2. \tag{9}$$

For the purposes of simplified analysis, we only consider $\alpha_{1,x} = \alpha_{1,y} = \alpha_1$. Notice that the eigenvalues of the Hessian of this matrix are strictly positive and distinct for $\alpha_j > 0$, so there are no degeneracies or unphysical values of parameters in the system.

The fourth order term in energy is described by a fourth order tensor; *i.e.*,

$$\mathcal{E}_4 = Q_4 \cdot \mathbf{u}_n \cdot \mathbf{u}_n \cdot \mathbf{u}_n \cdot \mathbf{u}_n \,.$$

Even when proper symmetries are included, the number of non-zero components of Q lead to an exceedingly large number of parameters. It is possible to estimate some of these components analytically if information about the orientation of crystalline axes, pillar shape and nonlinear elasticity is known. This will be done in further studies; for the purpose of this work, we shall consider a simpler particular case of the quartic energy as

$$\mathcal{E}_4 = \frac{1}{4}\sum_n \beta_1 \left(u_{n,x}^4 + u_{n,y}^4\right) + \beta_2 \left(\left(u_{n-1,x} - u_{n,x}\right) + \left(u_{n-1,y} - u_{n,y}\right)\right)^4 + \beta_3 \left(u_{n,x} - u_{n,y}\right)^4 \,. \quad (10)$$

The kinetic energy remains essentially the same as the monodirectional system,

$$\mathcal{K} = \frac{1}{2}\sum_n \dot{\mathbf{u}}_n^2 \,. \quad (11)$$

First, we write the Lagrangian $\mathcal{L}$,

$$\mathcal{L} = \mathcal{K} - \mathcal{P}. \quad (12)$$

Then we determine the generalized momenta, and perform the Legendre Transform.

$$\begin{aligned} \mathbf{p}_n &= \frac{\partial \mathcal{L}}{\partial \dot{\mathbf{u}}_n}\,, \\ &= \sum_n \dot{\mathbf{u}}_n \,. \end{aligned} \quad (13)$$

$$\mathcal{H} = \dot{\mathbf{u}}_n \mathbf{p}_n - \mathcal{L}\,, \tag{14}$$

$$= \mathcal{K} + \mathcal{P}\,. \tag{15}$$

Hamilton's Equations (16) say that:

$$\begin{aligned} \dot{q} &= \frac{\partial \mathcal{H}}{\partial p}, \\ \dot{p} &= -\frac{\partial \mathcal{H}}{\partial q}. \end{aligned} \tag{16}$$

In our particular case, this means,

$$\dot{\mathbf{u}} = \frac{\partial \mathcal{H}}{\partial \dot{\mathbf{u}}}\,, \tag{17}$$

$$\ddot{\mathbf{u}} = -\frac{\partial \mathcal{H}}{\partial \mathbf{u}}\,. \tag{18}$$

Equation (18) is conservative, in that $\mathcal{H}$ is conserved, so the dynamics are constrained to level sets of the Hamiltonian. Indeed, most models of ILM-oscillator mechanics in the literature are conservative, in that total energy is conserved. There is no dissipation, and no driving force. However, because the experimental setup for this study will occur in medium strength vacuum containing the array, and will be actively driven by some yet-undetermined mechanism, we must consider a nonconservative model. In this vein, we supplement two terms to the equations of motion: dissipation and forcing. Dissipation is given by $\gamma\dot{\mathbf{u}}$, where γ is the coefficient of dissipation; forcing is given by $\sigma(t)$, and is represented in present simulations by periodic sinusoidal forcing. Hence, we have for our equations of motion,

$$\ddot{\mathbf{u}}_n = -\frac{\partial \mathcal{H}}{\partial \mathbf{u}_n} - \gamma\dot{\mathbf{u}}_n + \sigma(t)\,. \tag{19}$$

Note that the dissipative model can recover the conservative model, simply by setting $\gamma = 0$, $\sigma = 0$. The corresponding equations of motion for two directions are

$$\begin{aligned}
\ddot{u}_{n,x} = & -\alpha_{1,x} u_{n,x} - \alpha_2\big((u_{n,x} - u_{n-1,x}) + (u_{n,x} - u_{n+1,x})\big) \\
& - \beta_1 u_{n,x}^3 - \beta_2\big((u_{n,x} - u_{n-1,x})^3 + (u_{n,x} - u_{n+1,x})^3\big) \\
& - \alpha_3(u_{n,x} - u_{n,y}) - \beta_3(u_{n,x} - u_{n,y})^3 \\
& - \gamma \dot{u}_{n,x} + \sigma_x(t)\,, \qquad (20) \\
\ddot{u}_{n,y} = & -\alpha_{1,y} u_{n,y} - \alpha_2\big((u_{n,y} - u_{n-1,y}) + (u_{n,y} - u_{n+1,y})\big) \\
& - \beta_1 u_{n,y}^3 - \beta_2\big((u_{n,y} - u_{n-1,y})^3 + (u_{n,y} - u_{n+1,y})^3\big) \\
& - \alpha_3(u_{n,y} - u_{n,x}) - \beta_3(u_{n,y} - u_{n,x})^3 \\
& - \gamma \dot{u}_{n,y} + \sigma_y(t)\,. \qquad (21)
\end{aligned}$$

This functional form of directional coupling allows a comprehensive study of ILM formation for any set of parameters. In addition, we have added the dissipation in the pillars, described by the term $\gamma\dot{\mathbf{u}}_n$, and the forcing term σ. Table 1 summarizes the meaning of each of the parameters appearing in the equations of motion.

For symmetry reasons, in the case considered here there is no cubic term in the energy. However, note that a cubic term may appear for particular arrangements of the crystal axes and pillar facets, in the geometries breaking the reflection symmetry of the system. The appearance and role of a cubic term in energy, leading to quadratic terms in the equations, is very interesting. However, this will lead to the investigation of a very large number of parameters, and will have to be addressed in further studies.

Table 1. Parameter Interpretation for Bidirectional Model

Parameter	Interpretation (Equations of Motion)
α_1	Linear restoration force
α_2	Linear nearest-neighbor coupling
α_3	Linear directional coupling
β_1	Cubic restoration force
β_2	Cubic nearest-neighbor coupling
β_3	Cubic directional coupling
γ	Damping Coefficient
σ	Driving function

3.2. Simulation. The overarching goal is to understand bidirectional vibrations in arrays of coupled oscillators. There are several prominent reasons for this. Foremost, the arrays are manufactured using techniques that will necessarily yield pillars with very similar side widths. This in contrast to the macromechanical and micromechanical arrays mentioned above [29, 30, 31, 32, 33, 34], where the monodirectional nature of oscillations is enforced by making one width of pillars significantly larger than the other. Secondly, the visualization using EUV requires out-of-plane oscillation; if the vibration was monodirectional the light field would be undisturbed. We use (20,21) to realistically simulate experimental conditions.

In order that the manufactured arrays display favorable characteristics for ILM detectability in the EUV holography system, we need the ILMs to form reliably from random initial conditions with distributed forcing. Not only this, but we need the ILM to be of high enough amplitude so as to stand out against a background of oscillating pillars; that is, it needs to be *detectable.* Since there are several candidates for realizing a forcing method, namely the noisy piezo method and the materials-problematic electronic forcing, we need to be able to simulate a wide variety of parameters. Previous studies have used conservative models to describe ILMs. Typically, artificially prepared initial conditions propagate across the array. In our case, dissipation will greatly affect the dynamics of the system, and requires the array to be driven. Hence, interaction between the terms $\gamma\dot{u}$ and σ will be important for prediction of the behavior of the experimental setup.

Table 2. Simulation Parameter Values

Parameter	Value
α_1	$0.0001 \ldots 1$
α_3	$0.0001 \ldots 1$
β_1	0.01
β_2	0.0001
β_3	0, 0.001
γ	0.001
$\lvert\sigma\rvert$	$0.01 \ldots 0.2$

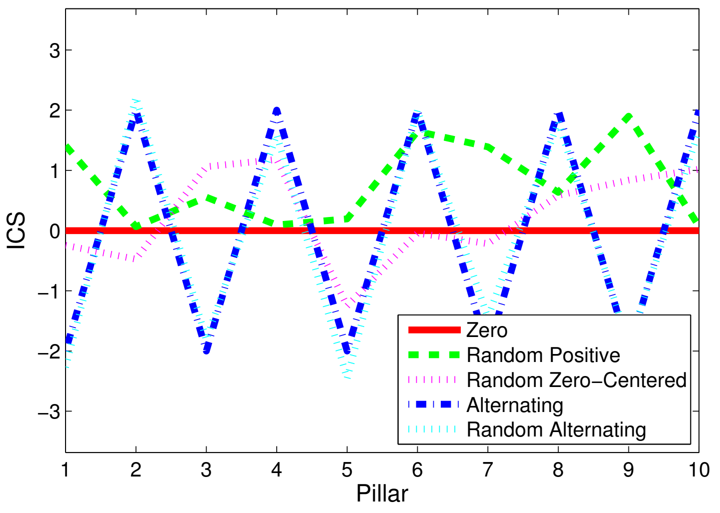


FIGURE 17. The various initial conditions available for the bidirectional full system. Magenta, green and cyan lines have random components.

3.2.1. *Initial Conditions.* Of the myriad initial conditions one can dream of, a wide variety of initial conditions are implemented in the simulation code, and are illustrated in Figure 17. In an experiment with sub-micron nanopillar arrays, the forcing will most likely be distributed among many pillars and artificial preparation of an ILM will be experimentally unfeasible. Thus, in our simulations, we have mostly analyzed ILMs that spontaneously and robustly appear from random initial conditions in deflection, with $\dot{u}_n = 0$, under the influence of uniform distributed forcing. In addition to the random positive and zero-average initial conditions, one can start from a zero-velocity, zero-deflection state. There are also two *alternating* initial conditions available, so that each pillar is deflected with opposite sign from both its neighbors, one uniformly, and another randomly perturbed from the uniform state. In each of the nonzero initial conditions described, the initial amplitude is a parameter.

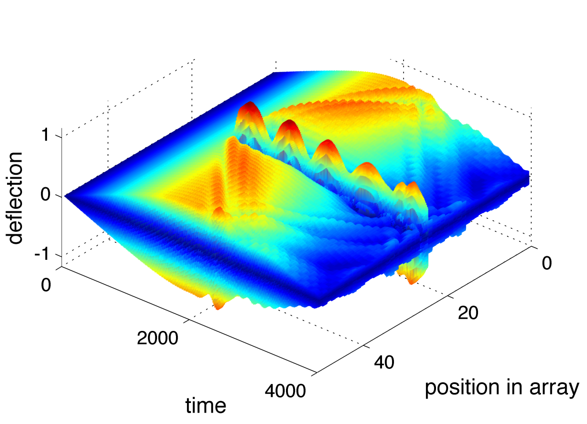
Zero. Running a simulation from uniformly zero initial condition permits investigation of an array at the beginning of an experiment, as driving is initiated. This corresponds the red line in Figure 17, and the two simulations in Figure 18(a),(b). In (a), building energy after vibration starts, the introduction of the 5% defect at time $t = 1000$ induces immediate ILM formation, with high detectability. In (b) even a 0.1% defect causes energy concentration, although small. Low frequency oscillations occur with persistent low-amplitude ILM.

Random Positive. The condition where the array is displaced entirely to one side or the other is conceivable. Therefore, such an initial condition is an option, as the green line in Figure 17, and the simulation in Figure 18(c). Even in the face of random initial condition, with the presence of dissipation and driving, ILM forms as soon as defect appears in array.

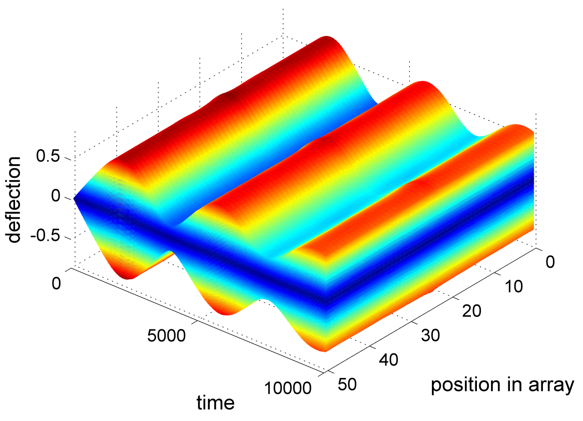
Random Zero-Centered. Perhaps more typical may be the situation where the array is randomly displaced, but averages to having zero displacement; the magenta line in Figure 17 represents this case, with a simulation appearing in Figure 18(d). Strong ILM formation immediately upon defect introduction.

Alternating. The alternating initial condition appears as the blue line in Figure 17.

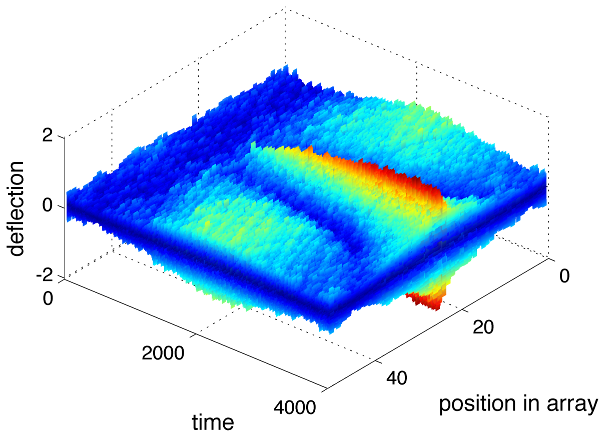
Random Alternating. Finally, we consider the case of each pillar's deflection having opposite sign, but being randomly perturbed. The cyan initial condition in Figure 17 displays a 10% variation in amplitude from the pure alternating case. Such simulations, without defect, appear in Figure 18(e),(f). Neither random alternating simulation here displays an ILM, and for good reason – there is no defect in these simulations. Starting from alternating random initial conditions produce interesting wave forms, but localization of energy is not strong, and does not pin to any one specific pillar.



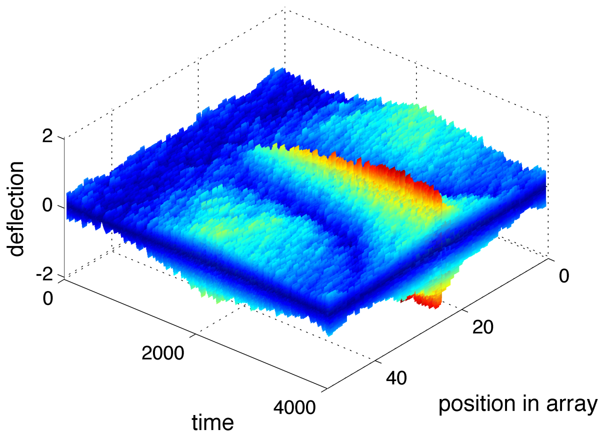
(A) Zero, 5% defect



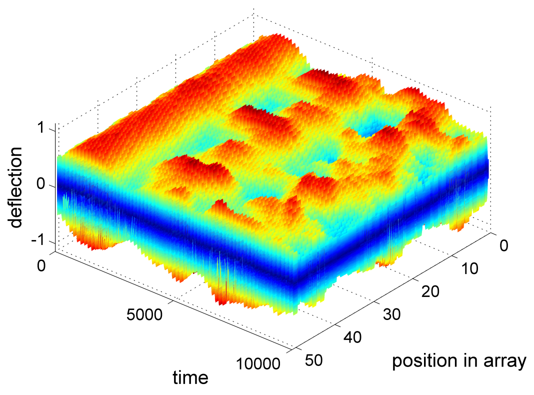
(B) Zero, 0.1% defect



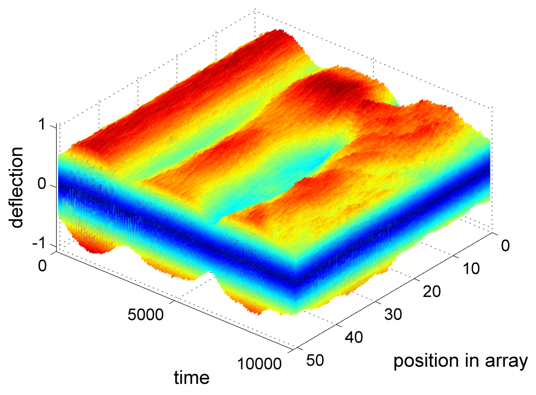
(C) Random Positive, 5% defect



(D) Random Zero-Centered, 5% defect



(E) Random Alternating, 0% defect



(F) Random Alternating, 0% defect

FIGURE 18. Array of simulations coming from a variety of initial conditions.

3.2.2. *Detectability.* Here, we present the results for spontaneous formation of ILMs in the driven dissipative system. For completeness, we include values of parameters used in simulations in Table 2. While the typical value of α_1 is around 1, we have chosen to scan a large set of values for other parameters in order to accommodate both cases when there is large discrepancy in natural frequencies in two directions, and, alternatively, when the natural frequencies of vibration are very close to each other.

We define detectability in terms of concentration of energy. Using the energies $\mathcal{K}$, $\mathcal{P}$ defined above, we can define the energy of a single pillar:

$$E_n = \sum_{x,y} \frac{1}{2}\dot{u}_n^2 + \frac{1}{2}\left(\alpha_1 u_n^2 \quad + \alpha_2 (u_n - u_{n-1})^2 - \alpha_3 (u_{n,x} - u_{n,y})^2\right) \tag{22}$$

$$+ \frac{1}{4}\left(\beta_1 u_n^4 + \beta_4 (u_n - u_{n-1})^4 + \beta_3 (u_{n,x} - u_{n,y})^4\right) \tag{23}$$

The relative energy concentrated at the n^{th} pillar at any particular moment, then is

$$\frac{(N-1)E_n}{\sum_{j\neq n} E_j} \tag{24}$$

which we further refine into $\mathbb{D}$, the detectability for a time period from t_0 to t_1:

$$\mathbb{D} = \frac{1}{t_1 - t_0}\int_{t_0}^{t_1} \frac{(N-1)E_n}{\sum_{j\neq n} E_j}\, dt \tag{25}$$

Detectability value unity indicates that the energy is uniformly distributed throughout the array over the time period. Values $\mathbb{D} > 1$ indicate ILM formation, and sustained energy concentration, while $\mathbb{D} < 1$ indicates energy repulsion. Appropriate time scales for t_0, t_1 are not immediately apparent.

Discretizing and repeatedly simulating the full system, and computing $\mathbb{D}$ produces the parameter scans in Figure 19. We plot the ILM detectability, *i.e.*, the energy concentrated in a defected pillar divided by the average energy of all other pillars, which is non-zero due to the competition between the forcing and dissipation. Each point on this plot is the result

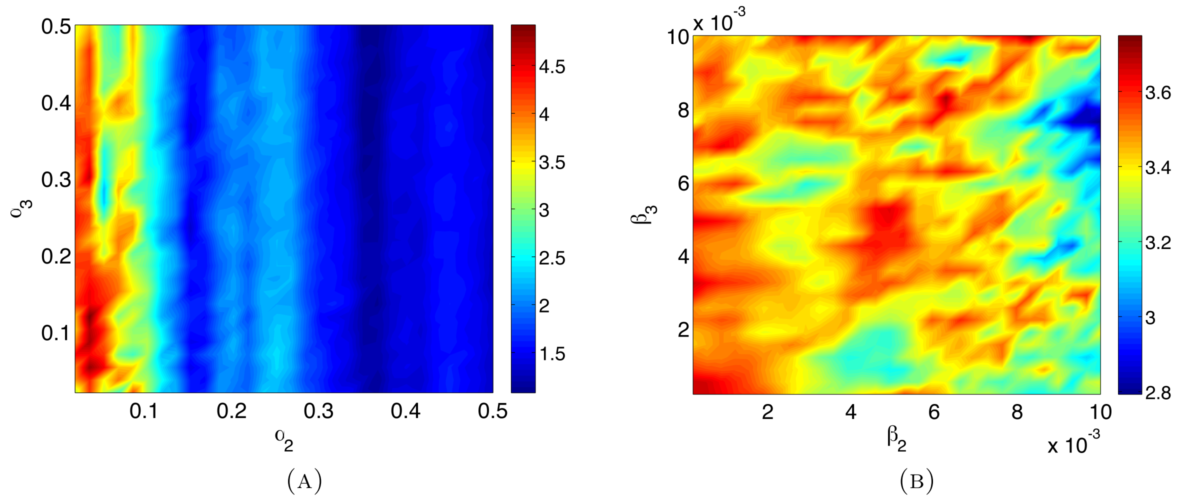


FIGURE 19. Phase diagram for ILM detectability at defected pillar in 50-pillar crystal array. Coupling parameters α_2 and α_3 are examined in the left, β_2 and β_3 in the right. Red areas denotes strong ILM formation whereas blue indicates practically no spontaneously forming ILMs present in the system.

of simulation until $t = 2000$, with the amplitude of ILM computed from the last 100 time units of the interval.

Scanning through α_2 and α_3 produces Figure 19(a). The horizontal and vertical axes are α_2 and α_3, respectively. Red areas represent high values of detectability, and blue represent the low values. As we see, there are areas of parameter where ILM appearance is very robust; however, these areas are relatively small, and relegated to $\alpha_2 \approx 0.1$. The parameter coupling the directions of motion has small affect on detectability and pinning. This band of detectability agrees reasonably well previous studies, which used parameter values $\alpha_1 = 1$, $\alpha_2 = 0.1$.

Figure 19(b) is a detectability parameter scan, for β_2 and β_3. In contrast to the α scan, regions of enhanced ILM pinning are mixed throughout parameter space. There is not a single set of β_i which one would be well advised to choose in design. The lower limit presented in this Figure is $\mathbb{D} \approx 2.3$, indicating that even for poor choices of β_i, one is likely to see ILMs.

The major lesson to take away from Figure 19 is that, with α_1 fixed, the inter-pillar coupling parameter α_2 has the strongest affect on ILM formation and pinning. However, choosing a material with suitable dissipation γ, and the correct driving mechanism, will still likely be a challenge. Thus, the simulation work presented here warrants more detailed studies of ILM formation and detectability, particularly the realization of physical nanosystems, for correlation with experiment.

3.2.3. *Discussion.* We have been able to answer the two following related questions: Does the positive or negative defect value ϵ cause attraction or repulsion? Will the introduction of a defect to the linear term $(\alpha_1 + \epsilon)$ in the equations of motion cause the ILM to repel or attract?

If $\epsilon > 0$, the ILM will form on the defected location, in an attractive fashion. Energy focuses on the defect. Simulations indicate that concentration factors of up to $10\times$ or even more are achievable, if parameters in the system are suitably chosen. More typical is $\mathbb{D} \approx 3$, but this should be easy to detect using the holography setup.

If $\epsilon < 0$ energy diminishes at the defected pillar. Comprehensive simulations determining $\mathbb{D}$ have not been performed, but there is anecdotal evidence that a -5% defect strongly deflects energy. There probably will still be a useful signature against which one could detect molecule attachment, but the vibrational energy is *removed* from the attachment location.

There are many unanswered questions related to this system. For example, it is clear from simulations that sometimes and ILM will travel the array. What is the rate of this travel, and on which parameters does it depend? What are the typical lifetimes of ILMs in the bidirectional system?

The use of an array as a sensor will almost certainly cause multiple simultaneous attachments. What is the affect of a second or third attachment when an ILM is already established on the array? When a molecule detaches, what happens?

These are all important questions, and merit further research.

4. Simplified Model for Monodirectional ILM Travel

In order to understand ILM dynamics, we derive and simulate a reduced system. Preliminarily, we assume:

- Infinite array of oscillators,
- Monodirectional vibration,
- Conservative (no dissipation, no driving).

We choose a particular form for the ILM on the array as a symmetric exponential. Notation is consistent with that found above, so that u_n represents deflection from neutral at pillar number n. We work from the energy, with potential $\mathcal{P}$ and kinetic $\mathcal{K}$:

$$\mathcal{P} = \sum_{n=-\infty}^{\infty} \frac{1}{2}(\alpha_1 u_n^2 + \alpha_2(u_n - u_{n-1})^2) + \frac{1}{4}(\beta_1 u_n^4 + \beta_2(u_n - u_{n-1})^4)\,, \tag{26}$$

$$\mathcal{K} = \frac{1}{2}\sum_{n=-\infty}^{\infty} \dot{u}_n^2\,. \tag{27}$$

Previously, research has derived equations of motion from these energies, and simulated with a large number of pillars. Here, we turn this high-dimension model into a system with only *two* or *four* equations.

The form of the ILM is taken to be alternating, and centered at the point x, which is not necessarily an integer, and has peak amplitude A; let n_0 denote the ceiling of x. The ILM has temporal period $2\pi/\omega$, over which we will eventually average. The parameter λ controls the sharpness of the ILM.

$$u_n = A\exp(-\lambda|x(t) - n|)(-1)^n \sin(\omega t)\,. \tag{28}$$

We will proceed by substituting ILM form (28) into $\mathcal{P}$, $\mathcal{K}$, and averaging, rearranging, and collecting.

4.1. Two Variable Model. In this section, A is taken to be a constant, and only the position x is allowed to vary with time. In Section 4.2, we will allow both $x(t)$ and $A(t)$.

4.1.1. *Kinetic Energy.*

$$\dot{u}_n = (-1)^n A \exp(-\lambda|n-x|)\,(-\omega\sin(\omega t) + \lambda\cos(\omega t)\,\mathrm{sign}(n-x)\,\dot{x}) \tag{29}$$

Substitution of (29) into (27) yields,

$$\begin{aligned}
\mathcal{K} &= \frac{1}{2}\sum_{n=-\infty}^{\infty} \dot{u}_n^2\,, \\
&= \frac{A^2}{2}\sum_{-\infty}^{n_0-1} \exp(-2\lambda(x-n))(-\omega\sin\omega t + \lambda\cos(\omega t)(-1)\,\dot{x})^2 \\
&\qquad + \frac{A^2}{2}\sum_{n_0}^{\infty} \exp(-2\lambda(n-x))(-\omega\sin\omega t + \lambda\cos(\omega t)\,\dot{x})^2\,.
\end{aligned}$$

Recognizing the geometric sum,

$$\sum_{-\infty}^{m} e^{d\lambda n} = \frac{e^{d\lambda m}}{1-e^{-d\lambda}}\,, \tag{30}$$

we then get,

$$\begin{aligned}
\mathcal{K} = \frac{A^2}{2}&\left(\omega^2\sin^2\omega t + 2\omega\lambda\sin\omega t\cos\omega t + \lambda^2\cos^2\omega t\,\dot{x}^2\right)\frac{\exp(-2\lambda(x-n_0+1))}{1-\exp(-2\lambda)} \\
&+ \frac{A^2}{2}\left(\omega^2\sin^2\omega t - 2\omega\lambda\sin\omega t\cos\omega t + \lambda^2\cos^2\omega t\,\dot{x}^2\right)\frac{\exp(-2\lambda(n_0-x))}{1-\exp(-2\lambda)}\,.
\end{aligned}$$

The expressions involving each of $\sin^2$, $\cos^2$, and $\sin \cdot \cos$ are averaged over one period of oscillation,

$$\begin{aligned} \sin^2 \omega t &\rightarrow \frac{1}{2}\,, \\ \cos^2 \omega t &\rightarrow \frac{1}{2}\,, \\ \sin \omega t \cos \omega t &\rightarrow 0\,, \end{aligned}$$

so that we get,

$$\begin{aligned} \mathcal{K} &= \frac{A^2}{2} \frac{\omega + \frac{\lambda^2 \pi}{\omega} \dot{x}^2}{1 - e^{-2\lambda}} \left(e^{-2\lambda(x - n_0 + 1)} + e^{-2\lambda(n_0 - x)} \right)\,, && (31) \\ &= \dot{x}^2 \, \frac{\psi(x)}{2}\,. && (32) \end{aligned}$$

4.1.2. *Potential Energy.* We derive the equations for the potential energy in similar fashion. First we will substitute the ILM form into (28), and then we will simplify the equations.

$$\mathcal{P} = \frac{1}{2} \left(\alpha_1 \mathcal{U}_1 + \alpha_2 \mathcal{U}_2 \right) + \frac{1}{4} \left(\beta_1 \mathcal{U}_3 + \beta_2\,, \mathcal{U}_4 \right)\,, \qquad (33)$$

$$\begin{aligned} \mathcal{U}_1 &= \sum_{-\infty}^{\infty} u_n^2\,, \\ \mathcal{U}_2 &= \sum_{-\infty}^{\infty} (u_n - u_{n-1})^2\,, \\ \mathcal{U}_3 &= \sum_{-\infty}^{\infty} u_n^4\,, && (34) \\ \mathcal{U}_4 &= \sum_{-\infty}^{\infty} (u_n - u_{n-1})^4\,. \end{aligned}$$

Again, geometric sums as (30) may be simplified, and temporal oscillations averaged.

$$\begin{aligned}\mathcal{U}_1 &= \sum_{-\infty}^{\infty} u_n^2\,, \\ &= A^2\cos^2\omega t\sum_{-\infty}^{\infty}\exp(-2\lambda|n-x|)\,, \\ &= \frac{A^2}{2(1-e^{-2\lambda})}\left(e^{-2\lambda(x-n_0+1)}+e^{-2\lambda(n_0-x)}\right)\;. \qquad (35)\end{aligned}$$

$$\begin{aligned}\mathcal{U}_2 &= \sum_{-\infty}^{\infty}(u_n-u_{n-1})^2\,, \\ &= A^2\cos^2\omega t\sum_{-\infty}^{\infty}\left(\exp(-2\lambda|n-x|)-\exp(-2\lambda|n-1-x|)\right)^2\;, \qquad (36) \\ &= 2A^2\cos^2(\omega t)\left(\frac{e^{-2\lambda(x-n_0+1)}+e^{-2\lambda(n_0-x)}}{1-e^{-2\lambda}}+\frac{e^{-2\lambda(x-n_0+1)}e^{-\lambda}+e^{-2\lambda(n_0-x)}e^{-\lambda}}{1-e^{-2\lambda}}+e^{-\lambda}\right)\;, \\ &= A^2\left(\frac{1+e^{-\lambda}}{1-e^{-2\lambda}}\left(e^{-2\lambda(x-n_0+1)}+e^{-2\lambda(n_0-x)}\right)+e^{-\lambda}\right)\;. \qquad (37)\end{aligned}$$

The equations for $\mathcal{U}_3$ are directly analogous to those for $\mathcal{U}_1$:

$$\begin{aligned}\mathcal{U}_3 &= \sum_{-\infty}^{\infty} u_n^4\,, \qquad (38) \\ &= \frac{A^4\cos^4(\omega t)}{1-e^{-4\lambda}}\left(e^{-4\lambda(x-n_0+1)}+e^{-4\lambda(n_0-x)}\right)\;, \qquad (39) \\ &= \frac{3A^4}{8(1-e^{-4\lambda})}\left(e^{-4\lambda(x-n_0+1)}+e^{-4\lambda(n_0-x)}\right)\;. \qquad (40)\end{aligned}$$

Finally, the expansion of $\mathcal{U}_4$ is similar to $\mathcal{U}_2$, except that there are a few extra multiplicative factors and additive terms.

$$\begin{aligned}\mathcal{U}_4 &= \sum_{-\infty}^{\infty}(u_n - u_{n-1})^4\,, \qquad (41)\\
&= A^4\cos^4(\omega t)\Big(2\frac{e^{-4\lambda(x-n_0+1)}+e^{-4\lambda(n_0-x)}}{1-e^{-4\lambda}} + \left(4e^{-\lambda}+6e^{-2\lambda}+4e^{-3\lambda}\right)\frac{e^{-4\lambda(x-n_0+1)}}{1-e^{-4\lambda}}\\
&\quad + \left(4e^{\lambda}+6e^{2\lambda}+4e^{3\lambda}\right)\frac{e^{-4\lambda(n_0-x+1)}}{1-e^{-4\lambda}} + 4e^{-2\lambda(n_0-x)}e^{-\lambda} + 6e^{-2\lambda} + 4e^{-2\lambda(x-n_0+1)}e^{-\lambda}\Big)\,,\\
&= \frac{3A^4}{8}\Big(\frac{2+4e^{-\lambda}+6e^{-2\lambda}+4e^{-3\lambda}}{1-e^{-4\lambda}}\left(\exp(-4\lambda(x-n_0+1))+\exp(-4\lambda(n_0-x))\right) \qquad (42)\\
&\quad + 4e^{-\lambda}\left(\exp(-2\lambda(x-n_0+1))+\exp(-2\lambda(n_0-x))+\frac{3}{2}e^{-\lambda}\right)\Big)\,.\end{aligned}$$

Equations for $\mathcal{U}_j$ can be significantly simplified by making the following substitutions:

$$\begin{aligned}E_2 &= e^{-2\lambda(n_0-x)} + e^{-2\lambda(x-n_0+1)}\,,\\
E_4 &= e^{-4\lambda(n_0-x)} + e^{-4\lambda(x-n_0+1)}\,,\\
\tilde{E}_2 &= e^{-2\lambda(n_0-x)} - e^{-2\lambda(x-n_0+1)}\,, \qquad (43)\\
\tilde{E}_4 &= e^{-4\lambda(n_0-x)} - e^{-4\lambda(x-n_0+1)}\,,\\
\xi_2 &= 1-e^{-2\lambda}\,,\\
\xi_4 &= 1-e^{-4\lambda}.\end{aligned}$$

And as a result of the substitution,

$$\mathcal{U}_1 = \frac{A^2}{2}\frac{E_2}{\xi_2}\,, \tag{44}$$

$$\mathcal{U}_2 = A^2\left((1+e^{-\lambda})\frac{E_2}{\xi_2} + e^{-\lambda}\right)\,, \tag{45}$$

$$\mathcal{U}_3 = \frac{3A^4}{8}\frac{E_4}{\xi_4}\,, \tag{46}$$

$$\mathcal{U}_4 = \frac{3A^4}{8}\left((2+4e^{-\lambda}+6e^{-2\lambda}+4e^{-3\lambda})\frac{E_4}{\xi_4} + 4e^{-\lambda}E_2 + 6e^{-2\lambda}\right)\,. \tag{47}$$

Finally, we combine (44-47) into (33)

$$\mathcal{P} = A^2\left(\frac{E_2}{2\xi_2} + (1+e^{-1})\frac{E_2}{\xi_2} + e^{-1}\right) \tag{48}$$

$$+ A^4\frac{3}{8}\left(\frac{E_4}{\xi_4} + (2+4e^{-\lambda}+6e^{-2\lambda}+4e^{-3\lambda})\frac{E_4}{\xi_4} + 4e^{-\lambda}E_2 + 6e^{-2\lambda}\right)\,. \tag{49}$$

4.1.3. *Defect.* A defect in the array of pillars, introduced to mimic molecule attachment to a nano scale crystal array, can be introduced into this system easily. For demonstration, we defect parameter α_1. Suppose the defect occurs on pillar k_0, so that *at pillar* k_0, $\tilde{\alpha}_1 = \alpha_1 + \alpha_D$. Then we simply have an additive term to $\mathcal{P}$, because the equation manifests from a sum of positive terms. Therefore, in the presence of a defect,

$$D = \frac{A^2}{2}\alpha_D e^{-2\lambda|x-k_0|}\,, \tag{50}$$

and we can redefine,

$$\mathcal{P} = \mathcal{P} + D\,. \tag{51}$$

4.1.4. *Hamiltonian Mechanics.* We finally get to the meat of the discussion, as we write the Lagrangian $\mathcal{L}$ for the simplified two dimensional system:

$$\mathcal{L} = \mathcal{K} - \mathcal{P}\,, \tag{52}$$

$$= \dot{x}^2\,\frac{\psi(x)}{2} - \mathcal{P}\,. \tag{53}$$

Then we define the canonical variable, or generalized momentum P_x, as

$$\begin{aligned} P_x &= \frac{\partial \mathcal{L}}{\partial \dot{x}}, \\ &= \psi(x)\dot{x}\,, \\ \text{inverting} \qquad &\Rightarrow\ \dot{x} = P_x/\psi\,. \end{aligned}$$

Going through the Legendre transform, we get the Hamiltonian $\mathcal{H}$:

$$\mathcal{H} = \dot{x}\,P_x - \mathcal{L}\,,$$

$$\mathcal{H} = \frac{1}{2}P_x^2/\psi + \mathcal{P}\,. \tag{54}$$

The equations of motion for the two-variable system are then,

$$\begin{aligned} \dot{x} &= \frac{\partial \mathcal{H}}{\partial P_x}\,, \\ \dot{P}_x &= -\frac{\partial \mathcal{H}}{\partial x}\,. \end{aligned}$$

To examine the dynamics of this system, rather than running ODE simulations, we look at the energy landscape. Indeed, $\mathcal{H}$ is conserved in time, as $\frac{d}{dt}(\mathcal{H}) = 0$. Hence, the level sets of $\mathcal{H}$ indicate paths in $(x,\, P_x)$-space through time. An example of such level sets is given in Figure 20. There are periodic orbits appearing in the bottom of the energy wells, seen in dark blue, and escaping trajectories further up the walls, seen in yellow and red. These are

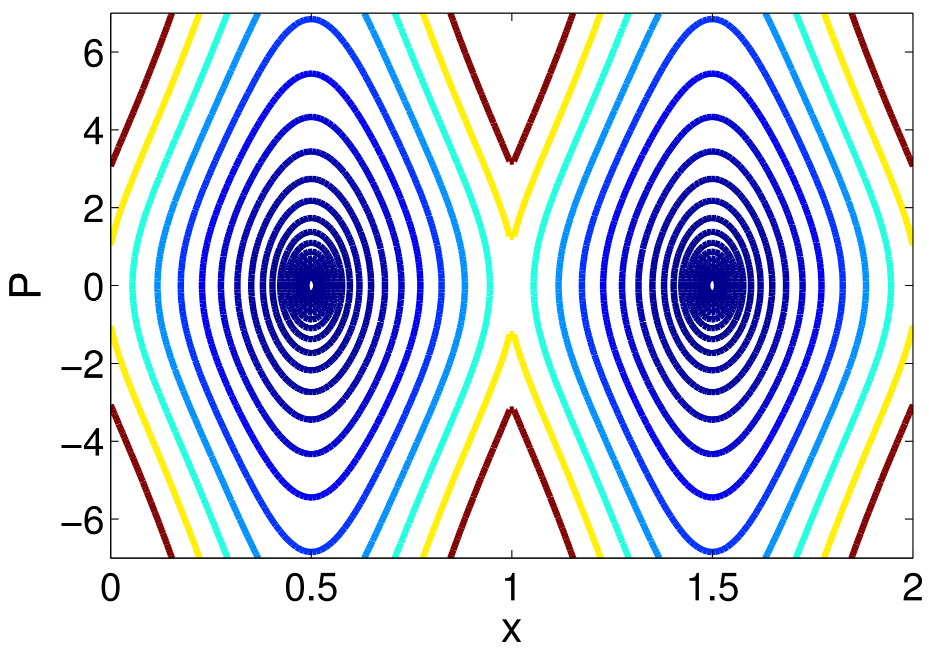


FIGURE 20. Level sets, for the two variable system, of $\mathcal{H}$ (54), which is conserved. Trajectories are either oscillatory about a local minimum, or escaping. Defects lower or raise the boundary between minimal energy states, strengthening pinning, or allowing travel between multiple pillars.

the only two types of trajectories appearing as level sets, so the dynamics of this system are rather dull.

The inclusion of defect in the system serves to lower or raise the boundary between local minima in the energy landscape. If the defect lowers the parameter, it will permit movement of the ILM between pillars, but the net effect will still be simple style oscillation within its energy well. Increasing parameter values by defect reduces movement between pillars, containing ILM of higher energy at the defect location.

Physical systems have demonstrated complex dynamics of ILM, which this two variable reduced system fails to capture. We improve our model in the next section by introducing another time dependence.

4.2. Four Dimensional Model. The four-dimensional model accounts for change in both the amplitude A of the ILM , as well as its peak position x. Whereas the two variable model was too simple to allow complicated dynamics (as could have been predicted from a dynamical systems point of view, since 2D real systems are never chaotic), our 4D system will be much more realistic.

4.2.1. *Kinetic Energy.* Still working with (28) for the ILM form, but with $x = x(t)$, $A = A(t)$, we have a new expression for the velocity of each pillar:

$$\begin{aligned} \dot{u}_n &= \frac{\partial u_n}{\partial A}\dot{A} + \frac{\partial u_n}{\partial x}\dot{x}\,, \\ &= \left(\dot{A}\cos\omega t - A\omega\sin\omega t - \lambda A\,\mathrm{sign}(x-n)\dot{x}\cos\omega t\right)\,, \qquad (55) \\ \Rightarrow\ \dot{u}_n^2 &= \frac{e^{-2\lambda|x-n|}}{2}\left(\omega^2A^2 + \dot{A}^2 + A^2\lambda^2\dot{x}^2 - 2A\lambda\mathrm{sign}(x-n)\dot{A}\dot{x}\right)\,, \qquad (56) \end{aligned}$$

after averaging the time functions. Substituting (56) into $\mathcal{K}$ and averaging over the period of the oscillation, we get

$$\begin{aligned} \mathcal{K} &= \frac{1}{2}\sum_{n=-\infty}^{\infty}\dot{u}_n^2\,, \\ &= \frac{E_2}{4\xi_2}\left(\omega^2A^2 + \dot{A}^2 + A^2\lambda^2\dot{x}^2\right) - A\lambda\dot{A}\dot{x}\frac{1}{2}\sum_n e^{-2\lambda|x-n|}\mathrm{sign}(x-n)\,. \qquad (57) \end{aligned}$$

We turn our attention to the the sum in (57). As long as x is not an integer, this may be written as

$$-\sum_n e^{-2\lambda|x-n|}\mathrm{sign}(x-n) = \frac{1}{\xi_2}\left(e^{-2\lambda(n_0-x+1)} - e^{-2\lambda(x-n_0+1)}\right) + e^{-2\lambda(x-n_0)}\mathrm{sign}(x-n_0)\,, \qquad (58)$$

$$= \mathbb{E}\,. \qquad (59)$$

$$x \neq n_0 \quad \Rightarrow \quad \mathbb{E} = \frac{\tilde{E}_2}{\xi_2} \tag{60}$$

$$x == n_0 \quad \Rightarrow \quad \mathbb{E} = 0 \tag{61}$$

$\mathbb{E}$ has a discontinuity at $x = n_0$, when the ILM is centered *at* a pillar; this occurs at a set of measure 0, so we disregard it, and hereafter assume $\mathbb{E} = E_2/\xi_2$.

Therefore, we write the final expression for our kinetic energy:

$$\mathcal{K} = \frac{1}{4}\left(\dot{A}^2 + A^2\omega^2 + \lambda^2 A^2 \dot{x}^2\right)\frac{E_2}{\xi_2} + \frac{1}{2}A\dot{A}\lambda\dot{x}\mathbb{E}\,, \tag{62}$$

$$= \frac{1}{4}\frac{E_2}{\xi_2}\dot{A}^2 + \frac{\lambda^2 A^2}{4}\frac{E_2}{\xi_2}\dot{x}^2 + \frac{A\lambda}{2}\mathbb{E}\dot{A}\dot{x} + \frac{A^2\omega^2}{4}\frac{E_2}{\xi_2}\,. \tag{63}$$

4.2.2. *Potential Energy.* The potential energy $\mathcal{P}$ for the four dimensional reduced system is the same as $\mathcal{P}$ for the two variable system, as in (33, 48).

4.2.3. *Hamiltonian Mechanics.* We first write the Langrangian

$$\mathcal{L} = \mathcal{K} - \mathcal{P}\,, \tag{64}$$

$$\mathcal{L} = \frac{F_1}{2}\dot{A}^2 + A^2\frac{F_2}{2}\dot{x}^2 + AF_3\dot{A}\dot{x} - c_2A^2 - c_4A^4\,. \tag{65}$$

where

$$F_1 = \frac{E_2}{2\xi_2}\,, \tag{66}$$

$$F_2 = \frac{\lambda^2 E_2}{2\xi_2}\,, \tag{67}$$

$$F_3 = \frac{\lambda\mathbb{E}}{2}\,. \tag{68}$$

$$c_2 = \left(-\frac{\omega^2}{4\xi_2} + \frac{1}{4\xi_2}(\alpha_1 + \alpha_2(2 + 2e^{-\lambda}))\right)E_2 + \frac{\alpha_2}{2}e^{-\lambda} + \frac{\alpha_D}{2}e^{-2\lambda|x-K|}\,, \tag{69}$$

$$c_4 = \frac{3}{8}\beta_2 e^{-\lambda}E_2 + \frac{3}{32}\left(\beta_1 + \beta_2(2 + 4e^{-\lambda} + 6e^{-2\lambda} + 4e^{-3\lambda})\right)\frac{E_4}{\xi_4} + \frac{9}{16}\beta_2 e^{-2\lambda}\,. \tag{70}$$

Hamilton's Equations allow us canonical variables,

$$\begin{aligned} P_A &= \frac{\partial \mathcal{L}}{\partial \dot{A}}\,, \\ &= F_1\dot{A} + AF_3\dot{x}\,. \end{aligned} \tag{71}$$

$$\begin{aligned} P_x &= \frac{\partial \mathcal{L}}{\partial \dot{x}}\,, \\ &= A^2F_2\dot{x} + AF_3\dot{A}\,. \end{aligned} \tag{72}$$

Inverting these, and letting $\Delta = F_1F_2 - F_3^2$,

$$\dot{A} = \frac{1}{\Delta}(F_2P_A - A^{-1}F_3P_x)\,, \tag{73}$$

$$\dot{x} = \frac{1}{\Delta}(-A^{-1}F_3P_A + A^{-2}F_1P_x)\,. \tag{74}$$

$$\Delta = F_1F_2 - F_3^2 \tag{75}$$

$$= \frac{\lambda^2}{4\xi_2^2}(E_2^2 - \tilde{E}_2^2)\,, \tag{76}$$

$$= \frac{\lambda^2}{\xi_2^2}e^{-2\lambda}\,. \tag{77}$$

Using the Legendre Transform,

$$\mathcal{H} = < p\,, \dot{q} > - \mathcal{L}(q, \dot{q}), \tag{78}$$

we can move fully into the canonical system, and $\mathcal{H}$ becomes,

$$\mathcal{H} = \frac{1}{\Delta}\left(\frac{F_2}{2}P_A^2 + A^{-2}\frac{F_1}{2}P_x^2 - A^{-1}F_3P_AP_x\right) + c_2A^2 + c_4A^4. \tag{79}$$

Armed with $\mathcal{H}$, we can derive the equations of motion for the canonically transformed system.

$$\dot{P}_x = -\frac{\partial \mathcal{H}}{\partial x}\,, \tag{80}$$

$$\dot{P}_A = -\frac{\partial \mathcal{H}}{\partial A}\,, \tag{81}$$

$$\dot{x} = \frac{\partial \mathcal{H}}{\partial P_x}\,, \tag{82}$$

$$\dot{A} = \frac{\partial \mathcal{H}}{\partial P_A}\,. \tag{83}$$

$$\dot{P}_x = -\left(\frac{1}{\Delta}\left(\frac{1}{2}\frac{\partial F_2}{\partial x}P_A^2 + \frac{1}{2}\frac{\partial F_1}{\partial x}A^{-2}P_x^2 - \frac{\partial F_3}{\partial x}A^{-1}P_AP_x\right) + A^2\frac{\partial c_2}{\partial x} + A^4\frac{\partial c_4}{\partial x}\right)\,, \tag{84}$$

$$\dot{P}_A = -\left(-\frac{F_1}{\Delta}A^{-3}P_x^2 + \frac{F_3}{\Delta}A^{-2}P_AP_x + 2Ac_2 + 4A^3c_4\right)\,, \tag{85}$$

$$\dot{x} = \frac{F_1}{\Delta}A^{-2}P_x - \frac{F_3}{\Delta}A^{-1}P_A\,, \tag{86}$$

$$\dot{A} = \frac{F_2}{\Delta}P_A - \frac{F_3}{\Delta}A^{-1}P_x\,. \tag{87}$$

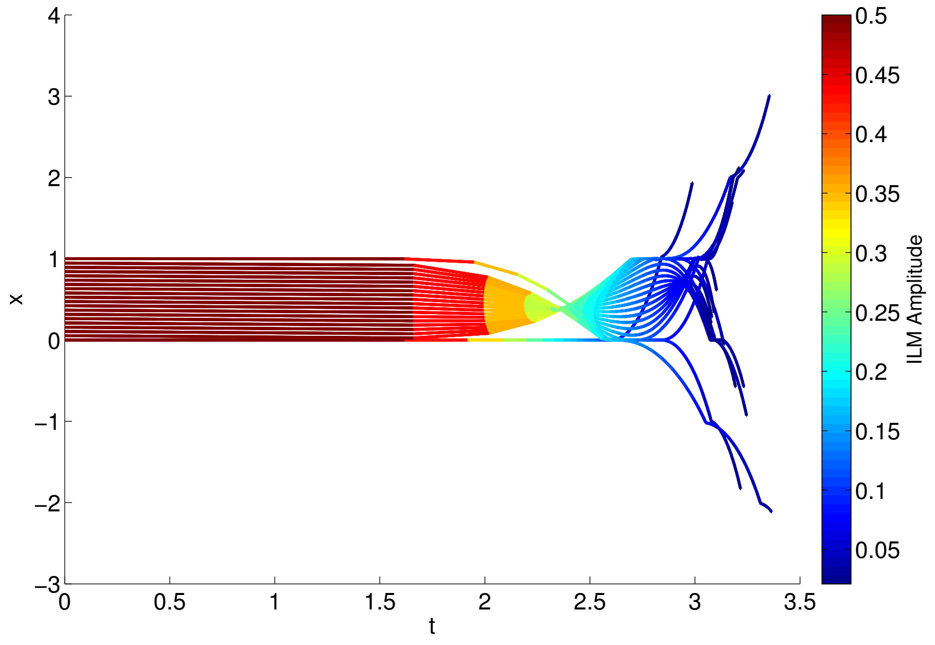
4.3. Discussion. Initial coding of (84 - 87) resulted in ODE runs which did not complete properly, due to the singularity at $A = 0$. Reprogramming with the lowest absolute powers of A dramatically improved the simulations, yet simulation failures continue to happen.

To allow us to pass near the $A = 0$ singularity, we rescale time from t into τ as $dt = A^p d\tau$, where p is the lowest power which removes negative powers from the ODEs. In this case, $p = 3$. This forces a multiplication in the code for the differential equation of $dy = A^p dy$. Essentially, we are rescaling by values of the variable for which there is a singularity, so that as we approach the 'bad' value, time slows down. After a simulation has concluded, we return to normal time by integrating.

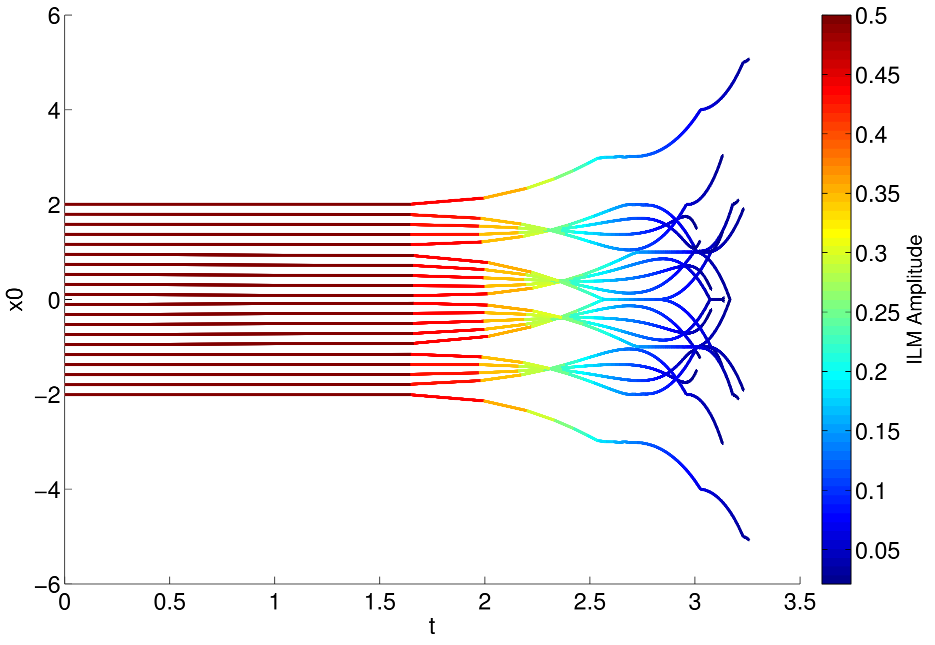
Figure 21 shows a suite of simulations with identical parameters, except the initial position of the ILM x_0. Twenty initial conditions are presented here, with the emphasis of interpretation being on the presence of a defect in the array at $x = 0$. In an undefected system, simulations for initial positions being symmetric between integer values will themselves be symmetric. However, with the defect present, the symmetry is broken. If the defect is negative in nature, i.e. the defect makes α_1 smaller at that pillar, ILMs are attracted to that location. Conversely, if the defect is positive, ILMs are repelled.

In Figure 22 is a set of simulations for which the *initial position* is held constant, and instead the *defect amplitude* is varied. For these, $x_0 = 0.5$, so that the ILM is exactly halfway between two pillars, of which $x = 0$ has the defect. Trajectories are more quickly attracted to the defected pillar, although it appears that they reflect away from pillar locations regardless of defect or not.

Both figures depict a landscape rich with complexity. Defects affect dynamics, which is good, considering that we wish to exploit formation of defects by attachment of molecules. One thing which is lacking in both the derived simplified systems is bidirectionality. Future work will allow there to be two amplitudes, one each in the x and y directions of deflection. The number of parameters in the system is large, and is difficult to study in its entirety.



(A)



(B)

FIGURE 21. Four-variable reduced system, simulations of 20 initial conditions. (a): In the absence of defect, the interval of initial conditions between $x_0 = 0$ and 1 would be perfectly symmetrical; in this plot, a -5% defect at pillar 0 causes a breaking in symmetry. ILMs are pulled towards the defect, but reflect and bounce. (b): This plot demonstrates a -5% defect, again at pillar 0. This plot would be periodic in x_0, but placing the defect breaks this, and the only symmetry is reflection across the defected pillar.

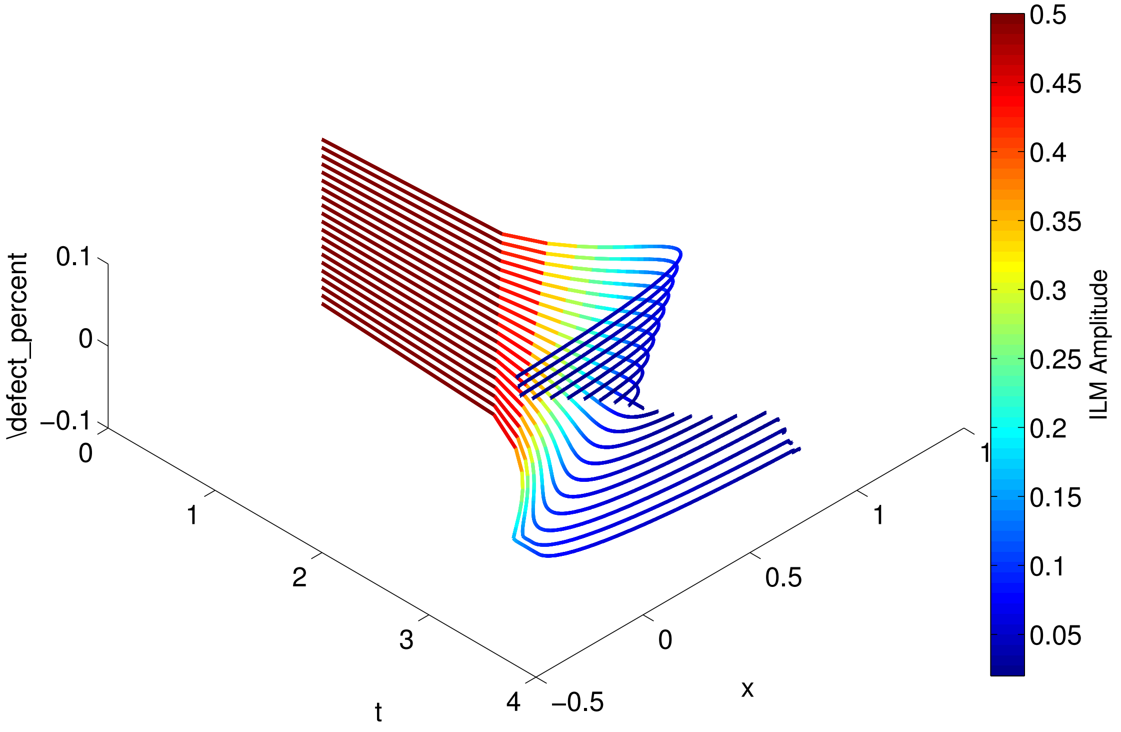


Figure 22. Four-variable simulations of reduced ILM dynamics. The color of the lines indicates ILM amplitude. In the vertical axis is an exploration of defect amplitude for the linear restorative parameter, α_1. Each simulation started from the same point on the array, exactly half way between a normal pillar and one with a defect, at $x_0 = 0.5$. With negative perturbations, the ILM is attracted to the defect, but eventually pins to the normal pillar. Positive perturbations cause the opposite effect.

Chapter 3

Cooperative Robotic Workspaces

Robotics is a recent technological field, coming about only after the necessary dependent technologies and methods had been developed. The concept of an autonomous worker machine has appeared throughout many periods of history, but has been realized only in the last 60 years. Although the term dates to 1923, when the Czech word for "worker" was introduced via theatre to the English language, the first patent for a robot - that is, a programmable machine - appeared in 1954. This is the birth of modern robotics, with George Devol and Joseph Engelberger creating a machine for die casting [44].

Since those early days, the capabilities of robots have expanded greatly. Whereas at first they were single purpose machines, robots now include anthropomorphic and general purpose contraptions. Modern visionary examples include: Honda's ASIMO [45], Lockheed Martin's Multifunction Utility/Logistics and Equipment (MULE) [46], and even do-it-yourself home hexapod robots, such as that from MicroMagic Systems [47]. Given the wide range of robots currently available, at home, in industry, in the military, and in scientific fields, it is no surprise that there is a considerable body of literature on the topic, including a large number of journals and books, as well as professional societies devoted to the field.

Analysis of the functionality of a robot is critical to understanding and maximizing its usefulness, and minimizing costs. One of the fundamental questions to ask is "What is the workspace of this robot?" To begin to answer this question, we must first clarify the notion of 'workspace.'

We define here *workspace* to mean the set of points, or rather, since a workspace is usually not discrete, the subset of ambient space surrounding a robot, which can be reached in at least one configuration. The dimension of the workspace is dependent on the form of the robot. A two- or three-link *planar* robot will have its workspace W as a subset of $\mathbb{R}^2$. In contrast, a fully dextrous robot capable of controlling the placement and orientation of its end effector will have $W \subset \mathbb{R}^3 \times \mathrm{SO}^3$, where SO^3 is the space of three-dimensional rotations.

Since their inception, robots have been prone to failure. Actuators break, sensors give inaccurate readings, hydraulic lines burst, pneumatic bearings give way. Failure in the workplace costs time and productivity, as well as money for repairs or replacement. Failure of a critical component in a remote or dangerous location can be more devastating than failures close to home, as repair or replacement of the robot might be impossible, or impossibly costly, as well as risking damage to property or loss of life. Hence, consideration of fault-tolerance is essential and has been studied extensively.

Difficulties arising from considerations of reliability and safety were pointed out in the review [44]. That paper gives a brief history of robotics, as well as the current state of the art regarding population trends and expectations for a modern robot. Notably, the review cites a requirement of "well-designed robots are to be expected to have a useful life of at least 40,000 working hours, mean time to failure of at least 400 hours, and a mean time to return of no more than eight hours". Various methods of estimation of robot reliability are highlighted, as well as shortcomings. Furthermore, the review clarifies risks to safety due to robots in the workplace, including general overall risks, human factors, accidents, safety systems, and safety methods. Overall, that review emphasizes the need for ongoing improvement of robotic systems with respect to system failure and human interaction.

To help ease safety and reliability concerns, as well as increase robustness of a robotic system to total or catastrophic failure, we propose a method to analyze the workspaces of two robots working to achieve a goal. That is, if a system has two robots with stationary bases, and they are close enough to grasp each other, we present a way to recover lost workspace in the case of one free-swinging joint failure in one robot.

1. Characterizing a Robot

1.1. Parameters and Equations. In order to describe a robotic workspace, we must also describe the robot itself. This description is what gives rise to the forward and inverse kinematic equations. The system of description used here is the one given first by Jacques Denavit and Richard Hartenberg in 1955, at the beginning of the implementation of modern robotics [48]. Their convention, known today as 'DH Parameters', uses a set of four parameters per joint to compactly describe the configuration of a robot. Note here that a joint in DH convention relates to exactly one degree of freedom – compound joints are decomposed into multiple joints with coincident origins. For example, a spherical joint is denoted by a set of three rotary joints. The four parameters are defined as follows:

- θ_i, state variable for revolute joints. Angle between successive x-axes.
- α_i, angle between successive z-axes, about common normal.
- a_i, offset along common normal.
- d_i, state variable for prismatic joints.

Sometimes we give one additional parameter, which indicates the type of joint, t_i: type is $t_i = 1$ if the joint is revolute, and $t_i = 0$ if prismatic. For computational purposes an n-jointed robot could be represented by the matrix,

$$DH = \begin{bmatrix} \theta_1 & \alpha_1 & a_1 & d_1 & t_1 \\ \theta_2 & \alpha_2 & a_2 & d_2 & t_2 \\ \vdots & \ddots & & & \vdots \\ \theta_n & \alpha_n & a_n & d_n & t_n \end{bmatrix}.$$

As an example for the famed PUMA robot (See Figure 23), we have the DH parameters given in Table 3. Note that there may not be universal agreement regarding the DH parameters for a robot. In particular, the PUMA has several sets listed in the literature [49].

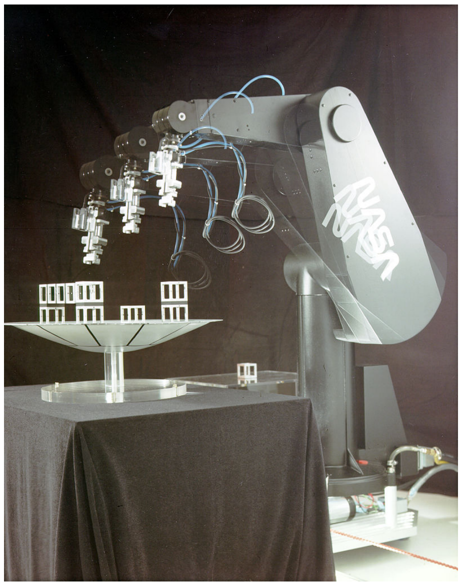


FIGURE 23. The infamous Programmable Universal Machine for Assembly, or PUMA, robot. Picture from the Wikipedia Commons.

The transformation matrix relating the ith joint frame to the $i+1$th is:

$$ {}^{i}T_{i+1} = \begin{pmatrix} \cos(\theta_i) & -\cos(\alpha_i)\sin(\theta_i) & \sin(\alpha_i)\sin(\theta_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\alpha_i)\cos(\theta_i) & -\sin(\alpha_i)\cos(\theta_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{88} $$

These are actually homogeneous matrices, and can be written

$$ {}^{i}T_{i+1} = \begin{pmatrix} R & \tau \\ 0 & 1 \end{pmatrix}, $$

where R is a 3×3 rotation matrix, and the vector τ is the translation vector from the relative origin. By composing these matrices, we can determine the forward kinematic equations for a robot with an arbitrary number of joints. The complexity of the formula grows dramatically with the number of joints, so no general form is given here. For an example of the kinematic equations obtained from these matrices, see Appendix A, which contains equations for a

PUMA robot. For now, denote the joint variables, regardless of whether the joints are rotary or prismatic as θ and simply write the forward kinematics as

$$x = f(\theta). \tag{89}$$

The forward kinematics problem is not difficult to solve, since given the joint variables, in order to find the location and orientation of the end effector we simply multiply the matrices ${}^{i}T_{i+1}$ for all i. Then the location is given by the 3×1 column vector in the upper right corner of the resulting matrix, and the orientation vectors are the column vectors in the first, second and third columns of the matrix, corresponding to the $x-$ $y-$ and $z-$axes of the frame. These correspond to ${}^{0}\tau_n$ and ${}^{0}R_n$ respectively.

"Inverse kinematics" refers to the problem of finding joint variables that configure a robot so as to place the end effector in a certain position and orientation. That is, given a desired position and orientation x, we wish to find θ to satisfy (89); *i.e*,

$$\theta = f^{-1}(x). \tag{90}$$

Inverse kinematics can be solved analytically for special classes of robots, such as the 2R and 3R robots. It involves inverting the kinematic equations, and solving for joint angles or lengths (depending on the joint type), based on a desired position and/or orientation of the end effector. For a versatile and complicated robot such as the PUMA, which has six revolute joints, the problem must be solved numerically with methods such as Jacobian Control (Section 2) or Homotopy Continuation (Sections 1,B).

In general, the dimension of the solution set of the inverse kinematics problem (90) is equal to the difference between the number of joints of the robot, and the dimension of the ambient space. So, if the number of joints is equal to the dimension of the workspace, the solution to the inverse kinematics typically will be a discrete set of points. In contrast, for a kinematically redundant robot, one obtains not pointwise solutions but rather solution curves, surfaces, or hypersurfaces, depending on the degree of kinematic redundancy.

2. Robot Control

There are many ways to control a robot. Some robots, as sometimes seen in primary or secondary education institutions, may be programmed directly with a controller, and the robot will execute *ad nauseum* the program. Another input method is the teaching method, where an instructor/operator performs the desired action using a special teaching apparatus, a computer interface records the motion, and the stored program is transmitted to the robot controller. These two primitive methods of control are static, however, leaving the robot unable to alter its instruction to achieve its goal in the face of changing circumstances, surroundings, or mission objectives.

A current dynamic method of controlling a robot is 'Jacobian Control'. Using the kinematic equations derived from (88), and the mathematical construct of the pseudo-inverse [50], one can develop a dynamic control scheme, which can overcome such problems as nearness to kinematic singularities occurring in the workspace or at workspace boundaries, as well as achieving secondary goals such as obstacle avoidance or minimization of proneness to critical failure [51, 52].

The Jacobian matrix J of an arbitrary transformation is typically given by differentiating the transformation with respect to each variable. Fortunately, in robotics, the Jacobian has another convenient method of computation which is computationally advantageous compared to computation of partial derivatives. Let z_i be the unit z vector of the right-handed orthonormal frame attached to the ith joint of the robot, and p_i the (probably non-unit) vector from the ith origin to the end effector. Then J can be written as the horizontal concatenation of the vertical concatenation of two vectors: the cross product $z_i \times p_i$, and the

z_i vector:

$$J = J(\theta) = \begin{pmatrix} z_1 \times p_1 & \dots & z_n \times p_n \\ z_1 & \dots & z_n \end{pmatrix} \tag{91}$$

The Jacobian arises also from the kinematic equations. Write the forward kinematics as $x = f(\theta)$, and differentiate with respect to time to see,

$$\dot{x} = \frac{\partial f}{\partial \theta}\dot{\theta}. \tag{92}$$

The matrix $\frac{\partial f}{\partial \theta}$ is simply the Jacobian J, and hence (92) can be rewritten as,

$$\dot{x} = J\dot{\theta}. \tag{93}$$

To solve the dynamic inverse kinematics problem, we wish to invert the equation (93) to find $\dot{\theta}$,

$$\dot{\theta} = J^{-1}\dot{x}. \tag{94}$$

However, symbolic inversion of J is difficult or impossible for complicated robots, and J is often far easier to work with numerically than symbolically. Furthermore, we will frequently work with robots having nonsquare J, and sometimes J will even become singular! This gives rise to the following definition: a *kinematic singularity* is any point (or curve, or surface, etc.) in the workspace (boundary inclusive) of a robot, where the Jacobian (91) becomes singular. This often occurs internally to the workspace at a point where the dimension of the inverse solution set increases. Singularities occur also at the boundary of a workspace – where independent movements of several different joints induce the same infinitesimal movement at the end effector.

One of the key properties of a matrix often exploited in robot control is the *singular value decomposition* (SVD). The SVD is the factorization of a matrix A into the product of three matrices as $A = UDV$. The matrices U, V are unitary matrices, and in robotics correspond to transformations of coordinate frames; the matrix D is a diagonal matrix holding the singular values of A, and indicates compression or stretching of primary directions of the change-of-coordinate matrices U, V. The SVD makes it apparent when the robot is at a kinematic singularity, because at such a location one of the entries in D which is otherwise nonzero will become zero.

There are two generic spaces that arise in robotics: the forward space – the workspace we have been previously discussing; and the inverse space – which could easily be called the 'joint space' or 'parameter space', or even 'angle space'. Regardless of what one calls them, these two spaces have very different functions. The forward space can be understood as far as the robot interacts with the environment and its tasks. The joint space is interpreted in terms of control; as is seen in Section 1.1, the controller (whether human or computer) must find trajectories in joint space so as to place the robot in a suitable configuration in forward space.

Robot safety and reliability is greatly increased if the control and mapping of workspaces is well-understood. To this end, many researchers have proposed algorithms or methods for doing so. As a primary example, for robots with continuously variable joints, one can use the 'Jacobian Method' to test a point P in the ambient space for workspace membership. A generic formulation of the method is the following.

(1) Scatter samples, either randomly or on a mesh, in joint space,

(2) Transform the joint space samples forward to obtain a workspace sampling,

(3) Find the sample point nearest the candidate point, call it N,

(4) Attempt to move robot from N to P,

(5) Declare P in or out of workspace based on whether the manipulator reached P, after reaching termination condition of Jacobian control method.

More advanced methods for mapping objects such as obstacles and barriers into joint space for trajectory planning have been explored, including in the realm of cooperating robots. The paper [53] gives two algorithms for mapping paths for planar robots cooperating to move a rectangular object, modeling the assembly as a six-link closed chain, while [54] gave an efficient method for mapping obstacles into joint space, as well as describing the topology of an obstacle-riddled space transformed into joint space. There are also methods for robots with discretely controlled joints, such as binary actuators [55].

2.1. Fault Tolerance.

The detection and isolation of faults in various components of a robot, such as sensors, controllers, and actuators, as well as collision with environmental elements, is critical to maintaining a high level of usefulness. Various schemes for doing so have been proposed.

One such method was an architecture of fault adaptive control (FAC), where the functionality of the robot is broken into three entities: the controller, interface, and plant [56]. This study aims to detect, isolate, and identify faults occurring in the robotic system. In a similar fashion, [57] proposes an algorithm for detecting single actuator or sensor failure, and estimates the fault signal, working with residuals of expected versus measured signals. Sensorless on-line fault detection and 'passivity-based control' by modifying a Newton-Euler method were treated in [58].

In the case of a parallel manipulator, for example a Stewart-Gough platform, failures of various components are just as disastrous. However, [59] describe methods for improving fault tolerance in such manipulators, such as designing the manipulator to have additional degrees of freedom. The paper further gives measures of fault tolerance, which can be used for additional design parameters and path planning. Parallel manipulators are often used as pointing devices, and [60] indicates the use of a Stewart platform to "achieve active pointing, passive isolation, and fault tolerance", even to the degree that several actuators fail simultaneously. Generic parallel manipulators are examined in [59], which treats various kinds of actuator faults and methods to deal with each.

Robust performance of a robot and fault tolerance certainly involves detecting and mitigating failures, but one ought also to act to anticipate failures. By instructing the robot to move in certain ways, we can improve the ability of the robot's performance to degrade gracefully. Decomposing a task into primary and secondary goals is one method of dealing with these problems, namely by exploiting kinematic redundancy in the system [61], since a fully functional kinematically redundant manipulator's Jacobian always has nontrivial nullspace. Instead of the inverse kinematics solution as in (94), we can improve the solution using

$$\dot{\theta} = J^{+}\dot{x} + \left(I - J^{+}J\right) z, \tag{95}$$

denoting the pseudoinverse of the Jacobian as J^{+}, I being the suitably sized identity matrix, and vector z a quantity we wish to optimize, frequently calculated as the gradient of some function h. Measures of fail-tolerance include analysis of the singular values of the Jacobian matrix for a robot, such as the product of the singular values [62], and the 'manipulability index' as defined in [63].

Prioritization of tasks subject to other constraints (*e.g.*, environmental) was treated in [64, 65]. Paper [66] studied the anticipation of free-swinging joint failures, focusing on control of the manipulator so as to minimize the effects of swinging due to gravity, and measurement of the risk of damage due to a free-swinging failure. The paper [67] investigated the topic of the degree of kinematic redundancy, regarding the number of extra degrees of freedom necessary and sufficient to tolerate the failure of any one joint. Other methods of creating and exploiting kinematic redundancy appear in [59, 60, 67, 68, 69].

Robots can be designed and operated with fail-tolerance in mind, such as dual actuators at joints [68]. This is a clever idea, using a velocity-summing mechanism at each joint to allow two actuators to work in tandem to supply the torques. If one fails the other can pick up seamlessly. Reconfigurability is yet another method to allow a robot to deal with a wide variety of changing circumstances. The scheme presented in [70] is a robot that can essentially change on-the-fly its DH parameters by adjusting cylindrical joints. Reconfigurability offers additional benefits beyond failure tolerance; it also allows the robot to occupy a minimal volume for transport, and suitability to a variety of tasks.

Last but not least, we might wish to predict failure, to quantify weaknesses in design. Determining the reliability of a robot via interval methods, as opposed to Monte Carlo or fuzzy logic, in fault-tree analysis appeared in [71].

Fault-tolerance and robustness play important roles in the design of autonomous systems, including robotic arms. In this paper, we consider autonomously operated robotic systems that are deployed into a hazardous or remote location, such that the repair on a failed joint is impractical or impossible. If the system has to operate for an extended period of time, we want the system to operate to the best of its remaining physical capability after a joint failure.

3. Cooperative Workspace Study

This section closely follows the submitted papers [1, 2].

We consider autonomously operated robotic systems that are deployed into a hazardous or remote location, such that the repair of a failed joint is impractical or impossible. If the system has to operate for an extended period of time, we want the system to operate to the best of its remaining physical capability after a joint failure. Certainly, there are many ways for a robotic system to fail. Sensors give false or no readings; electronic components break; controllers fault; actuators seize or give way. Our goal is to supplement present methods regarding graceful joint failure, by informing designers and operators of possibilities for assistance to a broken robot by a functional one, should one be available. In the event of a free-swinging failure, our method provides information about optimal placement of a predetermined socket or other suitable apparatus on an articulated arm to allow assistance from a functional robot. Even with an actuator with non-free-swinging failure, such as unexpected resistance or friction, our method could contribute to mission success. The failure-proof of systems has been particularly important for applications such as space-faring vehicles [72].

We also note that a free swinging joint failure is, of course, a mathematical abstraction. In this paper we consider the free swinging failure as a failure with sufficiently low resistance that can be overcome by interacting robotic arms. The second arm provides torques and forces that may be impossible to obtain using duplicate motors and other considerations in the original arm, precisely because it is assumed to be highly complicated, so any further number of components in the arms could be undesirable.

An important consideration for fault-tolerance and cooperation is the multiplicity of solutions to the equations describing the position of the end effector of the robot as a function of control parameters. It is generally desirable for a robot to have several ways of reaching a particular point in space, for example, to evade obstacles that appear during the operation. Even in the case of built-in redundancy, where there is a one- or multi-variable family of solutions, it is interesting to consider the exact number of isolated solution sets for each point in the workspace, as some solution sets are more advantageous than others.

To make our consideration more realistic, suppose a system operating in a remote environment possesses several robotic arms for various tasks. Should one joint in one arm fail, the presence of a second arm may open the way to preserve some of the workspace of the failed arm. In particular, if the second arm can grasp the first, perhaps some of the lost workspace can be restored. In this work, we shall assume that the relative position of the bases for robotic arms cannot change, and only one joint fails at a time.

As far as fault-tolerance consideration of a *single* robot goes, the multiplicity of solutions may not be a relevant quantity. For the purpose of restoring workspace using *cooperating* robotic arms the multiplicity of solutions is of crucial importance. As we shall see below, the workspaces of cooperating robots can sometimes consist of several isolated sets.

The main goal of this paper is to outline a method that is alternative to the traditional ways of workspace computation. The main difference is that we use sampling *directly in the workspace*, and therefore are not constrained by the possible singularities of the forward kinematics mapping. Our method, which is based on the applications of ideas in algebraic geometry (homotopy continuation) to the solution of polynomial problems, can provide the exact number of solutions for each particular point of the workspace. In particular, the inverse kinematics problem can be cast as a polynomial system, and homotopy continuation provides a means for efficiently producing numerical approximations of *all* isolated complex and real-valued solutions of the polynomial system. Methods of algebraic geometry have been applied in the kinematic description of robots, see for example [7, 73, 74, 75, 76]. However, analysis of the cooperation of multiple arms via the tools of numerical algebraic geometry has never been explored.

We shall outline a particular application of the method: computation of workspace and optimization of the grasping point for two cooperating robots, in case of joint failure of the first robot. This example was chosen as a realistic demonstration, leading to interesting shapes of workspaces with several sets of multiple solutions. For the optimization, we ask two questions: *a)* What is the best placement of the two robots in relation to each other? *b)* Is there a best place for the second arm to grasp the first? Clearly, the answer to both questions depends on the choice of measure for the post-failure workspace, which is problem dependent. We will introduce a measure using the multiplicity of the solutions to the inverse kinematics equations, which can be suitably combined into a single measure depending on the application.

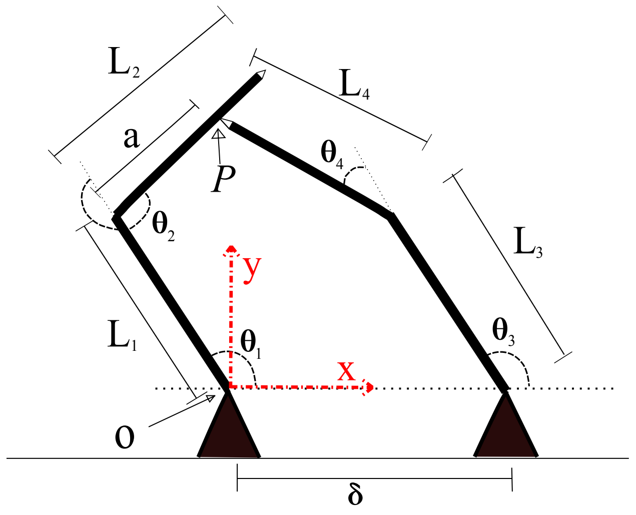


FIGURE 24. Two robots in grasping configuration, with functional robot contacting the disabled unit at point P. This is a simplified 2D model of the general 6D position and orientation scenario.

The key feature of this research is the application of methods of algebraic geometry to the description of fault tolerance of two cooperating robots. These techniques are guaranteed to give all isolated solutions to algebraic equations describing the positions of robotic arms, and thus robustly find solutions in arbitrarily complex settings, such as multiple joint values in isolated regions of joint-space. Thus, these techniques are capable of *autonomously* completing the workspace analysis, which can be useful in robots operating with high degree of independence, with little or no communication with the operator.

3.1. FORMAL PROBLEM STATEMENT. Given two articulated arms, we optimize the placement of grasping sockets to maximize post-failure workspace (we assume a socket attachment mechanism of the two arms as considered in [70]). That is, if one robot has a free swinging joint failure as in [77, 78], we would like to ensure that when the functional robot joins at the socket to assist in completion of tasks, the resulting cooperative workspace is as large as possible. A simplified example of this problem is in Figure 24.

Each robot has its own independent pre-failure workspace, W_1, W_2; when the two robots are placed near enough, there is an intersection workspace $W_\cap = W_1 \cap W_2$. In a grasping configuration, there is a post-failure workspace W_f, which contains all remaining workspace locations Robot 1 can reach, if Robot 2 attaches to a socket location on Robot 1. Both $|W_\cap|$ and $|W_f|$ depend on the separation between the robots, as well as their orientation. Note that the intersection workspace does not depend on the grasping point, as it reflects the workspace prior to failure for each robot. In both pre- and post-failure workspaces, we take into account joint limits, which are an important consideration for practical applications. Indeed, robots typically have limited range of movement for each joint, which reduces the size of all workspaces. These limits introduce a dependence on the relative rotations of the robots to each other, and to the measures of the various workspaces.

The parameters describing an optimal configuration of two cooperating robots include: separation between cooperating arms, relative orientation of the bases, and location of grasping point. In this paper, we do not optimize with respect to base position of each robot, as we consider that the position of the bases must be given from design limitations, *e.g.* the placement of power cords and motors on the apparatus. Nevertheless, our method is capable of optimizing socket placement for different base separations, and this is demonstrated in the examples.

In order to quantify post-failure workspace size and find an optimal configuration, we introduce a maximizing objective function Ω. There are many possible objective functions one can imagine, and the right choice depends on the application. In this paper, we define a measure of workspace W and objective function Ω through the multiplicity of solutions at a given point x, denoted $m(x)$, and r, the number of sample points:

$$\begin{aligned} |W| &= \frac{1}{r}\int_W m(x)\mathrm{d}x, \\ \Omega &= (1-\lambda)|W_f| + \lambda(|W_1| - |W_\cap|). \end{aligned} \tag{96}$$

We have chosen (96) for an example objective function in order to balance between the benefit of having a second robot, and the maximization of the post-failure workspace. A configuration which imparts entirely distinct workspaces would maximize $\lambda(|W_1| - |W_\cap|)$, but W_f would be an empty set. Contrarily, we might find a configuration which results in full restoration of W_f upon entering grasping stance, but which makes the pre-failure workspaces overlap greatly, so $\lambda(|W_1| - |W_\cap|)$ would be small. We prefer to balance between pre- and post-failure benefits of having two robots, and (96) is one way of doing this.

It seems that one obvious way to maximize $|W_f|$ could be to set $\delta = 0$ so the two robots have exactly the same base point, and make the two robots identical. However, this may be an impractical situation. First of all, there may be much greater benefit by compromising and having smaller intersection workspaces by placing the robots further apart; second, depending on robot geometry, it could be awkward or impossible for two identical robots to operate from the same base point.

3.2. Homotopy Continuation.

We recast the inverse kinematic problems for computation of (96) in the form of polynomial systems, and use the methods of numerical algebraic geometry, particularly homotopy continuation, to find the solutions of these polynomial systems. A more comprehensive treatment of this method appears in Section 1.

In general, given an arbitrary (not necessarily robotic) set of polynomials $\bar{f} = f_1, \ldots, f_N$ in N variables for which we seek the solutions, homotopy methods begin by choosing and solving some other related polynomial system $\bar{g} = g_1, \ldots, g_N$ for which the solutions are easily found. By varying the coefficients in $\bar{g}$ with those of $\bar{f}$, mathematical theory guarantees that, with probability one, the solutions will vary continuously, thus forming *solutions curves* or *paths* from the solutions of $\bar{g}$ to those of $\bar{f}$. These solution curves may then be tracked numerically with standard predictor/corrector methods, such as a combination of Runge-Kutta and Newton's method. Further details may be found in [7, 79].

There is a setting in which homotopy methods are particularly effective, and of which we make use in this paper. If instead of solving one polynomial system $\bar{f}$ we wish to solve a large number of polynomial systems that differ only in coefficients (i.e., we wish to solve $\bar{f}(\bar{p})$, where $\bar{p}$ is some set of parameters), there is an especially efficient homotopy method known as a *parameter homotopy.* The key of the method is to have a simple system $\bar{g}$ for $\bar{p} = \bar{p}^*$, and find a path in the parameter space leading to the system of interest as $\bar{p} = \bar{p}_0$. When constructing a starting system $\bar{g}$, it is possible that there will be many more solutions of $\bar{g}$ than $\bar{f}$. As a result, it may occur that hundreds of thousands of solution curves are tracked in order to find only a few solutions. In the special case of a parameter homotopy, we first solve $\bar{f}(\bar{p})$ at a single instance of $\bar{p}^*$ (typically chosen as random complex numbers, for theoretical reasons). This stage may involve a number of superfluous paths, which waste computation time. All other instances of $\bar{f}(\bar{p})$ may then be solved by simply following the handful of finite solutions from $\bar{p}^*$ to any other choice of $\bar{p}_0$. For example, the system used to solve W_f, grasping on the third link, for an initial random complex parameter choice, we need to follow 20736 paths to find the *16* solutions of interest. Then, for all other points in the parameter space, it suffices to follow just 16 paths.

Again, there is much theory and detail underlying these methods, most of which may be found in [7]. During the process of homotopy continuation, a certain number of paths will fail as they near a singularity in parameter space. In the context of robotic workspaces, these failed paths indicate proximity to workspace boundary or kinematic singularity. Right now, we are not using this information, and simply ignore failed paths; however, this property could ultimately be used for more accurate and efficient prediction of, for example, workspace boundaries.

Several software packages are available for these sorts of computations. We use a freely-available software package named *Bertini* [6], which has been under development for the past decade by Dan Bates, J. Hauenstein, A. Sommese, and C. Wampler. The repeated calls to *Bertini* were parallellized for efficiency using the *Paramotopy* method, and as described in [80].

3.3. Workspace Computation. In this section, we describe our solution to the problem outlined in Section 3.1. We start with notation. Let the separation between bases be δ and without loss of generality let this translation between coplanar bases be entirely along the x-axis; we sort out-of-limit solutions in a post-processing procedure. We consider $0 < \delta < \delta_{\max}$, the largest value of δ corresponding to the the sum of the lengths of each robot fully extended. The normalized distance to the grabbing point or socket, measured from the failed joint, is denoted by a, with $0 < a < 1$; the point at which a socket is attached to the left robot, hereafter Robot 1, is P; a test point in space is Q; and the origin at the base of Robot 1. Finally, let the fully functional machine be known as Robot 2. A simplified version of this notation is found in Figure 24.

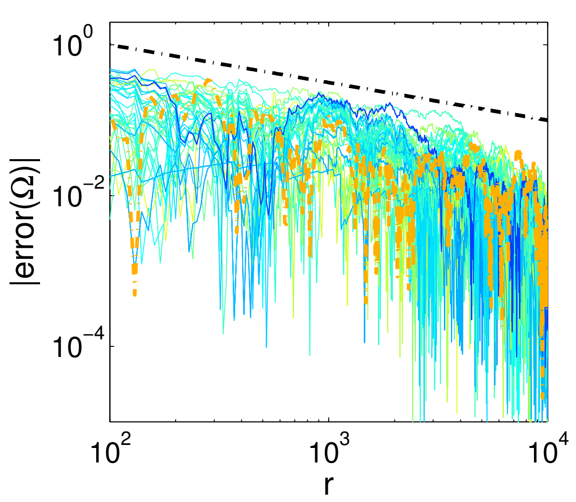
There is some probability that each joint could fail, so it is important to take into consideration the placement of a socket on each link (of non-zero length). We use the Denavit-Hartenberg convention to describe the configurations of the two robots, and describe the contact point P in terms of the DH parameters for Robot 1, as P is some distance a from the origin of the previous frame. A spatial sample Q can be said to be in the post-failure workspace of Robot 1 for a parameter pair (δ, a), if there exists a set of joint angles such that end effector of Robot 1 is at Q, and the end effector of Robot 2 is at P.

The equations for the inverse kinematics problem for each step of the method are first solved via a standard homotopy run at $\bar{p}^*$, a random point in complex parameter space. Then all subsequent runs at points in the workspace are treated as parameter homotopies, beginning at $\bar{p}^*$. For W_i, the parameters are spatial coordinates of Q; for $W_\cap$, we add parameter δ; and for W_f we add a. *Bertini* solves over $\mathbb{C}$ for each variable, so we find solutions for each point, regardless of whether it lies inside the workspace. However, the points that have solutions for which all variables are numerically real-valued are those lying within the workspace, while those with nonzero imaginary component lie outside.

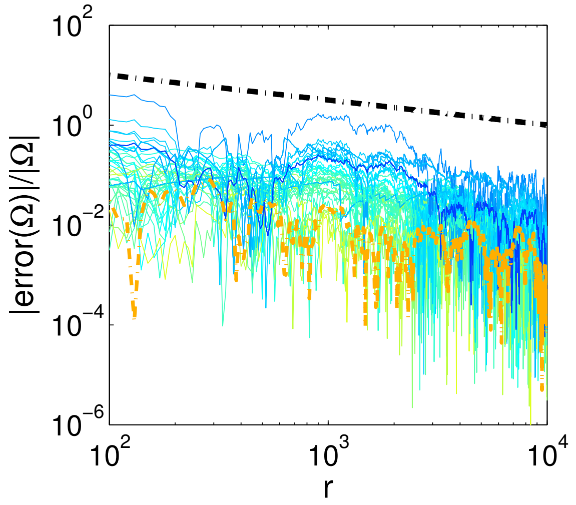
We note that the inverse kinematics equations for a robot with rotary joints are trigonometric in nature. In order to use homotopy continuation, we write the equations in polynomial form by treating each sine-cosine pair as a separate variable, mapping $\cos(\theta_j) = c_j$ and $\sin(\theta_j) = s_j$ and coupling with the algebraic condition $c_j^2 + s_j^2 - 1 = 0$.

We compute the initial pre-failure workspaces for each robot via a random sampling method on a subset of the space guaranteed to contain the workspace, and estimate the measure of the workspaces using the number of samples in the workspace pointwise multiplied by multiplicity factor for each point, as in (96). The error by this method is typical of Monte Carlo techniques, which is typically bounded by $r^{-1/2}$, where r is the number of samples. In Figure 25, we present a plot of absolute error for a host of configurations for the 3D example appearing below; the dashed line indicates the falloff rate associated with the $-1/2$ power law.

For the case of $N \geq 3$ joints working in three dimensions without orientation, there are three algebraic inverse kinematic equations in $2N$ variables, coupled with N Pythagorean identities, when computing W_i. As long as the number of joints equals the number of degrees of freedom in the ambient workspace, *Bertini* will find all solutions, and we will be able to measure $|W_i|$. For kinematically redundant robots, the joint space consists of a set of higher $N - 3$ dimensional manifolds, which could be described by defining a mesh of the same dimensionality as coordinates on the joints. Our method readily applies to this higher dimensional problem. However, the issues we are facing are related to the curse of dimensionality, and hence in the computation of the data as a whole, not in the solution finding at a particular point in space, which is fast. For example, with $N = 5$ joints, and a three dimension workspace, each point in the workspace corresponds to a twodimensional manifold in joint space. To cut down the manifold to be zero-dimensional for solving via



(A) Absolute Error



(B) Relative Error

FIGURE 25. Error estimate of Ω for a host of 3D configurations with varying (δ, a) as a function of the number of sampling points r. The dashed straight line shows $-1/2$ power law, which is a typical upper bound for a Monte Carlo based method. The colors in this plot correspond to the Ω values appearing in Figure 29(b). Blue lines have low Ω value, and hence, a low number of samples are in the corresponding workspace; high variation in absolute error occurs for these lines, and relative error is higher. In contrast, the thicker orange line is computed from many points, absolute error fluctuates less, and the relative error is lower.

Bertini, we could discretely sample each pair of joints possible, and solve for the remaining three. However, combinatorial growth issues arise, *e.g.* if we wanted to solve each spatial sample for each possible combination of joints. The work load increases with the growth of dimensionality for both the robot and ambient workspace.

3.4. Two Examples.

3.4.1. *2D Case: Two Link Planar Robots.* We start with the illustration of the method for the two dimensional example that is shown in Figure 24. This was first considered this [1]; the previous work has been expanded to include more general method and examples, as in Section 3.4.2, where the more challenging three-dimensional problem is considered.

The robots are identical, both having two joints, all link lengths having length one; the DH parameters are summarized in Table 1. To demonstrate the method, we show results for an initial sampling of $r = 10^4$ points, taken from the two dimensional rectangle

TABLE 1. Denavit-Hartenberg Parameters for two-link planar robot.

θ_j	α_j	a_j	d_j	$\theta_{j,min}$	$\theta_{j,max}$
θ_1	0	1	0	$-120°$	$120°$
θ_2	0	1	0	$-120°$	$120°$

$Q \in [-2, 2] \times [-2, 2]$. Of these points, 7836 had at least one real solution; the estimated *non-normalized Euclidean* size of the workspace would be $4^2 \times 7836/10000 \approx 12.5$. Note that in principle, Ω can reach negative values. This happens when the workspace W_1 has solutions of low multiplicity, whereas $W_\cap$ has solutions of higher multiplicity. Even though pure Euclidian measure of W_1 is of course larger than $W_\cap$, the measure involving solution multiplicity that we are using here does not necessarily obey that rule. If one were not concerned with multiplicity, one could use an unweighted version of (96), which would make this positive.

Here, we define c_j and s_j to be the cosine and sine of the joint θ_j values, respectively, with $j = 1, 2$ corresponding to the failed robot and $j = 3, 4$ corresponding to the assisting robot. Let joint index $j \in \{1, 2, 3, 4\}$, so that we have a sine-cosine pair for each of the two joints in the two robots. In this notation, our equations for this step are,

$$0 = \begin{cases} c_1c_2 - s_1s_2 + c_1 - x \\ s_1c_2 + c_1s_2 + s_1 - y \\ ac_1c_2 - as_1s_2 + c_1 - (c_3c_4 - s_3s_4 + c_3 + \delta) \\ as_1c_2 + ac_1s_2 + s_1 - (s_3c_4 + c_3s_4 + s_3) \\ s_j^2 + c_j^2 - 1 \end{cases} \tag{97}$$

The first step in the computation is to estimate the pre-failure workspaces for each robot, as described in Section 3.3. Secondly, we intersect and obtain $W_\cap = W_1 \cap W_2$, for a specified set of δ values. Finally, for each pair of values (δ, a) for which we wish to estimate the post-failure grasping workspace, we solve a set of equations (97). For this example, we computed W_f for the set of δ and a, with $|W_f|$ normalized with respect to $|W_1|$.

TABLE 2. Estimates on the measure of workspaces and Ω with $\lambda = 1/3$.

δ	a	$\lvert W_1\rvert$	$\lvert W_\cap\rvert$	$\lvert W_f\rvert$	Ω
0.28	0.08	1.000	2.039	1.526	2.826
0.28	0.41	1.000	2.039	1.741	0.333
0.28	0.74	1.000	2.039	1.896	0.333
1.61	0.08	1.000	0.674	2.197	2.708
1.61	0.41	1.000	0.674	2.254	2.333
1.61	0.74	1.000	0.674	2.645	2.080
2.94	0.08	1.000	0.079	2.125	1.637
2.94	0.41	1.000	0.079	2.646	1.490
2.94	0.74	1.000	0.079	3.353	1.425

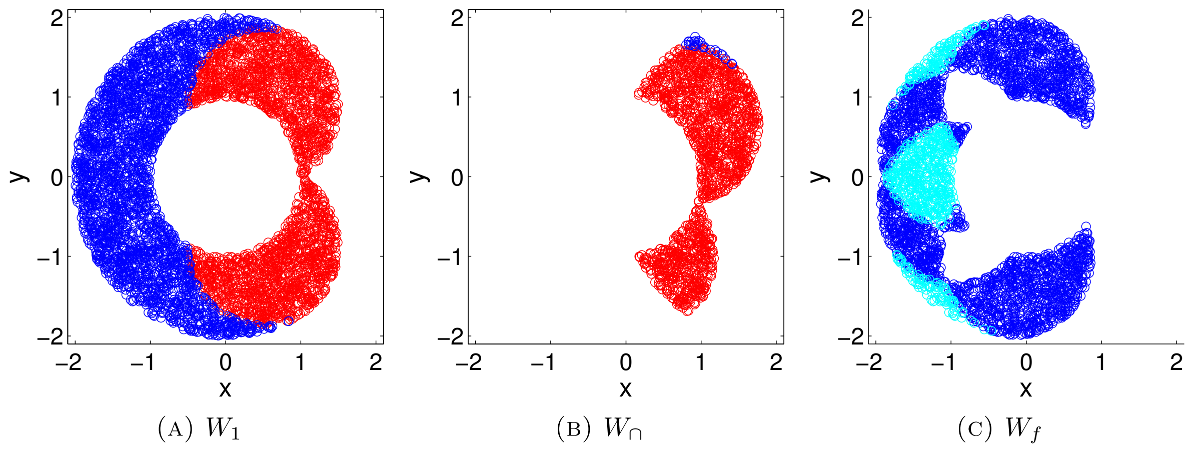
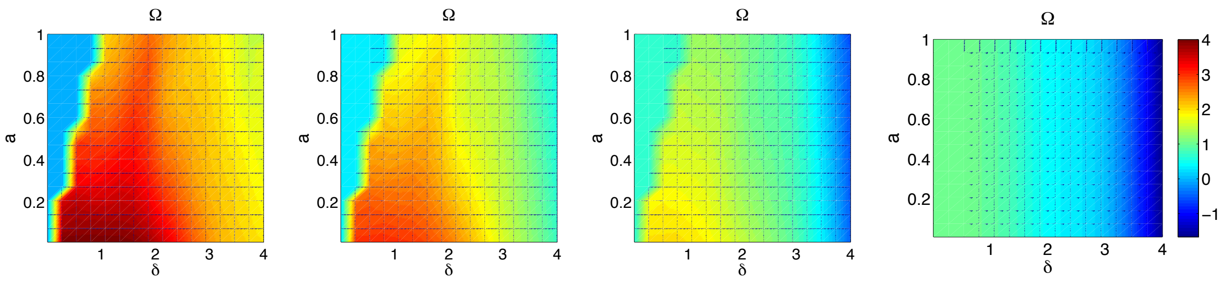


FIGURE 26. Examples of joint-limited workspaces, for $\delta = 1.87$, $a = 1$, and grasping on the *first* link, rotated relative to one another. Red indicates a point having one solution, blue indicates two, and cyan indicates four.

As the separation increases so that the workspaces barely overlap, Robot 2 may only grasp Robot 1 near the end effector, and the resulting W_f is small. See Figure 26, and column four of Table 2. In these plots, cyan area is fully accessible from 4 configurations, blue corresponds to 2, and red is reachable from only one configuration. The inaccessible area generally increases when δ is increasing, and larger δ lead to smaller post-failure workspaces for fixed a. The converse is not true: $|W_f|$ is not a monotonic function of a for a fixed δ. It is interesting to note that our method computes explicitly the number of configurations reaching the desired point in a post-failure workspace, even in the case when the configurations belong



(A) $\lambda = 0$. (B) $\lambda = 0.333$. (C) $\lambda = 0.666$. (D) $\lambda = 1$.

FIGURE 27. Objective function contours for a grid of (δ, a) pairs for 2D cooperating robots, with the broken robot being grasped on the first link.

to isolated domains in the workspace. Such problems are usually challenging for traditional methods of workspace computation.

Finally, in Figure 27 we show the objective function Ω from (96) to determine the optimal ball joint placement. For weighting factor $\lambda = 0$, Ω is simply the value of $|W_f|$, and the maximizer is $\delta \approx 1$, $a \approx 0$. With $\lambda = 1/3$, the optimal joint location is still at the base of the second link, but occurs with separation $\delta = 0.5$. Increasing λ further makes the size of the intersection workspace overpower the post-failure workspace, and by $\lambda = 1$ the maximum of Ω is invariant with respect to a. Therefore, the weighting factor λ plays a critical role in the determination of the optimal grabbing point.

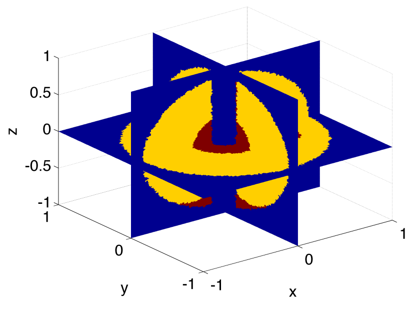
3.4.2. *3D Case: Three Joint Manipulators.* Equations determining the kinematics of spatial robots are more complicated than those of planar manipulators. This concerns both the higher number of relevant equations, due to the higher number of links in a typical 3D robots, and the structure of equations describing the motions. Yet, the method for optimizing cooperative workspaces remains fundamentally the same, as those equations can be brought to an algebraic form and then solved using the methods outlined in Section 3.3. Thus, in this section we only write a sketch of the method in 3D.

Consider two cooperating PUMA robots, ignoring orientation of the end effector. In this paper, we use DH parameters defined as in Table 3. Treating the first three links of the PUMA as our robot, we ignore the wrist; we do this to limit the curse of dimensionality and reduce the number of points to analyze. Instead, we use a tool frame to translate from the arm to the end effector. The frame we used retains the orientation of the arm, and translates along the final z-axis.

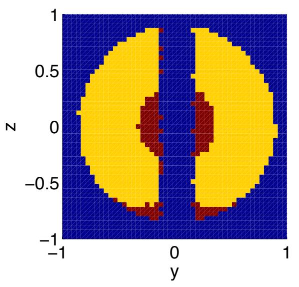
Each of the workspaces W_1, W_2, $W_\cap$ will have six equations in six variables, which are solved via Bertini after bringing the trigonometric part of these equations into algebraic form. Correspondingly, there will be twelve equations defining the post-failure workspaces with twelve variables. In order to find $W_{1,2}$ we sample randomly an oversized rectangular box surrounding the robot. In order to determine good bounds for the workspaces, we first do forward kinematics by randomly sampling the three joint variables $\theta_j \in [0, 2\pi]$. The result of

Table 3. Denavit-Hartenberg Parameters for PUMA robot.

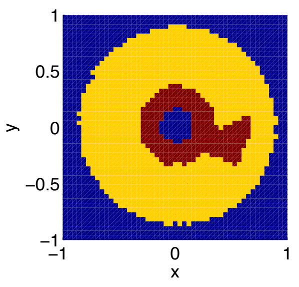
θ_i	α_i	a_i (m)	d_i (m)	$\theta_{i,min}$	$\theta_{i,max}$
θ_1	0	0	0	$-160°$	$160°$
θ_2	$-\pi/2$	0.4318	0.2435	$-225°$	$45°$
θ_3	0	-0.0203	-0.0934	$-45°$	$225°$
θ_4	$\pi/2$	0	0.4331	$-110°$	$170°$
θ_5	$-\pi/2$	0	0	$-100°$	$100°$
θ_6	$\pi/2$	0	0.5625	$-266°$	$266°$



(A)



(B)



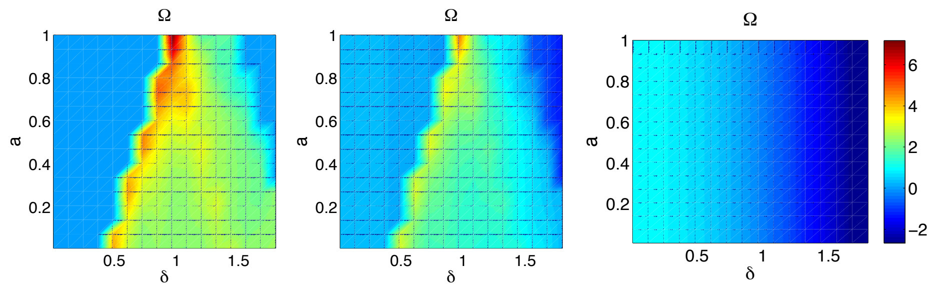
(C)

FIGURE 28. Top: Slices of the pre-failure PUMA workspace W_1 by the planes $x = 0$ (b) and $z = 0$ (c). (a): Combined picture of workspace with three slices $x = 0$, $y = 0$, $z = 0$. Yellow color: areas accessible the angles satisfying joint limits given in Table 3. Red color: real solutions violating joint limits. Dark blue region: inaccessible.

this estimate is that the cubic box $[-0.9, 0.9] \times [-0.9, 0.9] \times [-0.9, 0.9]$ contains W_i, and is thus a good starting point for finding the cooperative workspaces in which we are interested.

The PUMA has joint limits that reduce its workspaces significantly. The limits we use for this example appear in Table 3. Because joint limits are written as inequalities $\theta_{j,min} \leq \theta_j \leq \theta_{j,max}$, they are not algebraic equations. We simply use these joint limits in post-processing, only selecting the suitable joint angles among all real solutions.

Results for various workspace computations are shown in Figure 28. These data are based on 10^4 points in (x, y, z) space. Of particular interest is the presence of voids inside of workspaces. Because the data plotted is generated from a random sampling, to plot nicely we called the Matlab method `griddata`. Forbidden area is represented by the dark blue color. The red and yellow colors show the accessible work space, with the yellow areas being accessible only if the joint limits are satisfied. The resulting workspace is essentially a torus, although the realization we have chosen makes it hard to visualize as a volume in the three dimensional space, because of the relative narrowness of the "hole". Instead, we have chosen to represent the workspace through the slices. Two slices by the planes $x = 0$ and $z = 0$ are shown in the top of the figure, and the combined figure presenting the slices is shown in the bottom. The voids in W_i come from the offsets of the arm, and they expand for cooperating



(A) $\lambda = 0$. (B) $\lambda = 0.333$. (C) $\lambda = 1$.

FIGURE 29. Considering grasping on the second link to restore workspace after failure of joint 2 of a PUMA robot. a) Measurement of the intersection workspace $|W_\cap|$, as a function of δ. b)-c) Contours of the objective function Ω versus (δ, a) for 3D cooperating robots. Note that for the case a), $\lambda = 0$ and the objective function Ω is equal to the measure of post-failure workspace $|W_f|$.

robots. Our results show that great care needs to be taken in designing and arranging robotic arms for cooperation, as it may lead to large inaccessible regions of workspace.

We also present the plot of objective function versus parameters δ and a in Figure 29. To calculate Ω, we compute each of the workspaces W_1, $W_\cap$ and W_f. For the purposes of generating this figure, $W_\cap$ is computed over a discretization of $0 \leq \delta \leq 1.8$ into 16 values. Finally, we compute W_f, for a particular value of δ and a. The results are repeated over a 16×16 regular grid of $0 \leq \delta \leq 1.8$, and $0 \leq a \leq 1$, and considering grasping the failed robot on each of the three possible links. Thus, the total number of parameter combinations we considered for W_f is $7.68 \cdot 10^6$.

The objective function landscapes presented in Figure 29 for the PUMA robot are similar to those for the 2D planar robot above in Figure 27. Because each joint could fail independently of the others, we consider a grasping location for each link of the robot; hence, we have an objective function landscape for each of the three links of the PUMA. However, the landscapes for each joint are similar, so only those for the second link are presented here.

As with the 2D example above, the weighting factor λ plays a crucial role in optimizing. The limiting cases of $\lambda = 1$ and $\lambda = 0$ return simply $(W_1 - |W_\cap|)$ or $|W_f|$ respectively. The maximum values of Ω occur with $\delta \approx 0.9$ meters between the robots. Around $\lambda = 1/3$, we put more emphasis on increasing the remainder of workspace accessible in grasping configuration; Ω clearly indicates to grasp near the end of the second link. In any case, the optimal distance and grasping location depends on the link and on λ, making user preference crucial in the optimization procedure.

4. Conclusion

Robotics has come far since their inception. From anthropomorphic to highly redundant and robust, modern robots are used in many applications. They have been well-characterized with respect to operational parameters, control theory, workspace analysis, and graceful failure.

We used homotopy continuation, as implemented in *Bertini* and *Paramotopy*, to estimate the size of workspaces, the intersection workspaces, and post-failure grasping workspaces in the case of having two serial robots placed near one another, in two and three dimensions. We also solve the problem of the optimal configuration for these robots for one example of a user-defined objective function. A general algorithm for solving the problem of finding optimal placement and configuration of two such robots was also presented.

By using algebraic geometric methods, we avoid issues such as isolated domains and multiple solutions to inverse kinematics equations. Knowing multiple solutions may be important, especially for obstacle avoidance, where one or more solutions may not be collision free. The homotopy continuation algorithms will not encounter any difficulty in that case, whereas the Jacobian control method will need to be augmented with the specific knowledge from the problem and yet may fail to find all isolated solutions. Our algorithm can be extended to more general cases. For example, it will be relatively straightforward to account for different designs for the two robots (including unequal link lengths) in the objective function. Also, methodological choices in the algorithm, such as the use of homotopy continuation, have been made to make the generalization to higher-dimensional workspaces possible.

The method we present could be complemented by the software in [81]; while their focus is not on failure tolerance, their interactive CAD workspace mapper uses Jacobian method to find singularities and determine workspace boundaries of parallel manipulators, which could be supplemented by homotopy continuation.

It should be noted that the methods of numerical algebraic geometry may be used to compute *complex* positive-dimensional components of the solutions sets of polynomial systems. However, it is nearly impossible to detect positive-dimensional *real* solutions, particularly above dimension one. Therefore, kinematically redundant robots present difficulty for our method. Perhaps we could systematically reduce the underconstrained inverse kinematic system for such a robot to a fully constrained system; however, the curse of dimensionality will likely be problematic.

△
△ △

Bibliography

[1] D. A. Brake, D. J. Bates, V. Putkaradze, and A. A. Maciejewski, "Illustration of numerical algebraic methods for workspace estimation of cooperating robots after joint failure," in *IASTED Tech Conf*, (Pittsburg, PN USA), Nov. 2010.

[2] D. A. Brake, D. Bates, V. Putkaradze, and A. A. Maciejewski, "Workspace estimation of cooperating robots after joint failure using numerical algebraic methods," *Submitted to Adv. Robotics*, 2012.

[3] E. Lee and C. Mavroidis, "Solving the geometric design problem of spatial 3R robot manipulators using polynomial homotopy continuation," *J. Mech. Des.*, vol. 124, pp. 653–661, Dec. 2002.

[4] S. Varedi, H. Daniali, and D. Ganji, "Kinematics of an offset 3-UPU translational parallel manipulator by the homotopy continuation method," *Nonlinear Anal.-Real.*, vol. 10, no. 3, pp. 1767 – 1774, 2009.

[5] L. T. Watson and R. T. Haftka, "Modern homotopy methods in optimization," *Comp. Method Appl. M.*, vol. 74, no. 3, pp. 289 – 305, 1989.

[6] D. J. Bates, J. D. Hauenstein, A. J. Sommese, and C. W. Wampler, "Bertini," 2012.

[7] A. J. Sommese and C. W. Wampler, *The numerical solution to systems of polynomials arising in engineering and science.* Singapore: World Scientific, 2005.

[8] E. Bézout, *Théorie Génerale des Équations Algébriques.* Paris: Imprimerie de P.-D. Pierres, 1779.

[9] D. A. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer, 3rd ed., 2007.

[10] T. Y. Li, T. Sauer, and J. A. Yorke, "The cheater's homotopy: An efficient procedure for solving systems of polynomial equations," *SIAM J. Numer. Anal.*, vol. 26, no. 5,

pp. pp. 1241–1251, 1989.

[11] I. Fotiou, P. Rostalski, B. Sturmfels, and M. Morari, "An algebraic geometry approach to nonlinear parametric optimization in control," in *Am. Control Conf.*, June 2006.

[12] G. Craciun, J. W. Helton, and R. J. Williams, "Homotopy methods for counting reaction network equilibria," *Math. Biosci.*, vol. 216, no. 2, pp. 140 – 149, 2008.

[13] C. Thron, "The secant condition for instability in biochemical feedback controli. the role of cooperativity and saturability," *B. Math. Biol.*, vol. 53, pp. 383–401, 1991. 10.1007/BF02460724.

[14] M. Arcak and E. Sontag, "A passivity-based stability criterion for a class of interconnected systems and applications to biochemical reaction networks," in *Decision and Control, 46th IEEE Conf.*, pp. 4477 –4482, Dec 2007.

[15] B. N. Kholodenko, "Negative feedback and ultrasensitivity can bring about oscillations in the mitogen-activated protein kinase cascades," *Eur. J. Biochem.*, vol. 267, no. 6, pp. 1583–1588, 2000.

[16] S. Y. Shvartsman, M. P. Hagan, A. Yacoub, P. Dent, H. S. Wiley, and D. A. Lauffenburger, "Autocrine loops with positive feedback enable context-dependent cell signaling," *Am. J. Physiol.-Cell Ph.*, vol. 282, no. 3, pp. C545–C559, 2002.

[17] J. M. Ripalda, A. M. Sanchez, A. G. Taboada, A. Rivera, B. Alen, Y. Gonzalez, L. Gonzalez, F. Briones, T. J. Rotter, and G. Balakrishnan, "Relaxation dynamics and residual strain in metamorphic alsb on gaas," *Appl. Phys. Lett.*, vol. 100, no. 1, p. 012103, 2012.

[18] P. Wachulak, M. Marconi, A. Isoyan, L. Urbanski, A. Bartnik, H. Fiedorowicz, and R. Bartels, "Aspects of nanometer scale imaging with extreme ultraviolet (euv) laboratory sources," *Opto-Electron. Rev.*, vol. 20, pp. 1–14, 2012. 10.2478/s11772-012-0008-z.

[19] D. Cohen-Tanugi, A. Akey, and N. Yao, "Ultralow superharmonic resonance for functional nanowires," *Nano. Lett.*, vol. 10, no. 3, pp. 852–859, 2010. PMID: 20155914.

[20] M. Grisham, G. Vaschenko, C. S. Menoni, J. J. Rocca, Y. P. Pershyn, E. N. Zubarev, D. L. Voronov, V. A. Sevryukova, V. V. Kondratenko, A. V. Vinogradov, and I. A. Artioukov, "Damage to extreme-ultraviolet Sc/Si multilayer mirrors exposed to intense 46.9-nm laser pulses," *Opt. Lett.*, vol. 29, pp. 620–622, Mar 2004.

[21] G. Vaschenko, F. Brizuela, C. Brewer, M. Grisham, H. Mancini, C. S. Menoni, M. C. Marconi, J. J. Rocca, W. Chao, J. A. Liddle, E. H. Anderson, D. T. Attwood, A. V. Vinogradov, I. A. Artioukov, Y. P. Pershyn, and V. V. Kondratenko, "Nanoimaging with a compact extreme-ultraviolet laser," *Opt. Lett.*, vol. 30, pp. 2095–2097, Aug 2005.

[22] M. Capeluto, G. Vaschenko, M. Grisham, M. Marconi, S. Luduena, L. Pietrasanta, Y. Lu, B. Parkinson, C. Menoni, and J. Rocca, "Nanopatterning with interferometric lithography using a compact lambda;=46.9-nm laser," *IEEE Trans. Nanotechnol.*, vol. 5, pp. 3 – 7, jan. 2006.

[23] J. Verd, A. Uranga, G. Abadal, J. Teva, F. Torres, F. Perez-Murano, J. Fraxedas, J. Esteve, and N. Barniol, "Monolithic mass sensor fabricated using a conventional technology with attogram resolution in air conditions," *Appl. Phys. Lett.*, vol. 91, no. 1, p. 013501, 2007.

[24] A. J. Sievers and S. Takeno, "Intrinsic localized modes in anharmonic crystals," *Phys. Rev. Lett.*, vol. 61, pp. 970–973, Aug 1988.

[25] J. B. Page, "Asymptotic solutions for localized vibrational modes in strongly anharmonic periodic systems," *Phys. Rev. B*, vol. 41, pp. 7835–7838, Apr 1990.

[26] G. Huang, Z. Shi, and Z. Xu, "Assymetric intrinsic localized modes in a homogeneous lattice with cubic and quartic anhamonicity," *Phys. Rev. B*, vol. 47, June 1993.

[27] P. Kevrekidis, A. Bishop, and K. O. Rasmussen, "Twisted localized modes," *Phys. Rev. E*, vol. 63, no. 3, 2001.

[28] P. G. Kevrekidis and V. V. Konotop, "Bright compact breathers," *Phys. Rev. E*, vol. 65, p. 066614, Jun 2002.

[29] M. Sato, B. E. Hubbard, L. Q. English, A. J. Sievers, B. Ilic, D. A. Czaplewski, and H. G. Craighead, "Study of intrinsic localized vibrational modes in micromechanical oscillator arrays," *Chaos*, vol. 13, p. 702, May 2003.

[30] M. Sato, B. E. Hubbard, A. J. Sievers, B. Ilic, D. A. Czaplewski, and H. G. Craighead, "Observation of locked intrinsic localized vibrational modes in a micromechanical oscillator array," *Phys. Rev. Lett.*, vol. 90, p. 044102, Jan 2003.

[31] M. Sato, B. E. Hubbard, and A. J. Sievers, "Colloquium: Nonlinear energy localization and its manipulation in micromechanical oscillator arrays," *Rev. Mod. Phys.*, vol. 78, pp. 137–157, Jan 2006.

[32] M. Sato and A. J. Sievers, "Visualizing intrinsic localized modes with a nonlinear micromechanical array," *Low Temp. Phys.*, vol. 34, no. 7, pp. 543–548, 2008.

[33] M. Kimura and T. Hikihara, "Capture and release of traveling intrinsic localized mode in coupled cantilever array," *Chaos*, vol. 19, no. 1, p. 13138, 2009.

[34] M. Kimura and T. Hikihara, "Coupled cantilever array with tunable on-site nonlinearity and observation of localized oscillations," *Phys. Lett. A*, vol. 373, no. 14, pp. 1257 – 1260, 2009.

[35] M. Sato and A. J. Sievers, "Direct observation of the discrete character of intrinsic localized modes in an antiferromagnet," *Nature*, vol. 432, pp. 486–488, 11 2004.

[36] V. M. Burlakov, S. A. Kiselev, and V. N. Pyrkov, "Computer simulation of intrinsic localized modes in one-dimensional and two-dimensional anharmonic lattices," *Phys. Rev. B*, vol. 42, no. 8, pp. 4921–4927, 1990.

[37] P. G. Kevrekidis, S. V. Dmitriev, S. Takeno, A. R. Bishop, and E. C. Aifantis, "Rich example of geometrically induced nonlinearity: From rotobreathers and kinks to moving localized modes and resonant energy transfer," *Phys. Rev. E*, vol. 70, p. 066627, Dec 2004.

[38] V. Hizhnyakov, A. Shelkan, M. Klopov, A. J. Sievers, and M. Haas, “Intrinsic localized modes and trapped phonons in crystal lattices,” *J. Phys: Conf. Series*, vol. 92, no. 1, 2007.

[39] M. Sato and A. J. Sievers, “Visualizing intrinsic localized modes with a nonlinear micromechanical array,” *Fizika Nizkikh Temperatur*, vol. 34, pp. 687–694, June 2008.

[40] S. D. Hersee, X. Y. Sun, X. Wang, M. N. Fairchild, J. Liang, and J. Xu, “Visualizing intrinsic localized modes with a nonlinear micromechanical array,” *J. Appl. Phys.*, vol. 97, no. 12, p. 124308, 2005.

[41] B. A. Malomed, P. G. Kevrekidis, D. J. Frantzeskakis, H. E. Nistazakis, and A. N. Yannacopoulos, “One- and two-dimensional solitons in second-harmonic-generating lattices,” *Phys. Rev. E*, vol. 65, p. 056606, Apr 2002.

[42] R. B. Thakur, L. Q. English, and A. J. Sievers, “Driven intrinsic localized modes in a coupled pendulum array,” *J. Phys. D Appl. Phys.*, vol. 41, no. 1, p. 015503, 2008.

[43] L. Q. English, F. Palmero, A. J. Sievers, P. G. Kevrekidis, and D. H. Barnak, “Traveling and stationary intrinsic localized modes and their spatial control in electrical lattices,” *Phys. Rev. E*, vol. 81, p. 046605, Apr 2010.

[44] B. S. Dhillon, A. R. M. Fashandi, and K. L. Liu, “Robot systems reliability and safety: A review,” *J. Qaul. Maint. Eng.*, vol. 8, no. 3, pp. 170–212, 2002.

[45] “Honda worldwide — ASIMO.” `http://world.honda.com/ASIMO/`. Accessed: 31/10/2011.

[46] “Lockheed martin MULE program completes key review, begins work on final system design.” `http://www.lockheedmartin.com/news/press_releases/2008/MFC-022708-MULEProgramCompletesKeyReview.html`. Accessed: 31/10/2011.

[47] “Micromagic robotic systems lab.” `http://hexapodrobot.com/index.html`. Accessed: 31/10/2011.

[48] J. Denavit and R. Hartenberg, "A kinematic notation for lower-pair mechanisms based on matrices," *Trans. ASME J. Appl. Mech.*, vol. 23, p. 215221, 1955.

[49] P. I. Corke and B. Armstrong-Helouvry, "A search for consensus among model parameters reported for the PUMA 560 robot," in *IEEE Int. Conf. Robot. Automat.*, pp. 1608 –1613 vol.2, May 1994.

[50] R. G. Roberts and A. A. Maciejewski, "Repeatable generalized inverse control strategies for kinematically redundant manipulators," *IEEE Trans. Autom. Cont.*, vol. 38, pp. 689–699, May 1993.

[51] R. G. Roberts and A. A. Maciejewski, "Singularities, stable surfaces, and the repeatable behavior of kinematically redundant manipulators," *Int. J. of Robot. Res.*, vol. 13, pp. 70–81, Feb 1994.

[52] A. A. Maciejewski and J. M. Reagin, "A parallel algorithm and architecture for the control of kinematically redundant manipulators," *IEEE Trans. Robot. Automat.*, vol. 10, pp. 404–414, Aug 1994.

[53] Q. Xue, A. A. Maciejewski, and P. C.-Y. Sheu, "Determining the collision-free joint space graph for two cooperating robots," *IEEE Trans. Syst. Man Cyb.*, vol. 23, no. 1, pp. 285–194, 1993.

[54] A. A. Maciejewski and J. J. Fox, "Path planning and the topology of configuration space," *IEEE Trans. Robot. Autom.*, vol. 9, pp. 444–456, August 1993.

[55] I. Ebert-uphoff and G. S. Chirikjian, "Efficient workspace generation for binary manipulators with many actuators," *J. Robot. Systems.*, vol. 12, no. 6, p. 383400, 1995.

[56] M. Ji and N. Sarkar, "Supervisory fault adaptive control of a mobile robot and its application in sensor-fault accommodation," *IEEE Trans. Robot.*, vol. 23, no. 1, pp. 174–178, 2007.

[57] D. Brambilla, L. Capisani, A. Ferrara, and P. Pisu, "Fault detection for robot manipulators via second-order sliding modes," *IEEE Trans. Ind. Electron.*, vol. 55, no. 11,

pp. 3954–3963, 2008.

[58] A. De Luca and L. Ferrajoli, “A modified Newton-Euler method for dynamic computations in robot fault detection and control,” in *IEEE Int. Conf. Robot. Automat.*, pp. 3359–3364, May 2009.

[59] Y. Yi, J. E. McInroy, and Y. Chen, “Fault tolerance of parallel manipulators using task space and kinematic redundancy,” *IEEE Trans. Robot.*, vol. 22, no. 5, pp. 1017–1021, 2006.

[60] J. E. McInroy, J. F. O’Brien, and G. W. Neat, “Precise, fault-tolerant pointing using a stewart platform,” *IEEE/ASME Trans. Mechatronics*, vol. 4, no. 1, pp. 91–95, 1999.

[61] K. N. Groom, A. A. Maciejewski, and V. Balakrishnan, “Real-time failure-tolerant control of kinematically redundant manipulators,” *IEEE Trans. Robot. Autom.*, vol. 15, no. 6, pp. 1109–1116, 1999.

[62] A. A. Maciejewski, “Fault tolerant properties of kinematically redundant manipulators,” in *Proc. IEEE Int. Conf. Robot. Automat.*, (Cincinatti, OH, USA), pp. 638–642, 1990.

[63] R. G. Roberts and A. A. Maciejewski, “A local measure of fault tolerance for kinematically redundant manipulators,” *IEEE Trans. Robot. Automat.*, vol. 12, no. 4, pp. 543–552, 1996.

[64] Y. Chen, J. E. McInroy, and Y. Yi, “Optimal, fault-tolerant mappings to achieve secondary goals without compromising primary performance,” *IEEE Trans. Robot.*, vol. 19, no. 4, pp. 680–691, 2003.

[65] C. L. Lewis and A. A. Maciejewski, “Fault tolerant operation of kinematically redundant manipulators for locked joint failures,” *IEEE Trans. Robot. Automat.*, vol. 13, pp. 622–629, Aug. 1997.

[66] J. D. English and A. A. Maciejewski, “Fault tolerance for kinematically redundant manipulators: Anticipating free-swinging joint failures,” *IEEE Trans. Robot. Autom.*, vol. 14, no. 4, pp. 566–575, 1998.

[67] C. J. J. Paredis and P. K. Khosla, "Designing fault-tolerant manipulators: How many degrees of freedom?," *Int. J. Robot. Res.*, vol. 15, pp. 611–628, Dec. 1996.

[68] M. Hassan and L. Notash, "Optimizing fault tolerance to joint jam in the design of parallel robot manipulators," *Mech. Mach. Theory*, vol. 42, pp. 1401–1407, 2007.

[69] E. C. Wu, J. C. Hwang, and J. T. Chladek, "Fault-tolerant joint development for the space shuttle remote manipulator system: Analysis and experiment," *IEEE Trans. Robot. Autom.*, vol. 9, no. 5, pp. 675–684, 1993.

[70] F. Aghili and K. Parsa, "A reconfigurable robot with lockable cylindrical joints," *IEEE Trans. Robot.*, vol. 25, no. 4, pp. 785–797, 2009.

[71] C. Carreras and I. D. Walker, "Interval methods for fault-tree analysis in robotics," *IEEE Trans. Robot. Automat.*, vol. 50, no. 1, pp. 3–11, 2001.

[72] E. C. Wu, J. C. Hwang, and J. T. Chladek, "Fault-tolerant joint development for the space shuttle remote manipulator system: Analysis and experiment," *IEEE Trans. Robot. Autom.*, vol. 9, no. 5, pp. 675–684, 1993.

[73] A. Sommese, J. Verschelde, and C. W. Wampler, "Advances in polynomial continuation for solving problems in kinematics," *J. Mech. Des.*, vol. 126, no. 2, pp. 262–268, 2004.

[74] C. W. Wampler and A. P. Morgan, "Solving the kinematics of general 6R manipulators using polynomial continuation," in *Robotics: Applied Mathematics and Computational Aspects* (K. Warwick, ed.), pp. 57–69, Oxford: Clarendon Press, 1993.

[75] C. W. Wampler, A. P. Morgan, and A. J. Sommese, "Complete solution of the nine-point path synthesis problem for four-bar linkages," *J. Mech. Des.*, vol. 114, pp. 153–159, Mar. 1992.

[76] C. W. Wampler, J. D. Hauenstein, and A. J. Sommese, "Mechanism mobility and a local dimension test," *Mech. Mach. Theory*, vol. 46, no. 9, pp. 1193–1206, 2011.

[77] J. D. English and A. A. Maciejewski, "Measuring and reducing the Euclidean-space measures of robotic joint failures," *IEEE Trans. Robot. Automat.*, vol. 16, no. 1, pp. 20–28, 2000.

[78] J. D. English and A. A. Maciejewski, "Failure tolerance through active braking: A kinematic approach," *Int. J. Rob. Res.*, vol. 20, no. 4, pp. 287–299, 2001.

[79] E. L. Allgower and K. Georg, *Numerical continuation methods, an introduction*. Berlin: Springer-Verlag, 1990.

[80] D. J. Bates, D. A. Brake, and M. Niemerg, "Paramotopy: Parallel parameter homotopy via Bertini," *in preparation*, 2012.

[81] E. Macho, C. Pinto, E. Amezua, and A. Hernndez, "Software tool to compute, analyze and visualize workspaces of parallel kinematics robots," *Adv. Robotics*, vol. 25, pp. 675–698(23), 2011.

[82] B. Buchberger, "A theoretical basis for the reduction of polynomials to canonical forms," *SIGSAM Bull.*, vol. 10, pp. 19–29, August 1976.

Appendix A

3D PUMA Equations

The inverse kinematic equations for the first three joints of the PUMA robot are, as used in this paper are $f_i = 0$ for $i = 1, \ldots, 12$ with f_i defined as:

$$\begin{aligned}
f_1 =& 0.4318c_1c_2 - 0.2435s_1 + 0.0934as_1 + 0.4331c_1c_2s_3+\\
& 0.4331c_1c_3s_2 + 0.0203ac_1s_2s_3 - 0.0203ac_1c_2c_3-\\
& (0.4318c_4c_5 - 0.1501s_4 - 0.0203c_4c_5c_6+\\
& 0.4331c_4c_5s_6 + 0.4331c_4c_6s_5 + 0.0203c_4s_5s_6 + \delta)\\
f_2 =& 0.2435c_1 - 0.0934ac_1 + 0.4318c_2s_1 + 0.4331c_2s_1s_3+\\
& 0.4331c_3s_1s_2 + 0.0203as_1s_2s_3 - 0.0203ac_2c_3s_1-\\
& (0.1501c_4 + 0.4318c_5s_4 - 0.0203c_5c_6s_4+\\
& 0.4331c_5s_4s_6 + 0.4331c_6s_4s_5 + 0.0203s_4s_5s_6)\\
f_3 =& 0.4331c_2c_3 - 0.4318s_2 - 0.4331s_2s_3 + 0.0203ac_2s_3+\\
& 0.0203ac_3s_2 - (0.4331c_5c_6 - 0.4318s_5 + 0.0203c_5s_6+\\
& 0.0203c_6s_5 - 0.4331s_5s_6)\\
f_4 =& 0.4318c_1c_2 - 0.1501s_1 - 0.0203c_1c_2c_3+\\
& 0.4331c_1c_2s_3 + 0.4331c_1c_3s_2 + 0.0203c_1s_2s_3 - x
\end{aligned}$$

$$f_5 = 0.1501c_1 + 0.4318c_2s_1 - 0.0203c_2c_3s_1 +$$

$$0.4331c_2s_1s_3 + 0.4331c_3s_1s_2 + 0.0203s_1s_2s_3 - y$$

$$f_6 = 0.4331c_2c_3 - 0.4318s_2 + 0.0203c_2s_3 +$$

$$0.0203c_3s_2 - 0.4331s_2s_3 - z$$

$$f_7 = s_1^2 + c_1^2 - 1$$

$$f_8 = s_2^2 + c_2^2 - 1$$

$$f_9 = s_3^2 + c_3^2 - 1$$

$$f_{10} = s_4^2 + c_4^2 - 1$$

$$f_{11} = s_5^2 + c_5^2 - 1$$

$$f_{12} = s_6^2 + c_6^2 - 1$$

The coefficients appearing in these equations are 4 digit approximations of exact numbers coming from the DH parameters for the PUMA. The terms x, y, z, δ, a appear in the equations as parameters for *Paramotopy.* Functions $f_1 - f_6$ couple the positions of the two cooperating robots, and functions $f_7 - f_{12}$ are Pythagorean identities tying cosines and sines together.

Appendix B

Symbolic Computational Algebraic Geometry

The two main categories of methods employed in modern Computational Algebraic Geometry, for computing varieties as intersections of algebraic equations, are

- symbolic, and
- numeric.

Each has found appropriate use in different areas of application. Briefly, symbolic methods, such as Gröbner Bases, are fast and exact, being well-suited for small problems. Numerical methods, such as homotopy continuation, are better for large, complex problems, and when numerical approximations to solutions are acceptable in place of exact solutions.

In this section we will outline a symbolic method for determining the generators of an ideal. This content is referenced from [9].

We begin by setting up a bit of notation. In this section, and in fact this entire chapter, we are working over the ring of polynomials generated by taking our favorite field $\mathbb{F}$, and adjoining however many variables we want, say n variables. We denote the variables by $x_1, \dots, x_n$, so that our ring is $\mathbb{F}[x_1, \dots, x_n]$. The field $\mathbb{F}$ we will be working over typically will be $\mathbb{C}$. As the word 'polynomial' is typically used for a function of a single variable, we use the term multinomial.

Take α to be a *multiindex*. That is, α is a vector of nonnegative integers ($\alpha \in \mathbb{Z}^n_{\geq 0}$), denoting the powers of the variables appearing in a terms of a multinomial. Note that α consists of integers because we are working in the world of multinomials, although in principle, the entries in a multiindex α can be arbitrary numbers from $\mathbb{R}^+$.

As an example, suppose we have three variables, and a term in the ring of multinomials $h = cx_1^i x_2^j x_3^k$. This term has multiindex $\alpha = \langle i, j, k \rangle$, and we can write $h = cx^\alpha$.

We will additionally need the concept of a monomial ordering in order to be able to write our definitions. There are several term orderings generally used:

- **Lexicographic (lex)**

$$\alpha <_{\text{lex}} \beta \quad \Leftrightarrow \quad \text{leftmost entry of } \alpha - \beta < 0$$

- **Graded Lexicographic (grlex)**

$$\begin{aligned} \alpha <_{\text{grlex}} \beta \quad \Leftrightarrow \quad & |\alpha| < |\beta|, \text{ or} \\ & \text{leftmost entry of } \alpha - \beta > 0 \quad \& \quad |\alpha| = |\beta| \end{aligned}$$

- **Graded Reverse Lexicographic (grevlex)**

$$\begin{aligned} \alpha <_{\text{grevlex}} \beta \quad \Leftrightarrow \quad & |\alpha| < |\beta|, \text{ or} \\ & \alpha <_{\text{lex}} \beta \quad \& \quad |\alpha| = |\beta| \end{aligned}$$

These various monomial orderings give a total order to the set of multinomials. That is, we can now define the *leading term* of a multinomial.

Definition 2. *Take $f = c_j x^{\alpha_j}$. The* ***leading term*** *of a multinomial f is given by* $\text{LT}(f) = cx^{\alpha_m}$, *where α_m is the greatest multiindex according to the chosen term order.*

Definition 3. *The* ***leading terms*** *of an ideal I* $\text{LT}(I)$ *are given by*

$$\text{LT}(I) = \{cx^\alpha : \text{LT}(f) = cx^\alpha \text{ for some } f \in I\}$$

Definition 4. *The* ***leading term ideal*** *is the ideal generated by the leading terms of the ideal, written as*

$$\langle \text{LT}(I) \rangle$$

Note that if $I = \langle f_1, \dots, f_k \rangle$ then the set of monomials $\{\mathrm{LT}(f_1), \dots, \mathrm{LT}(f_k)\}$ does not necessarily generate $\langle \mathrm{LT}(I) \rangle$. In fact, it is this inequality that gives rise to the definition of the Gröbner Basis.

Definition 5. *Under a monomial order, $G = \{g_1, \dots, g_t\}$ is a **Gröbner Basis** for the ideal I if $\langle \mathrm{LT}(g_1), \dots, \mathrm{LT}(g_t) \rangle = \langle \mathrm{LT}(I) \rangle$.*

The Gröbner basis of an ideal has a few nice properties:

- We have uniqueness of remainder when dividing by the Gröbner Basis G (for how to compute the remainder, see Definition 8),
- Every ideal has a Gröbner Basis G.

This yields an ideal membership test; that is, we can see from G whether a candidate multinomial f is in an ideal I: $f \in I \Leftrightarrow$ the remainder of f upon division by G is 0. This is fine, but first we must find G, given a set of generators. This is done by using *Buchberger's Algorithm*, which requires yet another pair of definitions.

Definition 6. *The **multidegree** of a multinomial f is the vector of highest degrees appearing in f, for each variable.*

Example. *The multidegree of $f = x_1^2 x_2^3 + x_1 x_2^2 + x_3$ is $\langle 2, 3, 1 \rangle$.*

Definition 7. *Define γ to be a multiindex given by setting the ith entry $\gamma_i = \max(\mathrm{multideg}(f)_i\,, \mathrm{multide}$ The **s-polynomial** $S(f, g)$ of two multinomials is given by*

$$S(f,g) = \frac{x^\gamma}{\mathrm{LT}(f)} f - \frac{x^\gamma}{\mathrm{LT}(g)} g$$

Definition 8. *To compute the **remainder** r of a polynomial f with respect to a set of polynomials g_i, we do the following:*

(1) *Let a_i be the leading term of g_i. Let $f_0 = f$, and $j = 0$.*

(2) *For the smallest i such that a_i divides some term of f, let h be the largest term (with respect to the monomial ordering) of g such that a_i divides h.*

(3) *Increment $j = j + 1$. Let $f_j = f_{j-1} - (h/a_i) \cdot g_i$.*

(4) *Continue steps 2 and 3 until no term in f_j is divisible by any leading term, and set $r = f_j$.*

Now that we have the necessary tools, we present Buchberger's Algorithm (BA) for computing Gröbner Bases [82].

Algorithm 5 Buchberger's Algorithm

```
1: procedure BBA(A set of generators $\{f_i\}$ for an ideal, $I = \langle f_1, \ldots, f_k \rangle$)
2:     Set $G = \{f_1, \ldots, f_k\}$
3:     while NUMNEW $\neq 0$ do
4:         NUMNEW $= 0$
5:         Form all S-polynomials $Si, j = S(g_i, g_j)$ using Definition 7
6:         Compute all remainders $R_{i,j} = R(S_{i,j}, G)$, using Definition 8
7:         for $R_{i,j} \neq 0$ do
8:             NUMNEW = NUMNEW+1
9:             $G = G \cup R_{ij}$
10:        end for
11:    end while
12:    Return $G$, a Gröbner Basis for $I$.
13: end procedure
```

This algorithm has been proven both correct and convergent. Hence, we have a symbolic solution to the ideal membership problem: use Buchberger to compute Gröbner basis, then compute the remainder of the polynomial in question, when divided by the new basis.

To illustrate the above definitions and the Gröbner Basis computation, consider the following example.

Example. *Let $I = \langle x^2 + xy,\, y^3 + xy \rangle$. We will show that the two polynomials generating I form a Gröbner Basis, by using Buchberger's Algorithm. We use the* `grevlex` *monomial ordering.*

We must choose a labeling of the polynomials. Let

$$\begin{aligned} f_1 =& x^2 + xy \\ f_2 =& y^3 + xy \end{aligned}$$

Then we have the leading terms

$$\begin{aligned} a_1 =& x^2 \\ a_2 =& y^3, \end{aligned}$$

and the multidegrees

$$\begin{aligned} \gamma_1 =& < 2, 1 > \\ \gamma_2 =& < 1, 3 > . \end{aligned}$$

For step 1 of Algorithm 5, we set $G = \{f_1, f_2\}$. *We must next form the s-polynomial of* f_1, f_2.

$$\begin{aligned} S_{1,2} =& \frac{x^2y^3}{x^2}\left(x^2 + xy\right) - \frac{x^2y^3}{y^3}\left(y^3 + xy\right) \\ =& xy^4 - x^3y. \end{aligned}$$

There is only the one s-polynomial to consider adding to the Gröbner basis at this point. We must compute the remainder $R_{1,2}$ *of* $S_{1,2}$ *with respect to* G. *Both of the two monomials in* $R_{1,2}$ *are divisible by one of* a_i. *Denote the successive remainders by* r_k. *We first perform* *'division' by a_1:*

$$\begin{aligned}w_0 =& R_{1,2} = xy^4 + x^3\\ w_1 =& w_0 - (xy)\cdot\left(x^2 + xy\right)\\ =& xy^4 - x^2y^2\end{aligned}$$

Notice that the degree of w_1 is less than w_0 using the monomial ordering. Repeating similar calculations, we get the sequence

$$\begin{aligned}w_2 =& xy^4 + xy^3\\ w_3 =& xy^3 - x^2y^2\\ w_4 =& 0\end{aligned}$$

The sequence terminates with remainder $R_{1,2} = 0$. Hence, $S_{1,2}$ does not need to be added to G. Furthermore, since this was the only s-polynomial we formed, G as it started is a Gröbner Basis.
(>^.^)>

Gröbner basis methods and Buchberger's algorithm are fairly standard now. There is a Gröbner package in Maple, GAP, Singular, Macaulay2, and others. For small systems, with few functions and few terms, Algorithm 5 runs fast. However, when the system becomes large or complex, the runtime grows combinatorially. Namely, the number of s-polynomials to be checked for inclusion in G is the major problem. In this case, we turn to numerical methods.

Appendix C

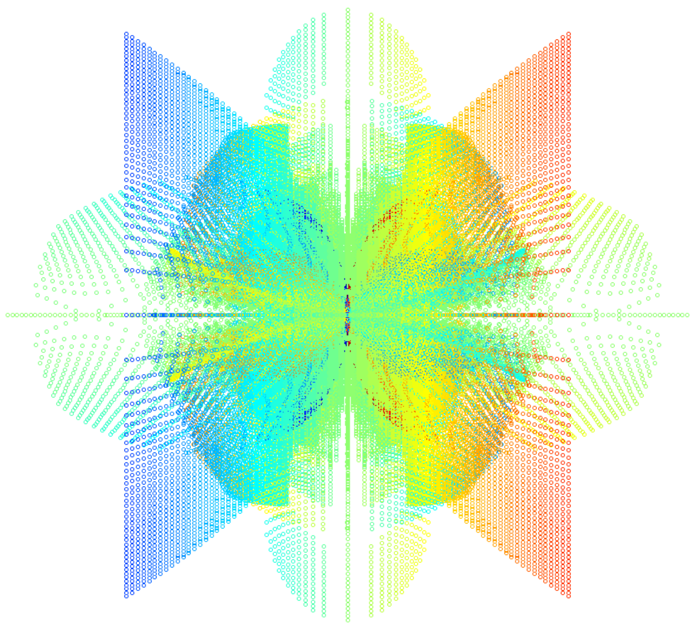
Paramotopy User's Manual

Paramotopy

Parallel Parameter Homotopy

via Bertini

Daniel Brake and Matt Niemerg



Manual by
Daniel Brake
Colorado State University
Mathematics

Thursday 1st November, 2012

1. Introduction

The Paramotopy program is a compiled linux/unix wrapper around Bertini which permits rapid parallel solving of parameterized polynomial systems. It consists of two executable programs: `Paramotopy`, and `Step2`, and further depends on having a copy of the parallel version of Bertini. `Paramotopy` is called from the command line, and in turn calls `Bertini` and `step2`.

Briefly, homotopy continuation is the tracking of solutions from one system to another through continuous deformation; a simplified example appears in Figure 30. Using a combination of prediction and correction, Bertini follows solutions through such a deformation. First, it obtains the solutions to the initial system. Taking discrete steps in complex-valued time, each solution path is tracked individually. If suitable options are chosen in the configuration of the run, precision will be automatically adjusted to compensate for loss of accuracy near system singularities due to poorly conditioned matrices. Additionally, Bertini has various "endgame" schemes for bringing the paths to successful completion, as well as determining whether paths crossed or other problems happened along the way. Nearly all these options may be accessed through Paramotopy. For a thorough description of the Bertini program, including many capabilities not used within Paramotopy, see the Bertini User's Manual.

Paramotopy is implemented to make use of both the basic zero-dimensional solve Bertini performs by default, as well as exploiting the `userhomotopy` mode. An illustration of the scheme is in Figure 31. First, Paramotopy performs what we define here to be a 'Step1' run. Bertini finds all zero-dimensional solutions to the system supplied to it, for a set of complex parameter values randomly determined; for this solve, there will likely be superfluous paths. Second, Paramotopy tracks all meaningful solutions found in Step1 (from `nonsingular_solutions`) to each parameter point at which the user wishes to solve; this is called ‘Step2’. Performing the second step in this way will eliminate the extra paths from Step1.

With respect to the soundness and reliability of this method, theory dictates that off a set of measure zero, an algebraic system will have the same number of (complex) roots throughout its parameter space. Therefore, we are virtually guaranteed that for a *randomly* chosen point in parameter space, we will find the full generic number of solutions. Then for Step2, as we track from the random point to the specific points at which we wish to solve, we will find all the solutions.

Paramotopy uses a specially compiled library of Bertini, MPICH2 for process distribution, OpenMP for accurate timing measurements, TinyXML for preference and information storage, and Boost for filesystem operations. System requirements and compilation information may be found in Section 2. In Section 3 is presented the format for the input files, as well as information on configuring and using Paramotopy. Specific descriptions of options are given in Section 4, and data output is described in Section 5. Coarse troubleshooting tips are in Section 6.

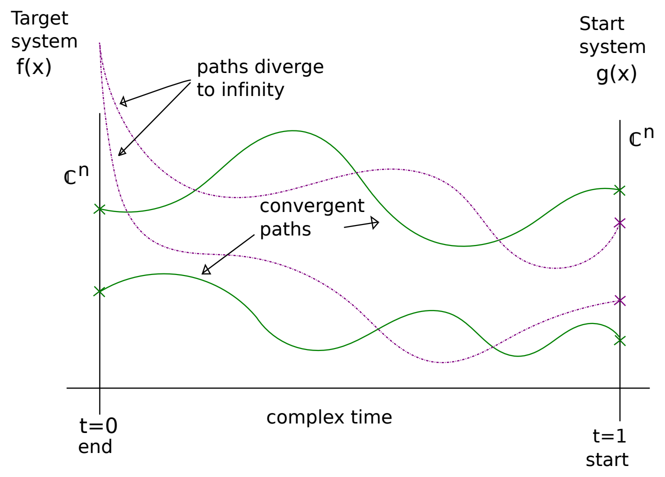


Figure 30. Generic Homotopy Continuation

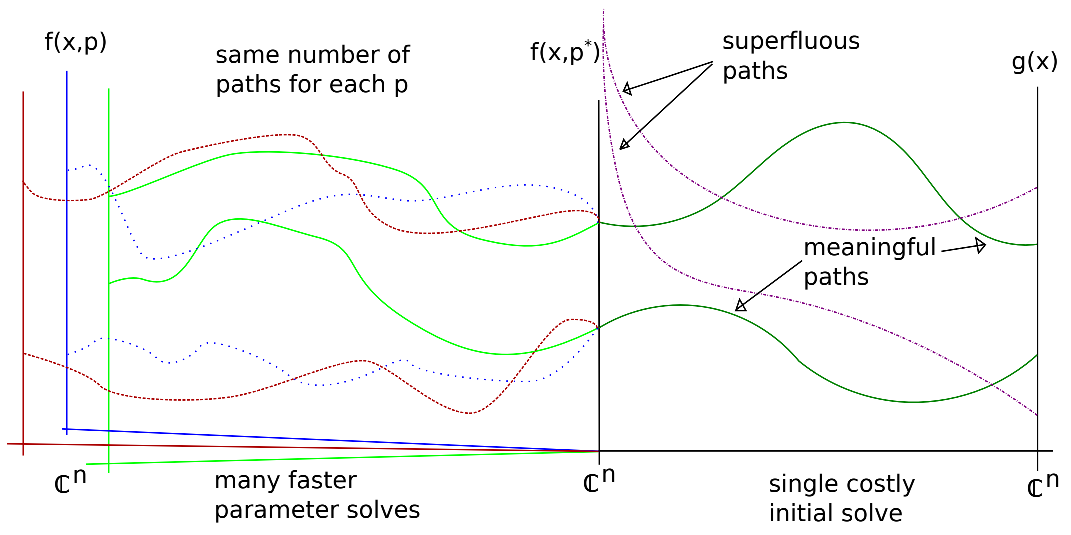


FIGURE 31. Parameter Homotopy, with initial 'Step1' run.

1.1. Contact.

For assistance with Paramotopy, please contact Daniel Brake at `danielthebrake@gmail.com`.

For help with Bertini, contact Daniel Brake, Dan Bates

`bates@math.colostate.edu`,

or one of the other Bertini authors.

ACKNOWLEDGEMENTS.

- This research utilized the CSU ISTeC Cray HPC System supported by NSF Grant CNS-0923386.
- This material is based upon work supported by the National Science Foundation under Grants No. DMS-1025564 and DMS-1115668.

1.2. Licencse.

The authors have not yet decided on the license for Paramotopy. It will be free, open source software with sufficient protection against commercialization as to protect against code theft and legal boggarting.

DISCLAIMER. Paramotopy and all related code, executables, and other material are offered without warranty, for any purpose, implied or explicit.

2. Getting Started

2.1. Compilation and Installation. Paramotopy has the following library dependencies:

- Tinyxml (lies internally to the program's folders, as per the license agreement).
- MPICH2,
- Boost – regex, system, filesystem (for file manipulation)
- OpenMP (for the most accurate timing available).

In addition, Paramotopy relies on a suitably compiled version of Bertini, which has the following dependencies:

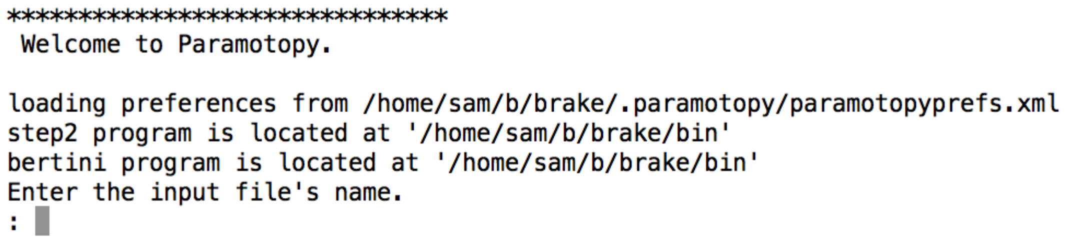
- mpfr
- gmp
- Bison
- Flex
- MPICH2

When Bertini is compiled for use with Paramotopy, it must be compiled in two ways: into an executable, and a library. When compiled into the executable, it must have the flag `-D_HAVE_MPI` (for use with Paramotopy). In contrast, the library *must not* have the `-D_HAVE_MPI` flag. In addition, we redefine `main()` to be `bertini_main` via the command `-Dmain=bertini_main`. Compilation is a little different of every machine, of course, and a future milestone is to use something like `./configure` to handle setting everything up properly for start-to-finish with minimal user input.

As it stands now, Paramotopy is compiled using a Makefile. Various options are available in the makefile:

- `MACHINE`: choose the name of the machine you are compiling on. The makefile contains a section for each machine, and sets the necessary variables suitably. To compile on a new machine, create a new section, with all the necessary options.
- `TIMING`: choose `YES` or `NO`. Conditional compilation using `#ifdef` in the code turns on or off timing statements for Step2.
- `OPT`: Sets available compiler options, such as the `-O` optimization flag, and compiler warning levels. These are left user-set, as every compiler has different options.
- `VERBOSE`: Chooses whether Step2 ought to be compiled with `#ifdefs` for extra output to the screen.

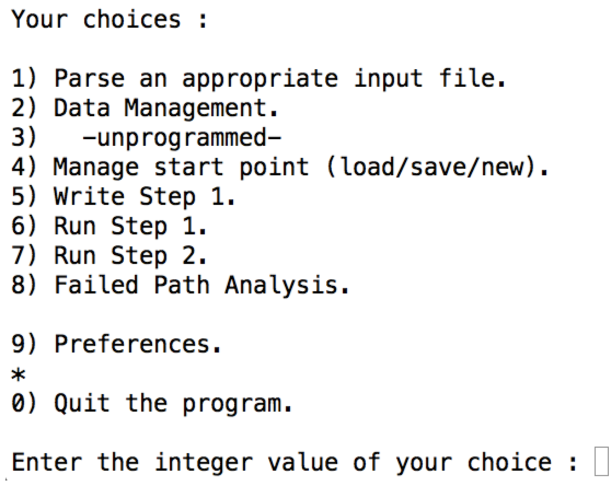
Simply issuing the `make` command should be enough to compile the programs once initial configuration is completed.



```
********************************
 Welcome to Paramotopy.

loading preferences from /home/sam/b/brake/.paramotopy/paramotopyprefs.xml
step2 program is located at '/home/sam/b/brake/bin'
bertini program is located at '/home/sam/b/brake/bin'
Enter the input file's name.
: 
```

FIGURE 32. The welcome screen. If you do not supply the filename as the first argument to Paramotopy from the command line, it will immediately prompt you for the name. You cannot use the program without an input file.

2.2. Using Paramotopy.



```
Your choices :

1) Parse an appropriate input file.
2) Data Management.
3)   -unprogrammed-
4) Manage start point (load/save/new).
5) Write Step 1.
6) Run Step 1.
7) Run Step 2.
8) Failed Path Analysis.

9) Preferences.
*
0) Quit the program.

Enter the integer value of your choice :
```

FIGURE 33. Main menu, describing the available choices. Use numeric input, or whatever it prompts for. If you just loaded the program, it will describe what directory it is working in.

3. Input Files

The input file format for Paramotopy is fairly simple. Extra white space on a single line is acceptable, but there must be no extra blank lines. Illustrative examples appear in Files C.1,C.2,C.3. The input file consists of:

(1) Declare the numbers of: functions (`a`), variable groups (`b`), parameters (`c`), and constants (`d`), in one line, separated by spaces; e.g.

```
a b c d
```

(2) Functions declaration, without any name declarations or terminating semi-colons.

(3) Variable groups come next, with each group getting its own line; variables are comma separated, and there is *no terminating character*.

(4) Constant declaration, with the word `constant` appearing before a comma separated, semi-colon terminated line containing the constant names. Each constant gets its own line, with `name = value;` being the format.

(5) Finally, the type of solve is indicated:

- 0 indicates the computer will supply a mesh. The final lines tell the name of each parameter, starting point of discretization, ending point of discretization, and number of discretization points. The format for a line of parameter declaration is

  `name a b c d e`,

  where startpoint = `a`+`b`i, endpoint = `c`+`d`i, and `e` indicates the number of discretization points, which must be an integer. See File C.2.
- 1 asserts the user supplies a text file containing parameter values; the next line is the name of the file, and the final lines simply indicate the parameter names. See File C.3.

```
nfunct nvargrp nconst nparam    <- all four must be present
function1
function2                       <- no terminating semicolon
...                             <- do not name functions
function n                      <- no '=' sign
vargroup1
vargroup2
constant c1,c2,...,ck;          <- note semicolon, omit if nconst=0
c1 = 0.1234;                    <- semicolon and equal sign
c2 = 0 + 9*I;                   <- capital 'I' for sqrt(-1)
...
ck = 1-1*I;
0                  or     1            <- choose mesh or userdefined
                          filename     <- only if userdefined
p1   0 0 1 0 10    or     p1           <- declare mesh discretization
p2   0 0 1 0 64    or     p2                or parameter name
...
pj   0 0 1 0 13    or     pj
```

File C.1. Generic Paramotopy input file.

Paramotopy has minimal error correction in this portion of the program, so an error in an input file is likely to cause a fairly benign (will not corrupt data) program crash. If the program is able to parse the input file without errors, it will display information to the screen and the user may check everything has been imported correctly. More syntax checking will be added later.

```
 1 7 1 2 21
 2 Fa−k1*a*b+k2*c−k4*a*f+k5*f−Da*a
 3 Fb−k1*a*b+k2*c+k8*g−k9*b*f−Db*b
 4 Fc+k2*a*b−k2*c−k3*c−k6*c*e+k7*g−Dc*c
 5 Fd+k3*c−Dd*d
 6 Fe−k4*a*e+k5*f−k6*c*e+k7*g−De*e
 7 Ff+k4*a*e−k5*f+k8*g−k9*b*f−Df*f
 8 Fg+k6*c*e−k7*g−k8*g+k9*b*f−Dg*g
 9 a,b,c,d,e,f,g
10 constant k3,k1,k5,k6,k7,k8,k9,Fa,Fb,Fc,Fd,Fe,Ff,Fg,Da,Db,Dc,Dd,De,Df,
     Dg;
11 k3=.80349;
12 k1=.980389;
13 k5=.8034;
14 k6=.8018;
15 k7=.16876;
16 k8=.7982;
17 k9=.58973;
18 Fa=.4264;
19 Fb=.5284;
20 Fc=.1687;
21 Fd=.167896;
22 Fe=.5673;
23 Ff=.69386;
24 Fg=.79827;
25 Da=.0692;
26 Db=.08762;
27 Dc=.2897;
28 Dd=.0828;
29 De=.26967;
30 Df=.4238;
31 Dg=.5872;
32 0
33 k4    0 0 1 0 40
34 k2    0 0 1 0 50
```

File C.2. Paramotopy input file demonstrating use of computer-generated mesh, and constant declaration. Line 1 indicates 7 equations (lines 2-8), in 1 variable group (line 9), with five parameters (lines 33-34), and 21 constant declarations (10-31). On line 32, `0` tells Paramotopy to make a mesh from the parameters discretized in lines 33-34. Parameter `k4` will be broken into 40 points on the complex line segment $0 + 0i$ to $1 + 0i$. Similarly, `k2` will be broken into 50 points, so a solve using this input file would have 2000 points total in the Step2 run.

```
1  12 1 5 0
2  0.4318*c1*c2 - 0.2435*s1 + 0.0934*a*s1 + 0.0203*a*c1*s2*s3 - 0.0203*a*
     c1*c2*c3 - (0.4318*c4*c5 - 0.1501*s4 - 0.0203*c4*c5*c6 + 0.4331*c4*
     c5*s6 + 0.4331*c4*c6*s5 + 0.0203*c4*s5*s6+delta)
3  0.2435*c1 - 0.0934*a*c1 + 0.4318*c2*s1 + 0.0203*a*s1*s2*s3 - 0.0203*a*
     c2*c3*s1 - (0.1501*c4 + 0.4318*c5*s4 - 0.0203*c5*c6*s4 + 0.4331*c5*
     s4*s6 + 0.4331*c6*s4*s5 + 0.0203*s4*s5*s6)
4  0.0203*a*c2*s3 - 0.4318*s2 + 0.0203*a*c3*s2 - (0.4331*c5*c6 - 0.4318*
     s5 + 0.0203*c5*s6 + 0.0203*c6*s5 - 0.4331*s5*s6)
5  0.4318*c1*c2 - 0.1501*s1 - 0.0203*c1*c2*c3 + 0.4331*c1*c2*s3 + 0.4331*
     c1*c3*s2 + 0.0203*c1*s2*s3 - x
6  0.1501*c1 + 0.4318*c2*s1 - 0.0203*c2*c3*s1 + 0.4331*c2*s1*s3 + 0.4331*
     c3*s1*s2 + 0.0203*s1*s2*s3 - y
7  0.4331*c2*c3 - 0.4318*s2 + 0.0203*c2*s3 + 0.0203*c3*s2 - 0.4331*s2*s3
     - z
8  s1^2+c1^2-1
9  s2^2+c2^2-1
10 s3^2+c3^2-1
11 s4^2+c4^2-1
12 s5^2+c5^2-1
13 s6^2+c6^2-1
14 s1,c1,s2,c2,s3,c3,s4,c4,s5,c5,s6,c6
15 1
16 robomc_10000
17 a
18 delta
19 x
20 y
21 z
```

File C.3. Paramotopy input file demonstrating use of user-defined parameter file. Line 1 indicates 12 equations (lines 2-13), in 1 variable group (line 14), with five parameters (named on lines 17-21), and zero constant declarations. Line 15's `1` indicates Paramotopy should look for the file on the next line, titled `robomc_10000`. The name of the file is arbitrary, but should be at the same path at the input file. The parameters are named `a`, `delta`, `x`, `y`, `z`.

3.1. Monte Carlo Input. Paramotopy enables the user to supply their own plain text file of parameter points over which to solve a parametrized family of polynomials. See File C.3 for an example of such an input file, and File C.4 for an example of the user-defined file.

```
1 0  1.56133 0 -0.156326 0 -0.611479 0 0.430234 0
1 0  1.56133 0 -0.423045 0 -0.178196 0 0.498284 0
1 0  1.56133 0 -0.683401 0 0.0113754 0 0.35804 0
1 0  1.56133 0 0.560595 0 0.178767 0 0.0282532 0
1 0  1.56133 0 -0.391407 0 -0.00876144 0 0.464572 0
1 0  1.56133 0 0.218755 0 0.530148 0 -0.429091 0
1 0  1.56133 0 -0.277281 0 0.620464 0 -0.18738 0
1 0  1.56133 0 0.436947 0 -0.116395 0 -0.503699 0
1 0  1.56133 0 0.464191 0 -0.185408 0 -0.365677 0
1 0  1.56133 0 0.183529 0 0.348453 0 0.344025 0
1 0  1.56133 0 0.327672 0 0.047893 0 -0.608647 0
1 0  1.56133 0 -0.0981337 0 0.505425 0 0.0346073 0
1 0  1.56133 0 -0.595407 0 -0.204762 0 0.601097 0
1 0  1.56133 0 0.798536 0 0.1768 0 -0.193611 0
1 0  1.56133 0 0.501227 0 -0.465084 0 0.515747 0
1 0  1.56133 0 0.234762 0 0.466421 0 -0.370907 0
1 0  1.56133 0 -0.0867984 0 0.540995 0 0.518395 0
1 0  1.56133 0 -0.477477 0 0.125201 0 0.561676 0
```

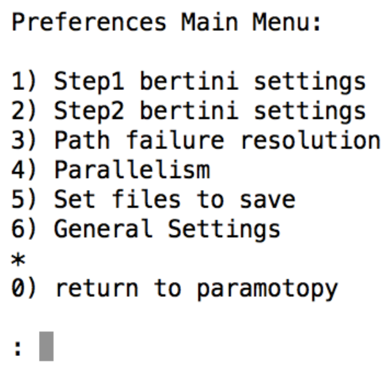
File C.4. User-defined parameter point file. This file is named in the input file (this is `robomc_10000`, as mentioned in File C.3), and must be placed in the same directory. Each line indicates one parameter point, and each real-complex pair is separated by a space.

4. Options & Configuration

Persistent configuration of Paramotopy is maintained through the

`$HOME/.paramotopy/paramotopyprefs.xml`

file located in the home directory. The following sections describe what the settings affect. Note that since the preferences file is merely text, the user could manually hack it with a text editor.



```
Preferences Main Menu:

1) Step1 bertini settings
2) Step2 bertini settings
3) Path failure resolution
4) Parallelism
5) Set files to save
6) General Settings
*
0) return to paramotopy

:
```

Figure 34. The Paramotopy main menu. Interaction is via mainly numeric input at the console.

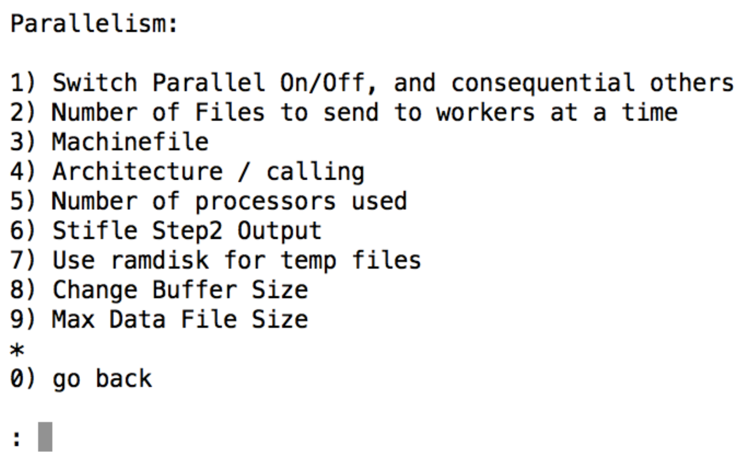
4.1. Parallelism.

Parallelism is achieved using MPICH2. Paramotopy itself acts as a gateway to the Step2 program, where all the real work is done. The gateway allows the user to load input files, make new data folders, change settings, run Bertini for Step1, call Step2, and perform Failed Path Analysis on a completed Step2 run. The gateway model was used because Bertini uses MPI itself, and `MPI_Init()`, `MPI_Finalize()` can only be called once within a program. Therefore, Step2 uses the nonparallel version, whereas Step1 can call either one.

- `Switch Parallel On/Off`

  turn on or off parallel solve mode. If off, then a single processor will be used for all portions of the program
- `Number of Files to send to workers at a time`

  to minimize network traffic, it is advisable to distribute the work in chunks. Setting



```
Parallelism:

1) Switch Parallel On/Off, and consequential others
2) Number of Files to send to workers at a time
3) Machinefile
4) Architecture / calling
5) Number of processors used
6) Stifle Step2 Output
7) Use ramdisk for temp files
8) Change Buffer Size
9) Max Data File Size
*
0) go back

: 
```

Figure 35. Parallelism menu.

this value close to one will kill performance, while setting it near the total number of parameter points will send all the work to a single processor. It is up to the user to find balance, though one rule of thumb would be to set this to:

(total#files /(k×#processors)),

with k perhaps 5 or 10, meaning that 5 or 10 batches would be sent to each processor throughout the computation.

- `Machinefile`

  some MPI installations require a machine file for process distribution.

- `Architecture / calling`

  set the call to start an MPI process on your machine. Two defaults are built in: `mpiexec`, `aprun`. You may also use your own, although there is no error checking for whether your custom command exists, or functions properly. We will soon add the feature of custom calling sequences for more complex command strings.

- `Number of processors used`

  Set the number of processors for Step1, Step2, and rerunning of failed paths (realized as Step2 runs). There should be at least two processors for this mode, as Paramotopy currently uses the master-slave model, with a single master. If you wish to use only one process, switch off parallel mode.

- `Stifle Step2 Output`

  Bertini produces an overwhelming amount of screen output. Not only is this annoying, but sending text to the screen overwhelms the network. It is suggested to stifle only when you are confident in the success of your Step2 runs. Stifling by redirection to `/dev/null` produces a significant speedup.

- `Use ramdisk for temp files`

  Bertini uses hard disk IO to communicate with itself and other processes. This can overwhelm the hard disk, uses read/write cycles reducing the lifetime of a hard drive, and injures performance. Using a location in memory as a hard disk, otherwise known as a ramdisk, will grant you enormous performance gains. Two locations are scanned for existence by default: `/dev/shm` and `/tmp`. If either are found, the user is prompted for confirmation. The user may also choose a custom location. However, at present time the name of the temporary file location root directory must be the same across all computers used. Note: Paramotopy does not guarantee that the chosen location is usable in the desired fashion; the user must ensure that the temporary location is read/writable.

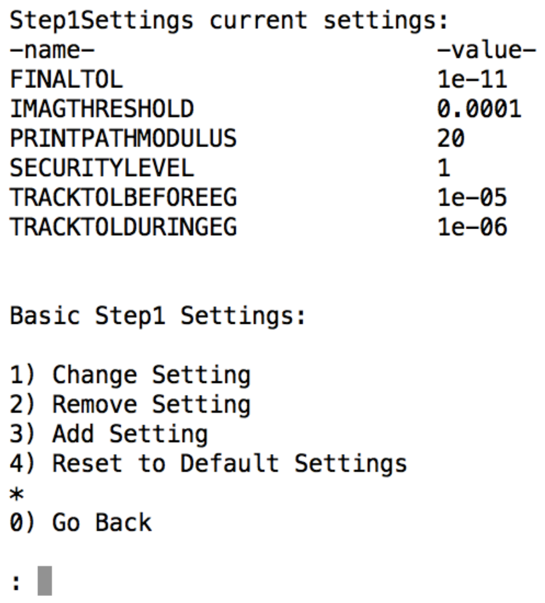
- `Change Buffer Size`

  to minimize writes to the hard drive, each worker buffers its files in memory, and writes to disk only when a threshold is reached. To change this threshold, change this buffer size.

- `Max Data File Size`

  The text files produced by Paramotopy grow in time, of course. To ensure that you don't produce unwieldly files, change the maximum file size for data files. The default is 64MB. This maximum is not a *hard* maximum, in the sense that a new file is started only after the max is achieved.

4.2. Step1 and Step2 Bertini Settings. Separate settings categories are present for both the Step1 and Step2 runs, and persist from run to run, and session to session. These are written directly into the input files for Bertini, and there is no error checking – if the user sets a setting to one disallowed, or misspells a name, Step1 or Step2 will just ignore the setting.

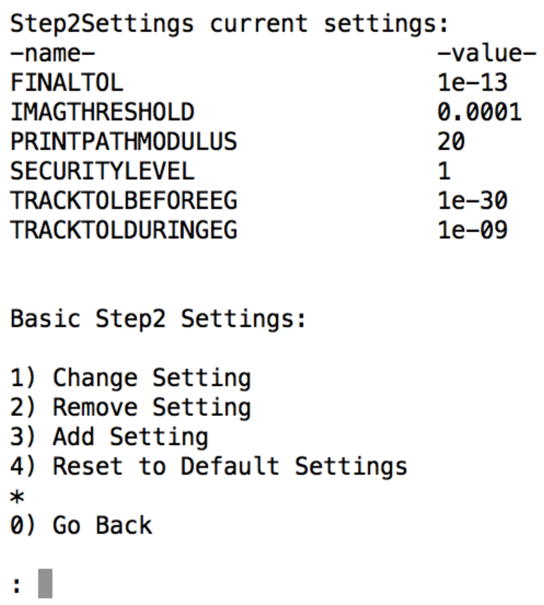


```
Step1Settings current settings:
-name-                        -value-
FINALTOL                      1e-11
IMAGTHRESHOLD                 0.0001
PRINTPATHMODULUS              20
SECURITYLEVEL                 1
TRACKTOLBEFOREEG              1e-05
TRACKTOLDURINGEG              1e-06


Basic Step1 Settings:

1) Change Setting
2) Remove Setting
3) Add Setting
4) Reset to Default Settings
*
0) Go Back

: 
```

Figure 36. Step1 Settings. All settings are direct Bertini config options.

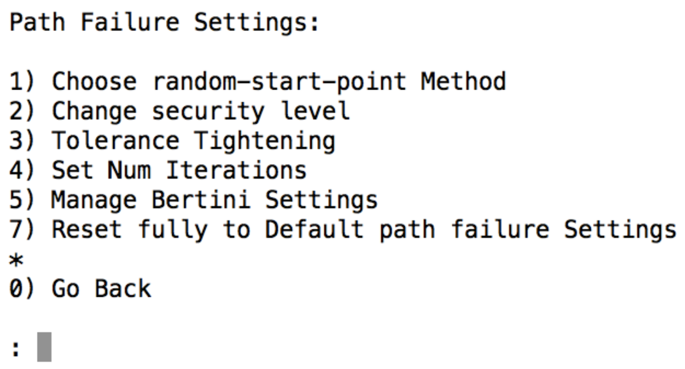
Options are to change, delete, and add a setting. User may also reset all of that Step's settings, and reset to default, which are the Bertini default values as well. Note that each setting has a type, being either a string, integer, or double value. Doubles may use the `1e-4` format for convenience. While capitalization does not matter in Bertini, Paramotopy is currently case-sensitive, so deleting or changing a setting requires the same capitalization as displayed.

4.3. Path Failure. Paramotopy detects failed paths that occur during a run, and has methods for resolving the system, repeatedly if necessary, to get the proper solution set.



```
Step2Settings current settings:
-name-                        -value-
FINALTOL                      1e-13
IMAGTHRESHOLD                 0.0001
PRINTPATHMODULUS              20
SECURITYLEVEL                 1
TRACKTOLBEFOREEG              1e-30
TRACKTOLDURINGEG              1e-09


Basic Step2 Settings:

1) Change Setting
2) Remove Setting
3) Add Setting
4) Reset to Default Settings
*
0) Go Back

:
```

FIGURE 37. Step2 Settings. All settings are direct Bertini config options.



```
Path Failure Settings:

1) Choose random-start-point Method
2) Change security level
3) Tolerance Tightening
4) Set Num Iterations
5) Manage Bertini Settings
7) Reset fully to Default path failure Settings
*
0) Go Back

:
```

FIGURE 38. Path Failure Settings menu. The path failure analysis section of the program utilizes the Bertini preferences from the Step1 and Step2 sections, optionally changing them with successive runs.

- `Choose random-start-point Method`

  When resolving points with failed paths, Paramotopy can optionally choose a new start point before solving the set of fails. To be developed is a variety of methods for choosing this start point; *e.g.* specifying a distance from the original start point, etc.

- `Change security level`

  ensure `securitylevel=1` is turned on, for more secure solution of failed paths.

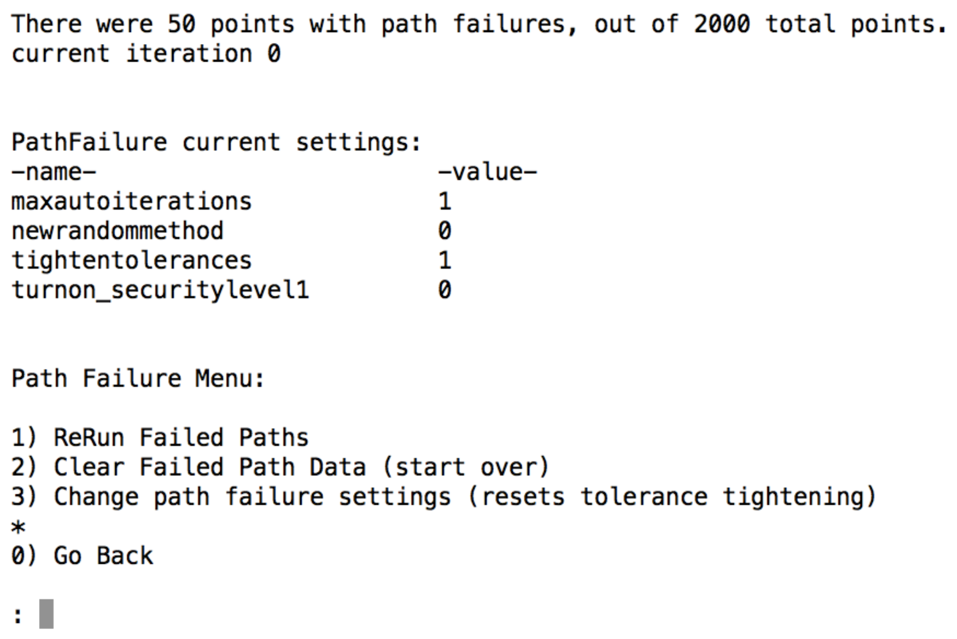

```
There were 50 points with path failures, out of 2000 total points.
current iteration 0


PathFailure current settings:
-name-                        -value-
maxautoiterations             1
newrandommethod               0
tightentolerances             1
turnon_securitylevel1         0


Path Failure Menu:

1) ReRun Failed Paths
2) Clear Failed Path Data (start over)
3) Change path failure settings (resets tolerance tightening)
*
0) Go Back

: 
```

FIGURE 39. Path Failure Re-solve menu. User can continue to attempt to re-solve, adjust settings, and delete files and start over.

- `Tolerance Tightening`

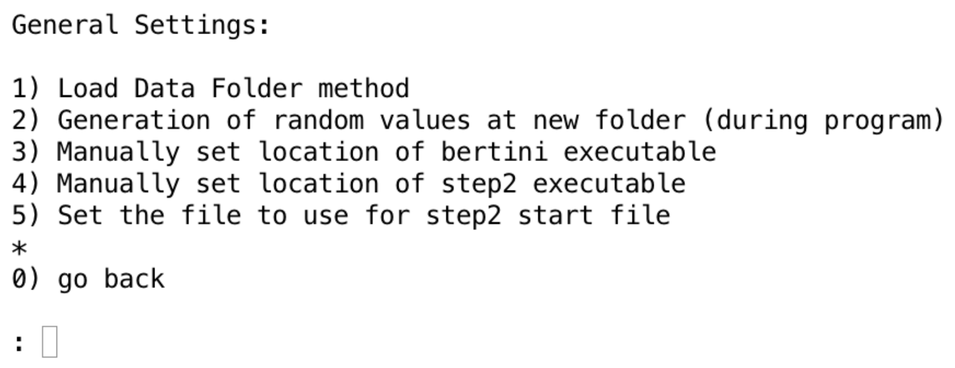
  When Failed Path Analysis is entered in Paramotopy, the settings are those currently used for Step2 runs. If tolerance tightening is enabled, at each iteration of resolve, the `finaltol`, `tracktolbeforeeg`, and `tracktolduringeg` values are decreased by a factor of ten.

- `Set Num Iterations`

  Path Failure mode will automatically try to re-solve failed paths [up to] this number of times before returning back to the user for input.

- `Reset to Default Settings`

  Reset PathFailure settings to default hardcoded values.



```
General Settings:

1) Load Data Folder method
2) Generation of random values at new folder (during program)
3) Manually set location of bertini executable
4) Manually set location of step2 executable
5) Set the file to use for step2 start file
*
0) go back

: 
```

Figure 40. General Settings menu. Sets behavior for such things as start point generation at creation of a new folder, and whether to create a new data folder at launch.

4.4. General.

- `Load Data Folder Method`

  at launch of Paramotopy, if not supplied with an argument of the filename to load, Paramotopy will ask the user for the filename. Each filename gets its own folder for storing data, and in this folder will be another folder containing the data for a run. At launch, Paramotopy can do one of the following:

  - Reload data from the most recent run,
  - Make a new run folder, and make new start point,
  - Ask the user what to do.

- `Generation of random values at new folder (during program)`

  If, while running the program, the user wishes to start a new run, should Paramotopy:

  - Keep the random values currently in use, or
  - Make a new set of random values.

- `Manually set location of bertini executable`

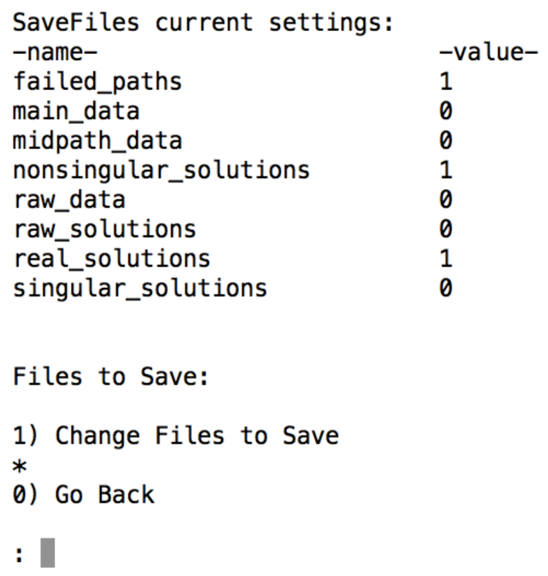
  Paramotopy automatically scans the current directory (`./`) and `$PATH` for Bertini at startup, in that order, and chooses the first instance of Bertini it finds. If the user wishes to manually change the location for Bertini, they should this option.

- `Manually set location of step2 executable`

  Same as for Bertini.

- `Set the file to use for step2 start file`

  User may select to use either the `nonsingular_solutions` or `finite_solutions` output from Step1 as the start file for Step2 solves.



```
SaveFiles current settings:
-name-                        -value-
failed_paths                  1
main_data                     0
midpath_data                  0
nonsingular_solutions         1
raw_data                      0
raw_solutions                 0
real_solutions                1
singular_solutions            0


Files to Save:

1) Change Files to Save
*
0) Go Back

: 
```

FIGURE 41. Paramotopy can save each of the output files from Bertini, for each point at which it solves the input system. `failed_paths` is always saved; the rest are optional.

4.5. FILE SAVING. Bertini produces several output files which may be desirable to keep. The file named `failed_paths` is always saved, to enable Failed Path Analysis. All other files are optional. Some are more useful than others for humans or computers. For a more thorough description of the output of Bertini, see the Bertini User's Manual.

5. Data

Paramotopy is capable of producing immense amounts of data. Probably the two most useful files you can choose to save are `nonsingular_solutions` and `real_solutions`. Both are easy to have a machine parse, with predictable numbers of lines per parameter point.

The polynomial systems at some points in parameter space will be faster to solve than others. Combined with the use of a parallel machine, this will result in data files which are out of order. Whatever data collection method you use, you must deal with this fact.

Data collection is achieved by simply copying the Bertini output file into memory, and then, along with the index of the parameter point and the parameter values, into a collective output file. The format is essentially what appears in File C.5:

```
 1 Pr1  Pi1  Pr2  Pi2
 2 0
 3 0  0  0  0  0  0  0  0
 4 2
 5
 6 -0.200890604423765803e-40  -0.8609545597332244710586e-41
 7 0.8394559733244710586e-41  0.21882676674265533030607e-40
 8
 9 0.000000000000000e+00  7.589415207398531e-19
10 7.589415207398531e-19  -9.351243737687476e-19
11
12 1
13 0.555556  0  0  0  0  0  0  0
14 2
15
16 -7.318364664277155e-19  9.520650327138336e-19
17 1.218033378151684e-18  0.000000000000000e+00
18
19 -0.8121868223093549866e-18  0.8704335463135312587e-18
20 0.1229193841385923155e-17  0.1420916030667911630e-19
```

File C.5. Example output file from Paramotopy. The first line of each file created by Paramotopy declares the names of the parameters. Each parameter point gets its own section, with the index (lines 2, 12), parameter values (lines 3, 13), and copied Bertini output. This is an example of `real_solutions` output, so lines 4,14 indicate the number of solutions which follow. The actual solutions appear as well. Note that lines 4-10, and 14-20, are unmodified Bertini output, complete with whitespace.

The author Daniel Brake has some generic MATLAB code which can do basic parsing of some file types, as well as data display methods. If you need help with any aspect, don't hesitate to ask! Contact info in Section1.1.

6. Troubleshooting

- Step1 fails
  - Are all the settings properly spelled, and allowed?
  - Was the calling sequence correct?
  - Do you have the *parallel* version of Bertini installed?
  - Try running Bertini on the created input file.
- Step2 fails
  - Are all settings spelled properly and allowed?
  - Was the calling sequence correct?
  - Did Paramotopy correctly parse your input file?
  - Did the temporary files write to an acceptable location?
  - Try running Bertini on the created input file. $\rightarrow$ Bertini might provide a useful error report.
  - Turn off stifling of output, so that you can read any generated error messages.