2012

Improvements in Genetic Approach to Pole Placement in Linear State Space Systems Through Island Approach PGA with Orthogonal Mutation Vectors

Arnold Cassell
University of North Florida, n00779662@ospreys.unf.edu

IMPROVEMENTS IN GENETIC APPROACH TO POLE PLACEMENT IN LINEAR STATE SPACE SYSTEMS THROUGH ISLAND APPROACH PGA WITH ORTHOGONAL MUTATION VECTORS

by

Arnold M. Cassell

A thesis submitted to the School of Electrical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

UNIVERSITY OF NORTH FLORIDA

COLLEGE OF COMPUTING, ENGINEERING AND CONSTRUCTION

December, 2012

The thesis of Arnold Mathew Cassell is approved: (Date)

_____ _____
Dr. Chiu Choi

_____ _____
Dr. Patrick Kreidl

_____ _____
Dr. Daniel Cox

Accepted for the School of Engineering:

_____ _____
Dr. Murat Tiryakioglu
Chair

Accepted for the College of Computing, Engineering, and Construction

_____ _____
Dr. Mark Tumeo
Dean

Accepted for the University:

_____ _____
Dr. Len Roberson
Dean of The Graduate School

ACKNOWLEDGEMENT

I would like to thank the members of my committee, who challenged me and supported me from the beginning. You've all treated me like more than just another student, and I appreciate that.

I would also like to thank my parents, who never thought I didn't have it in me, no matter what "it" was.

And I would like to thank Kelley. I'm just as excited to see what comes next as you are.

Dr. Choi introduced me to this topic in my first class at UNF, and I decided right away that it was important and exciting enough to research. We worked on this topic before and published a research paper in the 44th SSST Conference. This thesis is an extension of my previous work.

CONTENTS

List of Figures

ABSTRACT

This thesis describes a genetic approach for shaping the dynamic responses of linear state space systems through pole placement. This paper makes further comparisons between this approach and an island approach parallel genetic algorithm (PGA) which incorporates orthogonal mutation vectors to increase sub-population specialization and decrease convergence time.

Both approaches generate a gain vector K. The vector K is used in state feedback for altering the poles of the system so as to meet step response requirements such as settling time and percent overshoot. To obtain the gain vector K by the proposed genetic approaches, a pair of ideal, desired poles is calculate first. Those poles serve as the basis by which an initial population is created. In the island approach, those poles serve as a basis for n populations, where n is the dimension of the necessary K vector.

Each member of the population is tested for its fitness (the degree to which it matches the criteria). A new population is created each “generation” from the results of the previous iteration, until the criteria are met, or a certain number of generations have passed. Several case studies are provided in this paper to illustrate that this new approach is working, and also to compare performance of the two approaches.

GLOSSARY

characteristic equation: the characteristic equation of a square matrix A is the equation in one variable λ such that $det(A - I\lambda) = 0$ where det is the determinant and I is the identity matrix

closed system: a system that is controlled by a combination of system inputs and information gathered about the current internal state of the system

feedback: the return of part of the output of an electronic circuit, device, or mechanical system to its input, so modifying its characteristics.

model: a simplified representation or description of a system or complex entity, esp one designed to facilitate calculations and predictions

open system: a system that does not have any feedback loop to control its output. The output is dependent entirely on the input and the internals of the system.

order: in a state space model, the order is a number describing the number of internal (state) variables of the model. A second order system, therefore, would have two state variables.

overdamped: if the step response of a system has no overshoot and a slow response, it can be said to be overdamped

overshoot: the measure of the peak value of a system response minus the final, settled value of that response **percent overshoot:** a ratio of the overshoot of the response of a system as it relates to the final, settled value of that response. $percent\ overshoot = \frac{peak\ value - settled\ value}{settled\ value}\ x\ 100\%$

poles of a system: values of s such that the denominator of the transfer function of a system $H(s) = \frac{Y(s)}{X(s)}$ is driven to zero. Any value of s such that X(s) = 0

state space model: a model of a system which is defined by the existence of state variables. This method relates the time rate of change of those state variables to the system input and the values of the state variables, and also relates the output to the state variables.

state variable: a value internal to a system that may be affected by user input or the values of other state varaibles, and which may or may not contribute to the output of the system

settling time: the time required for the response curve to reach and stay within a range of certain percentage of the final value. For the purpose of this thesis, it is convention that this refers always to the 2% settling time, or the time required for the response of a system to reach and stay within 2% of the final value

stochastic process: a process involving a random variable the successive values of which are not independent

system poles: see "poles of a system"

system zeros: see "zeros of a system

transfer function: a mathematical representation, in terms of spatial or temporal frequency, of the relation between the input and output of a linear time-invariant system with zero initial conditions and zero-point equilibrium. In this thesis, the transfer function will always be shown as $H(s) = \frac{Y(s)}{X(s)}$ where $X(s) = \mathcal{L}\{x(t)\}$, being the Laplace transform of the input signal, and $Y(s) = \mathcal{L}\{y(t)\}$, being the Laplace transform of the output signal.

zeros of a system: values of s such that the numerator of the transfer function of a system $H(s) = \frac{Y(s)}{X(s)}$ is driven to zero. Any value of s such that Y(s) = 0

CHAPTER 1: INTRODUCTION

1.1 – The Importance of this Work

The goal of modeling a system is prediction. A well-defined model can predict a system's behavior given any stimulus. A good prediction can be used during the design phase, to build a system which will behave in a desired fashion. Such a model, and the predictions made from it, are used well in advance and inform the design of a system.

Another way to use that prediction is to maintain control of the system. If it is known how the system will respond, that information can be used in the moment to keep order, to minimize unwanted variations, and to speed the desired response. These predictions are calculated and used by a "controller" within the system.

This thesis sets out to improve a "controller" method. The problem it tackles involves state-space models. These models (which will be explained in detail in chapter 2) rely on internal measurements within the system, such as the ratio of oxygen to fuel in a combustion chamber, or the current speed of a wheel, to predict the output of a system.

The technique this paper focuses on is feedback: using the values of those internal measurements, those "state variables," in combination with the user input, to get the desired output. The particular difficulty of using this technique is that larger models (systems involving more than two of those state variables) are much harder to calculate the appropriate feedback for. Chapter 2 will discuss the difficulties in detail.

A genetic approach is described first, in which some initial guesses are used, and then some randomness is introduced to guide the search. This method is used only as a baseline. The real focus of this paper is on how to improve that search for the appropriate feedback.

The improvements discussed in this paper are in dividing up that search to multiple, specialized areas, while still maintaining an exchange of "good" answers in each area. It is through the combination of these techniques that the search ends faster, and the desired feedback to control the system is achieved in less time.

1.2 - Overview

The goal of this thesis is to compare a basic genetic algorithm with an "improved" algorithm. Both algorithms are set to the same task, which is modifying the behavior of an existing system to a set of performance criteria. They both achieve this by searching for appropriate feedback to supplement the input. Many different systems, with many different performance criteria are tested, and the strengths of the "improved" algorithm will be compared with the baseline performance of the original genetic algorithm.

1.3 - Outline

The first section of this paper will briefly describe state space models, and what is necessary to create them. This will be necessary to discuss pole placement, which is the feedback technique used to guide system performance in these algorithms.

Then a few examples will be given, displaying this pole placement technique, and the difficulties in applying it. Direct calculation of the solution is not always possible. A few other methods of searching for a proper solution will be described before settling on the motivation for a genetic approach.

The basic genetic algorithm will be described in detail first. This will highlight how the genetic methodology was tuned to this problem. Then the enhanced, "island" approach will be

discussed, along with the changes that were made and a brief description of their desired effect.

Lastly, the test procedure, including the tested systems and the results will be discussed.

CHAPTER 2: BACKGROUND

2.1 - State Space Models

A system is broadly defined as a group of regularly interacting bodies. [10] One very basic way of looking at a system is by observing its behavior with respect to time (t).

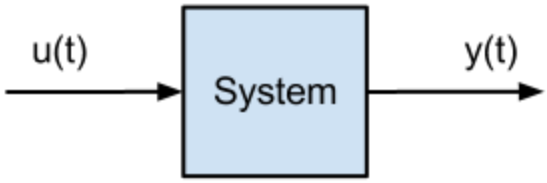

Figure 1: Simple System Diagram

Figure 1 shows the input to the system as *u(t)* and the output of the system as *y(t)*. The system block encompasses all the interacting bodies in-between. In this very basic model, nothing is known other than what goes in and what comes out.

Once the behavior of a system (given a predefined input) is known, it can be controlled. When a user plans the input of a system with the goal of producing a specific output, they are said to take control of that system.

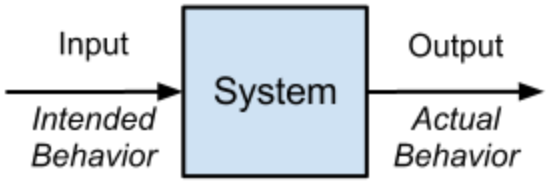

Figure 2: Simple System Diagram Revisited

Many real systems have some degree of difference between their intended behavior and the actual realized behavior. A toaster, for example, does not transform bread into toast instantly;

it introduces a delay between input and output. An air-conditioning unit does not keep all of the air in a house at the desired temperature; it turns on and off periodically to keep most of the air within a range of temperatures.

Most toasters are examples of open loop systems. When the user turns the toaster on, the same heat is applied regardless of what kind of bread is inserted, and the toaster does not monitor the color or temperature of the bread. The user may intend a golden-brown, but what is produced is controlled only by the user settings and bread type.

The air-conditioner, on the other hand, is a closed loop system. The ambient air temperature is constantly measured, as well as the temperature of the compressor coils, and the speed of the fans which distribute the cool air. The system monitors all of these internal variables and controls them in order to produce a desired output: to keep most of the air at the desired temperature.

This paper is only concerned with closed loop systems, and is targeted at finding the amount of internal feedback that will bring the actual performance of a system closer to the intended performance of a system.

To achieve this proper combination of feedback, it is necessary to model the system in what is known as state space. A state space model is more complex than the above diagrams, in that it details more of the internal workings of a system. Not only does a state space model monitor some of the internal variables, these state space models incorporate a method of feedback into the system. In this way, more information is gathered about the inner workings of the system.

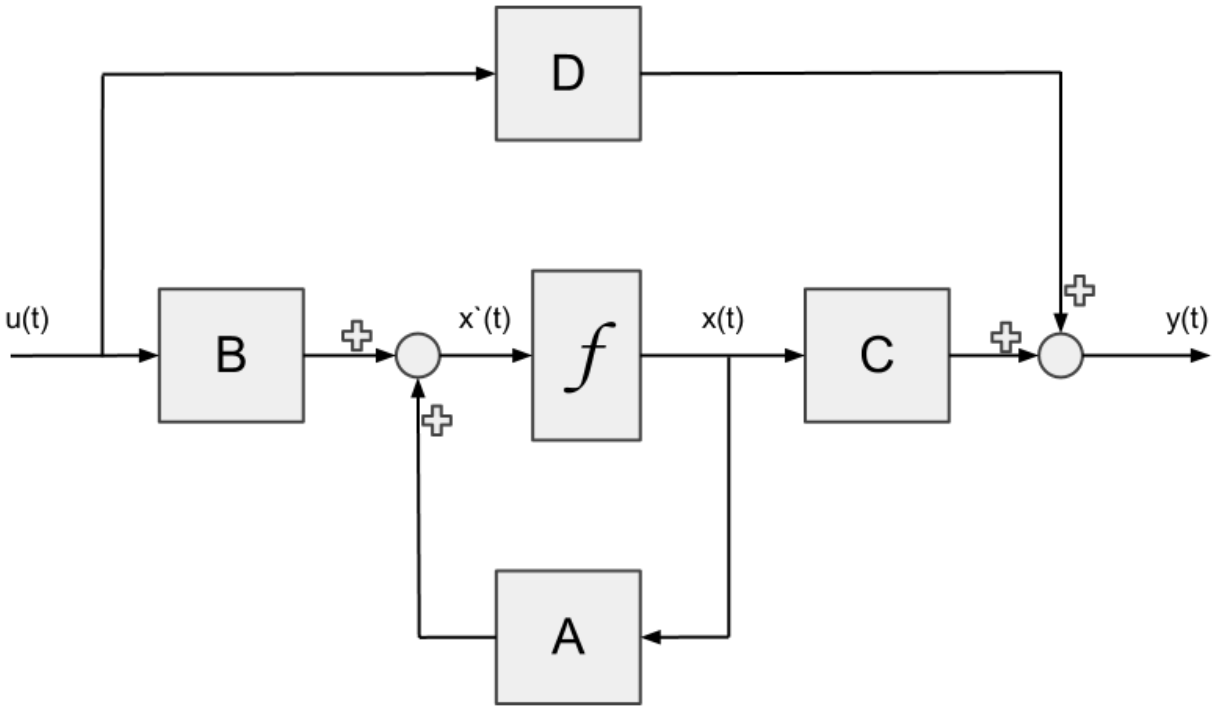

Figure 3: State Space Model Diagram

A state space model is a way of mathematically relating the system's input, the internal components of the system, and the system's output. The example system shown in Figure 3 has been broken down into three distinct functions.

- *u(t)* describes the input to the system
- *x(t)* describes the inner workings of the system
- *y(t)* describes the output from the system

This new component, *x(t)*, is the state vector of the model. This vector contains all of the state variables for the model.

It is also shown in the figure above that some parts of the inner workings, $\dot{x}(t)$ depend on both the input *u(t)* and also *x(t)*. $\dot{x}(t)$ is the time rate of change of the state variables, or $\frac{dx(t)}{dt}$. In

a simple RLC circuit, these variables may be voltages or currents, which are not directly controlled by the user input, but some combination of the input signal and their current state.

A state space model is characterized by this state vector, and its relationships to the other components. In order to represent a system in a state space model, it is necessary to be able to define the derivatives of each of the elements in the state vector as a linear combination of the values of the other elements in the state vector and some input to the system.

The general form of which is:

$$\dot{x}(t) = Ax(t) + Bu(t) \qquad (1)$$

Where $\dot{x}(t)$ is a vector containing all of the derivatives of all the state variables at some time *t*. The A matrix, then, would be a matrix of coefficients which are multiplied by the elements in the state vector *x(t)*, and B is a vector relating the input *u(t)* to $\dot{x}(t)$.

The system output is similarly composed of some part of the internal variables *x(t)* and some direct mapping from the input to the output.

$$y(t) = Cx(t) + Du(t) \qquad (2)$$

Techniques for how to model an existing system as a state space model will not be discussed in this paper, but it is necessary to understand the basics of this model and the terms used in it. Figure 3 will be expanded upon to include a feedback technique in the next section.

2.2 - Pole Placement Method

The state space model representation of the system is a powerful tool because much of how much information can be predicted from the A matrix itself. Since the A matrix has a

number of rows equal to the number of elements in the *x(t)* vector, and also a number of columns equal to that same number of elements, A is always a square matrix. A system's behavior can be predicted in large part by the eigenvalues of the A matrix.

Eigenvalues are defined as the solution(s) to the equation:

$$\det(A - \lambda I) = 0 \qquad (3)$$

Or, more graphically:

$$\det\left(A - \begin{bmatrix} \lambda & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda \end{bmatrix}\right) = 0 \qquad (4)$$

The determinant of the resulting matrix is commonly referred to as the "characteristic equation" of the matrix. The solutions for λ are commonly called the "poles" of the system. If the same system were instead represented as a transfer function, the eigenvalues of this A matrix would correspond to roots in the denominator of the transfer function.

The behavior of the state variables is dependent on the coefficients of the A matrix. Those coefficients detail the internal behavior of the system, as well as informing the output of the system. Since so much of the behavior is dependent on the workings of the A matrix, altering the A matrix is an easy way to alter the performance of the system. However, changing the coefficients seems to require altering the system itself, and changing how the internal components relate to one another.

In a mechanical system, this could require replacing gears or belting, for instance. In an electrical system, this might require replacing passive components or adding summing circuits.

Fortunately, there is another method. If an additional feedback channel is added to the input of the equation, one that is some combination of the system's actual state along with the user input, the A matrix can be effectively altered, and the system's performance can thereby change, without tampering with the existing system or model. This "full state feedback" or "pole placement" is a classic method in control theory, developed in part by Eduardo Sontag. [2]

The following substitution is made.

$$u(t) = r(t) - Kx(t) \qquad (5)$$

For simplicity's sake, the user input has been renamed from *u(t)* to *r(t)* so that the combined input is still named *u(t)*. The feedback vector K is multiplied by the state vector, and then subtracted from the user input to form the total input to the system.

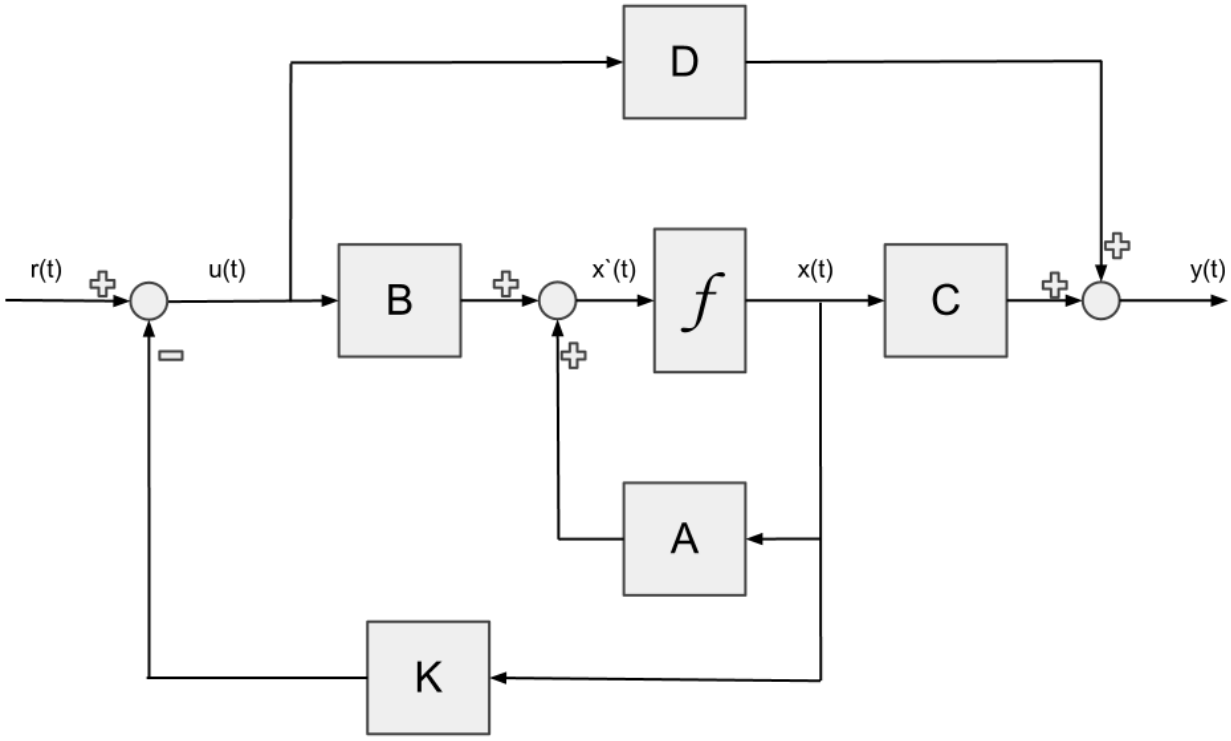

Figure 4: State Space Diagram with K Vector Feedback

The new equation governing the behavior of $\dot{x}(t)$ in Figure 4 above is:

$$\dot{x}(t) = (A - BK)x(t) + Bu(t) \qquad (6)$$

If the K vector is non-zero, the matrix relating the state variables to their derivatives will have changed. By changing the A matrix into (A-BK) the characteristic equation, the system's poles, and thereby its dynamic response have all been altered.

It is worth mentioning at this time that in order for a system to be stable the poles of the system must exist in the left half of the number plane: they must be non-positive values. A stable system is one that will settle into an equilibrium state. Additionally, the poles of these systems often exist as complex conjugate pairs, as real numbers, or some combination of both. [2,3,10]

Some work has been done to predict the behavior of a system based solely on the poles that exist in the A matrix [2,3,11]. The specific behavior that this thesis is chiefly concerned with is the system's response to a step input. A step input is defined as an instant change (in the input function *r(t)*) from zero to 1. This paper will always assume that the system was at rest before the input was applied. The performance metrics monitored will be the settling time of the system and the percent overshoot.

Settling time is the time delay in a system before it settles to its final value. Percent overshoot is the highest output value during the step response minus the final settled value, which is then divided by the final value and multiplied by 100%.

A prediction of the system's behavior (the overshoot and settling time) can be made by finding the poles of the system, whether they be eigenvalues of the A matrix, or zeroes in the denominator of the transfer function. The same equations can be worked in reverse. Starting

from a desired behavior, the desired poles can be calculated, and an A matrix can be constructed. Such methods center around finding damping coefficients and natural frequencies in the transfer function, and are outside the scope of this thesis. Interested readers are encouraged to find detailed execution and theory in "Control Systems Engineering" by Norman Nise.

Simple linear algebra can follow on and find a vector that transforms an existing matrix into one with the desired poles (eigenvalues). If the new, desired poles are represented as λ_i's, then K vector can be found by setting the following equal to the newly desired characteristic equation.

$$\det\left(\left(\lambda I - (A - BK)\right)\right) \qquad (7)$$

This direct method of calculation can only produce at most one complex conjugate pair of poles. If the A matrix is of dimension larger than 2x2, then it will have more eigenvalues, more poles, and the rest of the desired poles will have to be guessed at using heuristics. Often, it is sufficient to simply choose the rest of the desired poles as large negative real numbers. [1]

2.3 - A Successful Application of Pole Placement

An example is given below to illustrate the procedure of the method of Pole Placement. Consider the state space model:

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -5 & -6 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u \qquad (8)$$

$$y = \begin{bmatrix} 12 & 0 & 0 \end{bmatrix} x$$

This state space model is defined as being third order, because the model contains three elements in the state vector. For this example, an attempt is made to drive the percent overshoot to 7%, and the 2% settling time to 4.5s. Direct calculation leads to the desired system poles of -0.889+i*1.056 and -0.889-i*1.056. The methods for finding these poles are available in the literature. [2] Since this example system is third order, a third desired pole is needed. This third pole is to be selected far away so that the first two poles become the dominant poles, and also negative so that it does not drive the system to instability. The third pole is selected to be -10.

The gain vector *K* is chosen to place the eigenvalues of *A-BK* at the desired poles, which are -0.889 + i*1.056, -0.889 - i*1.056, and -10. The method used to compute *K* is implemented as the "place" command in MatLab© [3], which uses the A matrix, the B vector, and the desired poles as input.

The system was transformed to account for the inclusion of the K vector feedback, and a MatLab© simulation was performed. The step response of the system is shown in Figure 5.

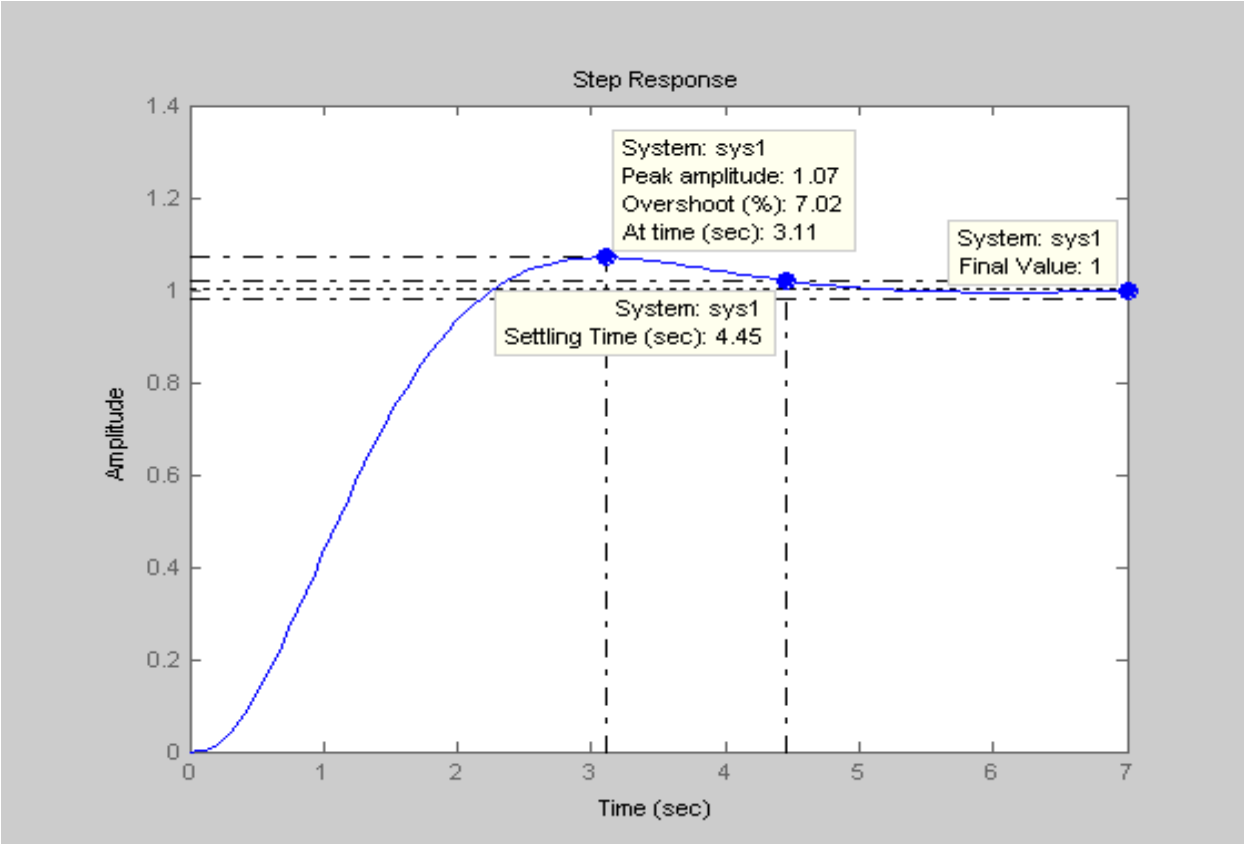

Figure 5: step response of (8) after pole placement

Notice the overshoot and settling time are close to the desired values. The third pole at -10 has had some effect, but it is overshadowed by the effect of the dominant poles.

2.4 - Weaknesses in Pole Placement Method

Let us look at another example, this one seemingly simpler because it only has two state variables, and therefore two poles. By convention, this is called a second order system.

$$A = \begin{bmatrix} -1 & 3 \\ 0.005 & 0 \end{bmatrix} \qquad (9)$$

$$B = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$D = [0]$$

The poles of the A matrix are currently -1.0148 and 0.0148. By itself, the system is not stable. The K vector will be used to stabilize the system, and also drive it towards the desired performance criteria. For this example, those criteria are 14% overshoot and a settling time of 2.8s. Direct calculation yields the desired poles at -1.3972 ± 1.6632i. Further, the K vector to transform the A matrix is

$$K = [0.3671 \quad 0.4757] \qquad (10)$$

The transformed system is as follows:

$$A = \begin{bmatrix} -1.367 & 2.524 \\ -1.096 & -1.427 \end{bmatrix} \qquad (11)$$

$$B = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

$$C = [0 \quad 1.57]$$

$$D = [0]$$

The eigenvalues of A can be verified to be -1.3972 ± 1.6632i, just as desired. When this system's step response is measured, however, the achieved percent overshoot is 55.50% and the achieved settling time is 3.33s. Direct calculation has not yielded the desired results of 14% overshoot and a 2.8s settling time. This is because the system has a zero as well as the two poles, and all three affect its performance. The estimation can only account for the impact of two dominant poles.

Converting the system from the state space model to a transfer function yields the following:

$$H(s) = \frac{4.71s + 4.718}{s^2 + 2.794s + 4.717}$$

The system has two poles, but it also has one zero; should *s* be equal to -1.0017 the numerator of the transfer function will become zero.

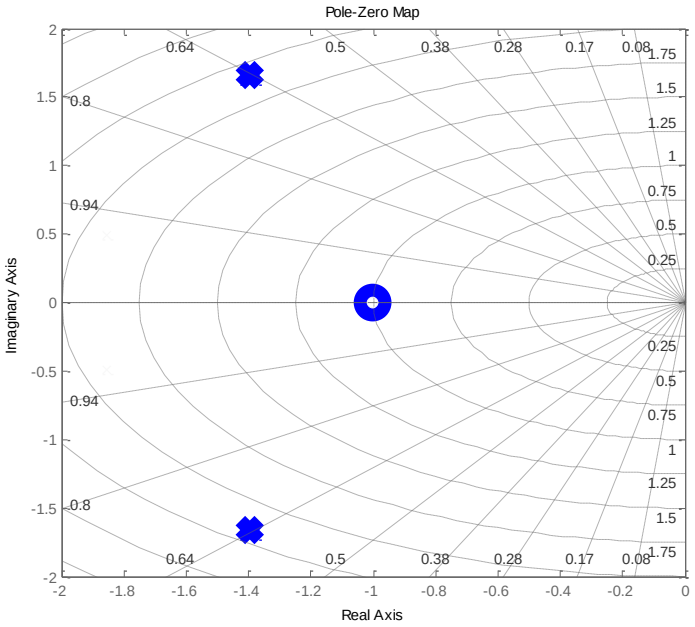

Figure 6: pole-zero map of (11)

Taking a step back, the original system demonstrates the exact same zero; should *s* be equal to -1.0017 the numerator of the transfer function will become zero.

$$H(s) = \frac{3s + 3.0051}{s^2 + s - 0.015}$$

The addition of the K vector feedback has altered the poles of the system, but it has had no effect on the zero, and both poles and zeros affect the step response.

A different K vector, found through experimentation, does better than direct calculation:

K = [0.122353 0.86207] (12)

This transforms the system into the following form:

$$A = \begin{bmatrix} -1.122 & 2.138 \\ -0.3621 & -2.586 \end{bmatrix} \quad (13)$$

$$B = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

$$C = [0 \quad 1.223]$$

$$D = [0]$$

The poles of this system are at -1.854 ± 0.4884i. Direct calculation tells us that the percent overshoot of this system should be 46.127, and settling time should be 2.11s. However, this method of prediction ignores the presence of zeros. When measured, the following is the observed:

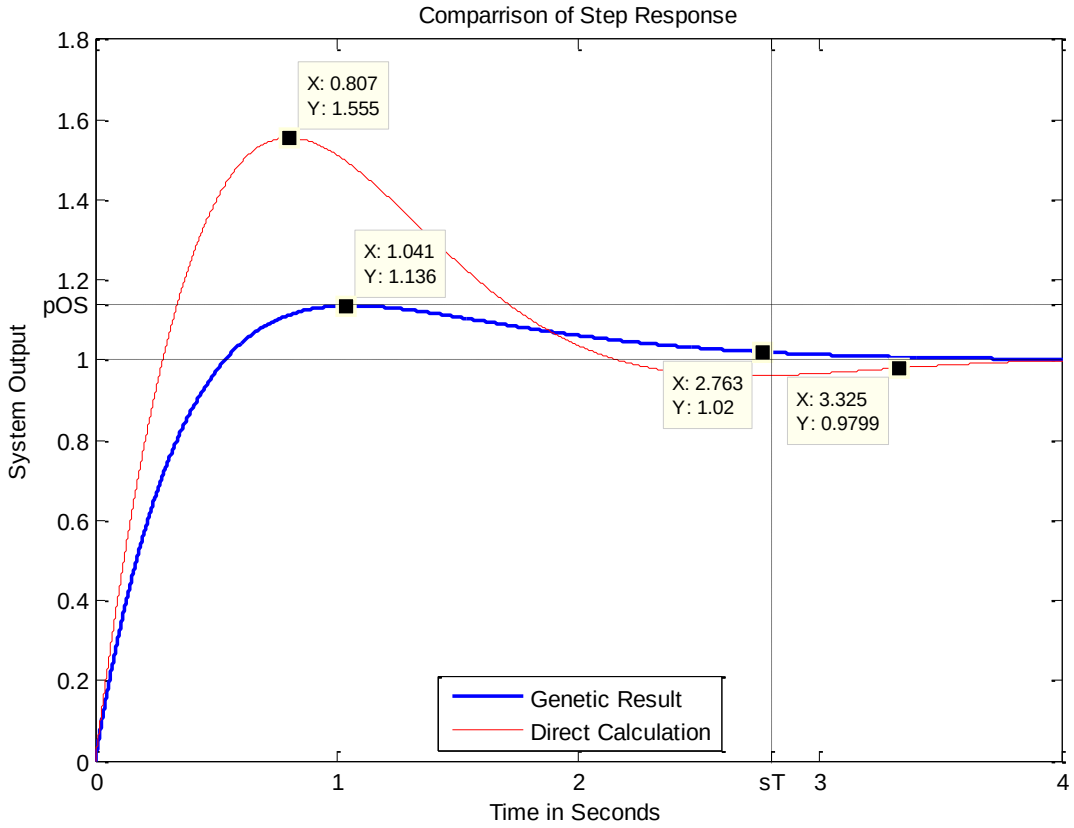

Figure 7: step response of (13) and (11)

Here the two results are presented together. The desired percent overshoot and desired settling time have been marked as well. The genetic result is observed to have a percent overshoot of 13.61% and a settling time of 2.76s. There can be no question that the later K vector, found by experimentation rather than direct calculation, is the better match for the desired performance criteria.

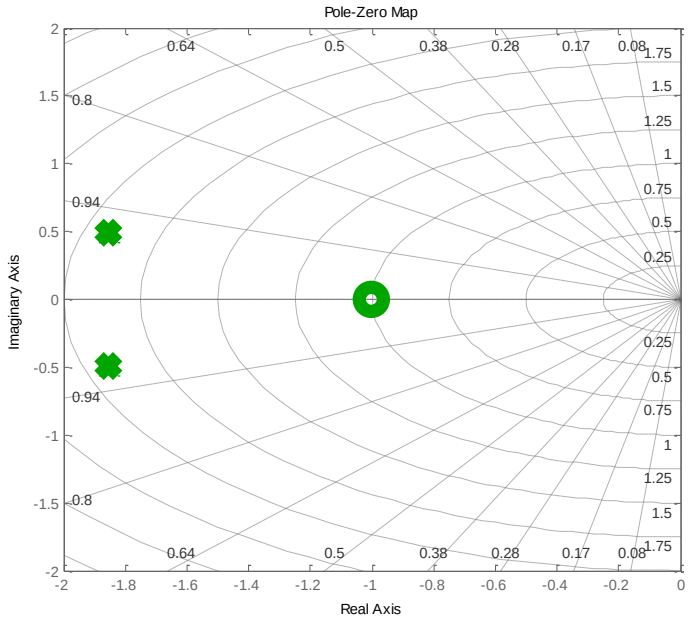

Figure 8: pole-zero map of (13)

Though the zero is still present, the new position of the poles has compensated for the zero, and the performance criteria are met. Finding the appropriate K vector is not trivial in the presence of zeros, but there are several methods to employ in the search.

2.5 - K Vector Search Methods

Feedback vectors may be calculated any number of ways. When direct calculation fails, or is impossible due to zeros in the transfer function, it is possible to use a gradient search. In such a search, an initial guess is created, and then the elements in that guess are changed iteratively until an optimal solution is reached.

The effect of a change to a single element of the K vector cannot be directly predicted, however, and must be observed through experimentation. The search is guided by avoiding

changes that increase settling time or overshoot, and favoring those changes instead that decrease those values.

When a change is found that guides the system closer to the performance criteria, a gradient search method would re-center itself at that new point, and begin to map the surrounding area again, looking for further improvements.

This approach is short-sighted by its very nature, and only considers small changes. It is very likely that, while a small positive change in the 3rd element might result in worse performance, a large positive change might overcome that performance decrease and improve the overall performance of the system.

It is also possible that some combination of elements need to be increased simultaneously to get the performance desired. Even if a gradient search were to attempt a large sequence of changes on the initial guess, a gradient search will only seek out localized minima, and cannot hope to fully map the solution space. It will ultimately settle at a point from which any small change decreases performance, and go no further. There's no guarantee the performance at that point is the best possible.

An iterative approach could also be considered. In such a method, many thousands of combinations of elements would be tried out in an attempt to coarsely map of the entire solution space. This also has severe drawbacks, as even small changes in the K vector could present very large changes in the system's dynamic response, driving it to instability. The granularity of the search would prove computationally exhausting.

A closed set of values would have to be chosen for each of the elements of the K vector. Even if 1000 points were chosen per n elements in the vector, then 1000^n points would have to be tested. The probability of any of those matching the desired criteria is essentially random, and

largely dependent on the bounds of the search and the granularity.

A combined approach could be made, using a coarse map as a guide, and then a gradient search starting from the “best” points in the initial map, but again, the solutions found would be localized minima.

A genetic approach is put forward in this paper, one which does begin with a few good guesses, as do the other methods, but one which allows for a wider search area guided by a stochastic process. This genetic approach has no pre-defined search area to be bounded within, and can look at several regions of the problem space simultaneously [9]. The breeding and mutation processes that follow will explain how this is possible.

Genetic algorithms are slower, but are often used in problems with a large amount of unknowns. The lower amount of structure and guidance is a benefit to the search, as it will not be easily guided towards localized minima. [17]

A new algorithm is proposed in this paper for finding the gain vector K such that the compensated system will meet the settling time and percent overshoot requirements. The basic algorithm is described first, and then specialized improvements are described in the next section. Such improvements to the algorithm are shown to increase the accuracy and decrease the computation time. As indicated in those case studies considered in this thesis.

CHAPTER 3: A GENETIC APPROACH

3.1 - Fundamentals of Genetic Algorithms

Genetic algorithms can trace their origins back to Alex Fraser, who explored the idea in 1957. [6] A genetic algorithm is a stochastic process; it is somewhat random, but the current state is not wholly independent of the previous states. There is no single way to implement a genetic algorithm, but most of these algorithms adhere to the same structure and include similar elements. [5,6] The algorithms detailed in this thesis are original work built from the basic principles which have been long established. The next few sections detail the customization and design process.

The aim of a genetic algorithm is to find a solution to a problem. Many potential solutions are generated, based on previous solutions and also on injected randomness, until one matches the exit criteria.

This requires that potential solutions (called genomes) be representable in a digital format. Further, a genetic algorithm requires a method of determining the suitability (called fitness) of each potential solution.

A genetic algorithm is characterized by a generation cycle. In this, a population of genomes is tested for fitness, and then members are selected for breeding. The breeding process creates new genomes by combining parts of other genomes. Also, to avoid converging around a localized minima or maxima, a random chance of mutation of some genomes is also included. This injection of new material seeks to prevent the genome population from stagnating, or resembling one another too much.

If breeding selection and fitness calculation favor more suitable genomes, then the algorithm will produce more and more "fit" genomes each generation until one or several solve the problem.

3.2 - Tailoring a Genetic Algorithm to Pole Placement

The main steps in tailoring a genetic algorithm to a problem are genome selection, fitness calculation, breeding method selection, and mutation selection. The genomes are the possible solutions to the problem, the fitness method is a measure of how well they solve the problem, the breeding method is how they are combined to form new solutions, and the mutation method describes how and how often new data is injected. Each of these steps will be described here

For a state space problem, the population of genomes is chosen to be feedback gain vectors (referred to in the rest of this paper as K vectors.) While it might have been possible to create a genome instead which contained the *desired poles* of the system, rather than the K vector, that approach had several drawbacks.

The poles of a system very often include one or more pairs of complex conjugate pairs. Breeding is the process of making new genomes from pieces of other genomes, which would be made more complicated because both members of a complex conjugate pair would absolutely have to end up in the same resultant genome. Further when mutation occurs, and any mutation scheme must ensure that both members of the pair were mutated in the same fashion. A system simply cannot have a complex pole and not have a matching complex conjugate pole.

The additional overhead in the breeding and mutation calculations was unfavorable, but additionally, some sets of poles cannot be achieved with K vector feedback given a specific

system. K vector selection always yields a testable, achievable system, even if that system is sometimes driven to instability.

This has several key implications:

- Each genome will be the same length (contain the same number of elements)
- Each element in the genome will be a real number
- The order of the elements within the genome has an effect on its fitness

Any breeding or mutation method must preserve those key elements.

3.2.1 An Appropriate Fitness Scheme

There are two user provided criteria for the performance of a genome in this algorithm. To limit the amount of information required from the user, no weight is given to one measure over the other. Instead, the fitness algorithm functions normally in "match" mode, seeking to get both criteria within 5% of the desired values. Other methods are provided as options, and all are discussed in detail below.

The fitness of each K vector is tested by using it to transform the system, and then measuring the new system's percent overshoot and settling time. If the fitness calculation fails (for reasons described in section 4.3) that genome is discarded without entering the population.

Breeding selection favors no particular genome. Some approaches that favored more fit parents were included in initial experimentation, but had negligible effect on the algorithm's performance. Random crossover method was chosen for breeding, mixing elements of two parents to create two new offspring simultaneously. This method was chosen because it often changes the order of elements in the resulting K vector, and thereby explores new areas of the problem space.

Mutation is applied to those offspring with a predetermined probability. When mutation occurs, it affects every element in a selected genome. A new normally distributed random number is added to each element. Some mutation methods affect a digitally representable genome in a bitwise fashion, but this was not possible in MatLab©. This normal distribution was chosen to most closely resemble a bitwise mutation. Smaller changes to the value are much more likely than large changes, and there's an equal chance of a negative or positive change. Composition of a genome and mutation procedures will be discussed in section 4.2.

CHAPTER 4: MatLab© IMPLEMENTATION

4.1 - Outline

The genetic algorithm was implemented as a function within MatLab. There were four input criteria, and two return values.

```
function [ kVector, fitnessMatrix ] =
      GeneticK( systemA, percentOS, settlingTime, modeOption )
```

The first input argument is a state-space model. The percentOS and settlingTime arguments are the desired percent overshoot of the impulse response, and the desired 2% settling time respectively. The last input argument is an optional argument to alter the way fitness is calculated. This will be discussed in detail in the fitness section.

At the completion of the algorithm, the function will provide the user with a kVector which has been selected to transform the system. That gain vector will drive the system to the performance criteria.

GeneticK will also provide a fitnessMatrix, which is a two dimensional arrangement of the evolution of the genome population. This will be discussed further in the section 4.7.

The function will return an error message if the first three input arguments are not provided, but does not require the fourth argument. The default fitness mode will drive the system towards an exact match of the user specified criteria.

The outline of the algorithm is as follows:

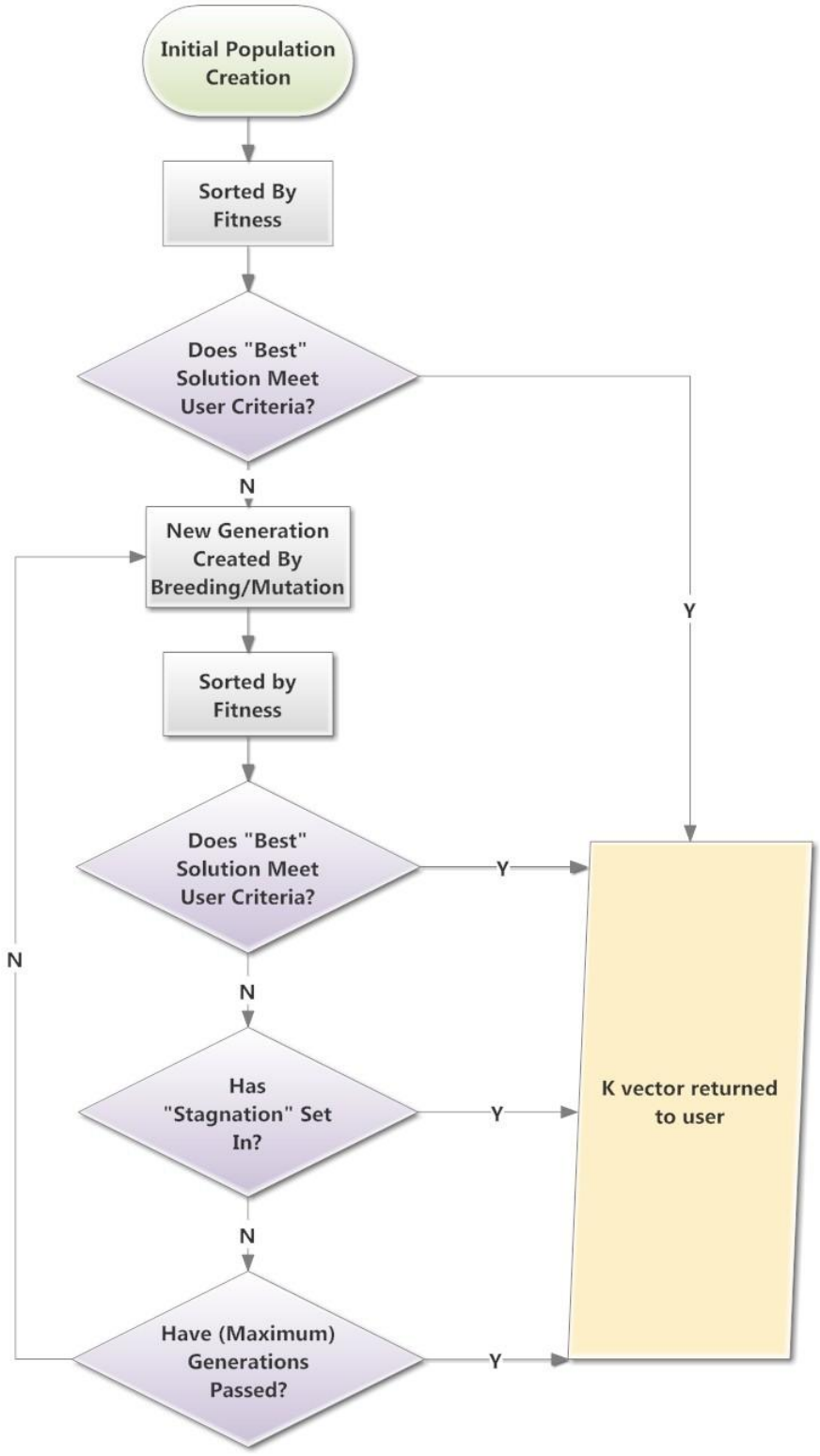

Figure 9: Block Diagram of Genetic Algorithm

4.2 - Genome Creation

The first task is to create an initial population of genomes. A genome cannot be added in to the population unless it can pass a fitness test. The creation and testing process repeats until the initial population is full.

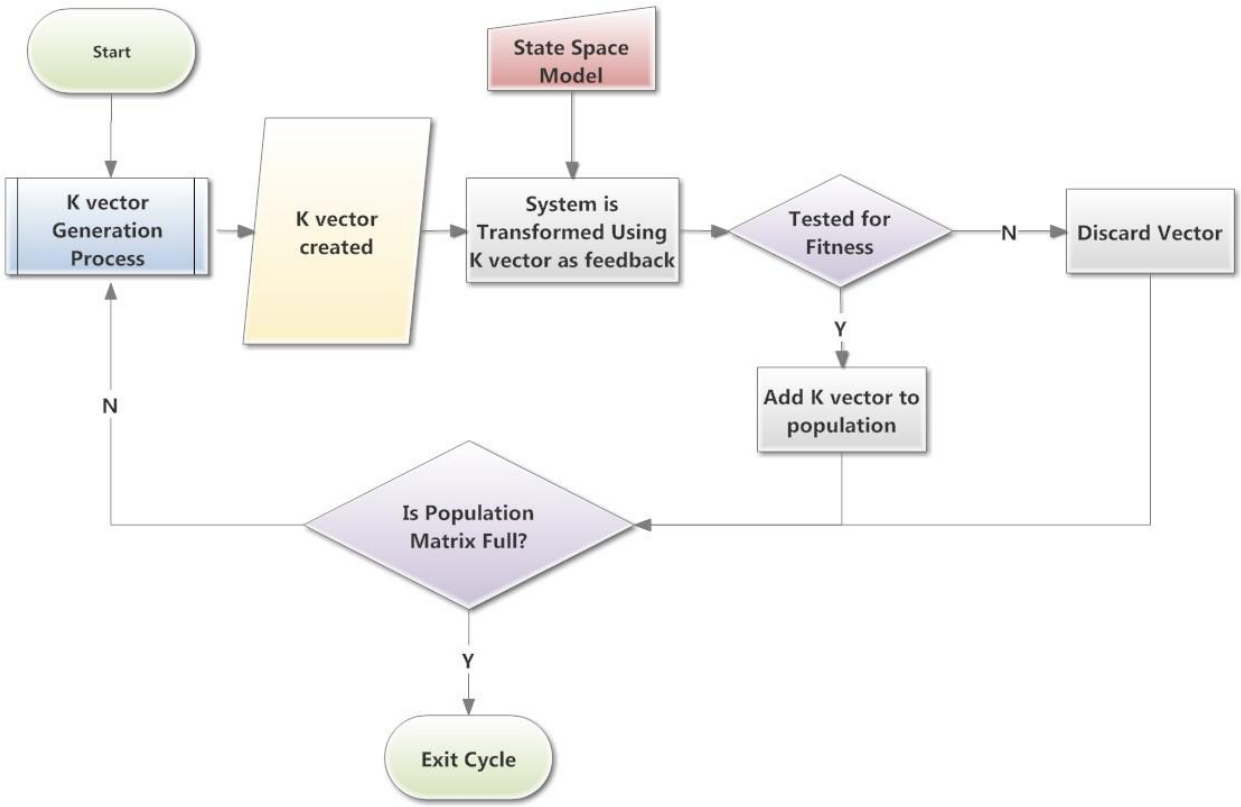

Figure 10: Initial Population Creation

Each potential K vector is created and then tested for fitness before being added to the population. First this thesis will look at the genome creation process, and then move on to discuss how fitness is calculated.

The user will have already provided a desired percent overshoot and a desired settling time. Those arguments are used to calculate a complex conjugate pair, according to second order dynamics.

If the system had only these two complex conjugates as poles, and had no zeroes, this would be sufficient to drive the desired behavior. This algorithm is meant to tackle a broader range of problems, but will use this base pair as a starting point.

The next step is to use MatLab's "place" command to find a K vector that will transform the system to have a desired set of poles. A state-space model has a number of poles (n) equal to the dimensions of the A matrix, and therefore n poles must be selected before the "place" command is called. In a system of dimensions greater than two, the complex conjugate pair serves as the first two desired poles, and the remaining are randomly generated non-positive real values.

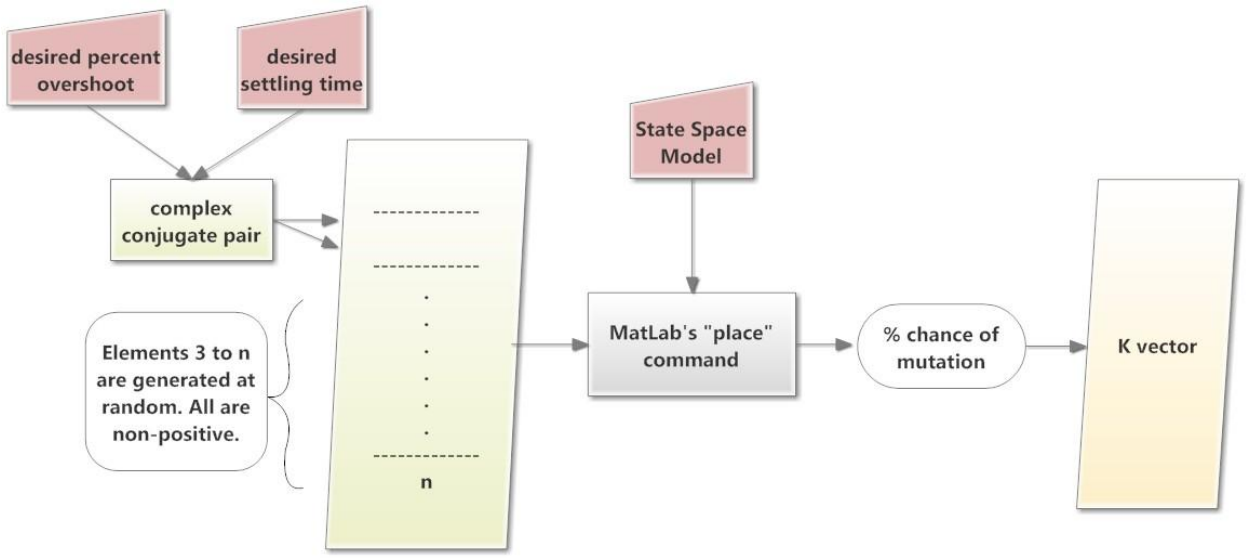

Figure 11: Initial Genome Creation

All of the poles (except the first two) are generated randomly for each newly created genome to ensure a diverse initial population. The algorithm chooses negative values which are at or around 10 times the magnitude of the complex conjugate pair.

In the special case that the state-space model is of dimension 2, only one set of desired poles can be created with the place command. Therefore, every K vector produced by the place command in this case is subjected to mutation before it proceeds to the next step.

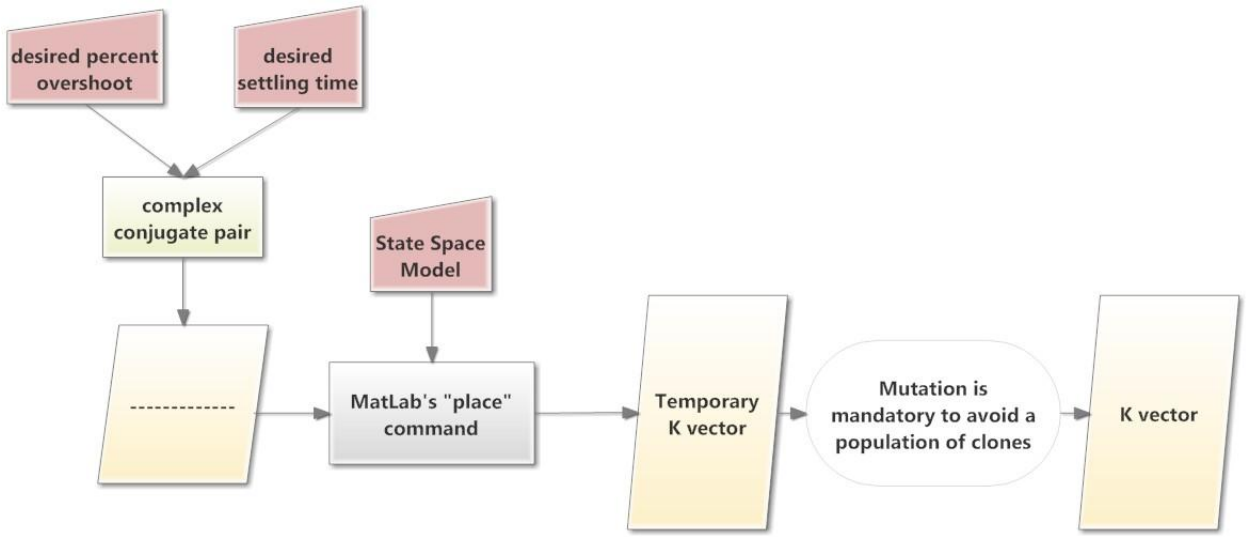

Figure 12: Genome Creation in a 2 x 2 State Space Model

4.3 - Fitness Calculation

Once a genome has been created, it must be tested to see what effect it will have on the system. It is used to transform the system (as seen earlier in section I) and the resultant system is tested for performance.

This same test is used a fitness score needs to be calculated, including for those genomes created through breeding and mutation cycles.

As a check before fitness is calculated, the eigenvalues of the transformed system's A matrix are calculated. If any of those eigenvalues are positive, the genome is discarded. Positive poles in the system indicate instability.

The "step" command is used to gauge the system's response. The "stepinfo" command is used to gather the data and extrapolate the performance features for the fitness calculation. How the K vector scores is dependent on the "modeOption" the user selected. As mentioned earlier, the default mode is "Match" mode.

All four modes are listed here. It is important to note that in all four cases, a lower fitness score is chosen to indicate a better match. The calculations are set up so that any fitness score of 1 or lower is within tolerance, and the algorithm will exit at the end of any breeding cycle where the best genome has a fitness lower than 1.

Even though "undershoot" is not a user supplied criteria, undershoot is indicative of ringing, and is assumed to be undesired. Undershoot is the amount a system will respond in the opposite direction before moving towards the desired state. For this reason, any measured undershoot value is multiplied by 10 and added to the rest of the fitness calculation. Recall that a higher fitness score is given to a "less" fit genome, so any undershoot will have a large effect on the fitness score.

4.3.1 - MATCH Method

MATCH is the default fitness calculation method. The calculation is designed to find a K vector that will drive the state-space model to within 5% of both user provided criteria. In order

to match both criteria, a score is given to each (overshoot and settling time) and the maximum between those is given to the genome.

$$fitness =$$

$$max\left(\left|\frac{measured\ overshoot - desired\ overshoot}{desired\ overshoot * 0.05}\right| \ , \ \left|\frac{measured\ settling\ time - desired\ settling\ time}{desired\ settling\ time * 0.05}\right|\right)$$

$$+10 * measured\ undershoot$$

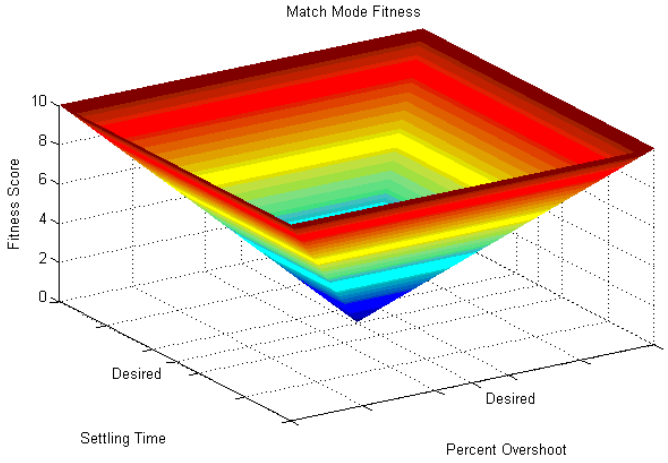

Figure 13: Match Mode Fitness

The lowest fitness scores are at the intersection of the desired Percent Overshoot and desired Settling Time. The fitness score can go as low as zero, but will exit anywhere in the region where the fitness score is below one.

4.3.2 - WEAK MATCH Method

WEAK MATCH also seeks to come within 5% of the user specified criteria. The reason it is called "weak" is that it does not necessarily return a K vector that drives the state-space model to both criteria. If one of the performance criteria is matched very well, (to within 1% for

example) the other criteria might only need to match to within 8% for the algorithm to exit as successful.

The less stringent fitness method will conclude faster, but often not as accurate as the "match" method.

$$fitness = \frac{|measured\ overshoot + measured\ settling\ time - desired\ overshoot - desired\ settling\ time|}{(desired\ overshoot + desired\ settling\ time) * 0.05} + 10 * measured\ undershoot$$

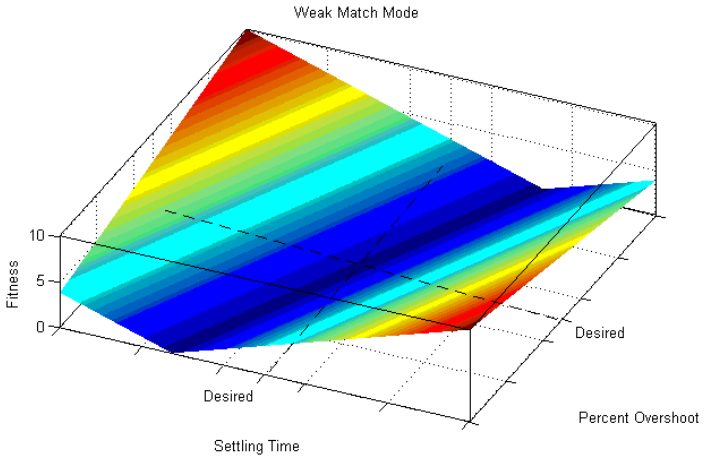

Figure 14: Weak Match Mode Fitness

Fitness is equally good in a channel that cuts diagonally across the plane. In this way, if Settling Time is higher than desired, but Percent Overshoot is lower than desired, the algorithm reports a good score.

4.3.3 - THRESHOLD Method

THRESHOLD uses the user criteria as a set of thresholds and finds a K vector that will transform the state-space model to do as well or better than both criteria. Since there are two

criteria for fitness, and fitness itself needs to be a single value, the maximum between the two criteria is used.

For instance, if the desired percent overshoot was 12% and the desired settling time was 5s the algorithm would not exit if it found a K vector that drove the state-space model to 2% overshoot if that same K vector drove the settling time to 9s. Good performance in one of the criteria is insufficient.

Initial testing of a similar method showed that an unfair bias was given to systems that did not register overshoot. In these systems, the measured response would have a comparatively large rise time (larger than the 2% settling time) and measure zero overshoot. The fitness calculation would rate these systems with very low fitness scores, and those genomes would dominate the population.

To avoid a flood of overdamped systems, the fitness calculation is changed slightly when a system is overdamped.

$$if\ (\ rise\ time\ <\ settling\ time\ )$$

$$fitness = max\left(\frac{measured\ overshoot}{desired\ overshoot}\ ,\ \frac{measured\ settling\ time}{desired\ settling\ time}\right)$$

$$+10 * measured\ undershoot$$

$$else$$

$$fitness = max\left(\frac{measured\ rise\ time}{desired\ settling\ time}\ ,\ \frac{measured\ settling\ time}{desired\ settling\ time}\right)$$

$$+10 * measured\ undershoot$$

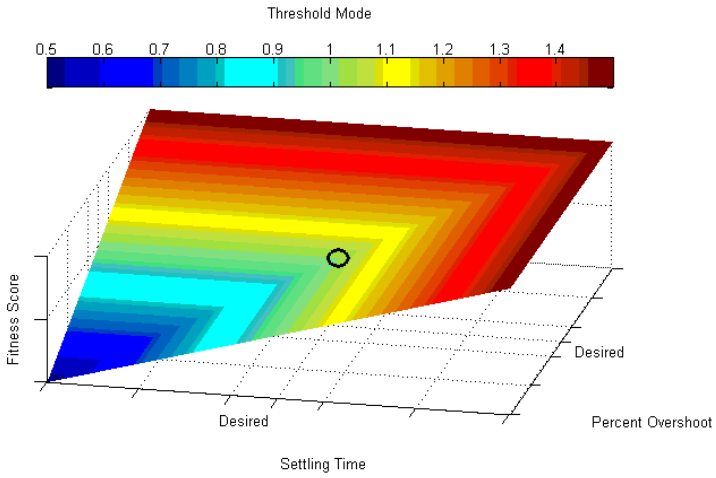

Figure 15: Threshold Mode Fitness

The point on the map where desired Percent Overshoot and desired Settling Time intersect is highlighted with an oval. It can be seen that the fitness at that point is a 1. If Settling Time or Percent Overshoot are increased, the fitness also increases. If either criteria decrease, the fitness score decreases.

4.3.4 - WEAK THRESHOLD Method

WEAK THRESHOLD uses the user criteria as a threshold and tries to guide the state-space model to do as well or better than asked. This "weak" version is designed to be a faster, less strict search. Both the desired percent overshoot and the desired settling time are used, but poor performance in one dimension can be compensated with good performance in another.

For instance, if the desired percent overshoot was 12%, and the desired settling time was 5s, the algorithm might exit when it found a K vector that drove the state-space model to 2% overshoot, even if the settling time was 9s.

This "weak threshold" method still suffers from the same weakness as the regular "threshold" method, in that considerations must be made for a system that has no overshoot whatsoever. Unless the user desired overshoot is also zero, the fitness calculation unfairly biases towards genomes that provide overdamped solutions.

$$if\ (\ rise\ time\ <\ settling\ time\ )$$

$$fitness = \frac{measured\ overshoot + measured\ settling\ time}{desired\ overshoot + desired\ settling\ time} + 10 * measured\ undershoot$$

$$else$$

$$fitness = \frac{measured\ rise\ time + measured\ settling\ time}{desired\ settling\ time * 2} + 10 * measured\ undershoot$$

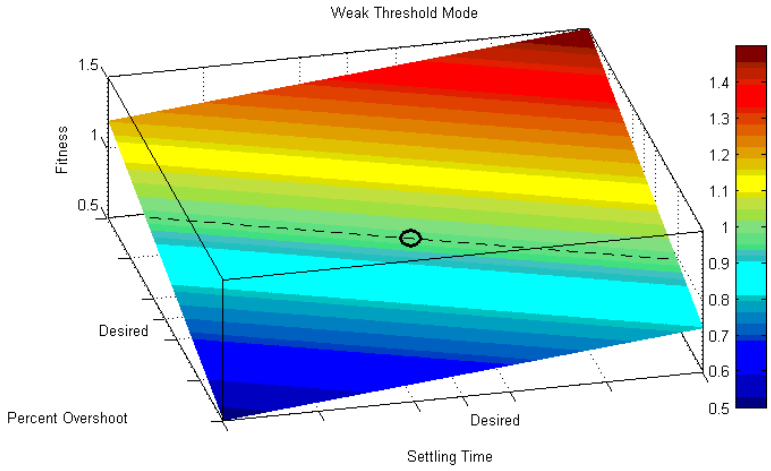

Figure 16: Weak Threshold Mode Fitness

The point where desired Percent Overshoot and desired Settling Time meet is highlighted with an oval in the figure above. The dashed line represents a fitness of 1, which is good enough to exit. It can be seen that Settling Time can be greater than desired and the fitness score can still be lower than 1 if the Percent Overshoot is significantly less than desired.

4.4 - Comparison of Fitness Methods

Simple visual comparison of a few fitness methods are given here:

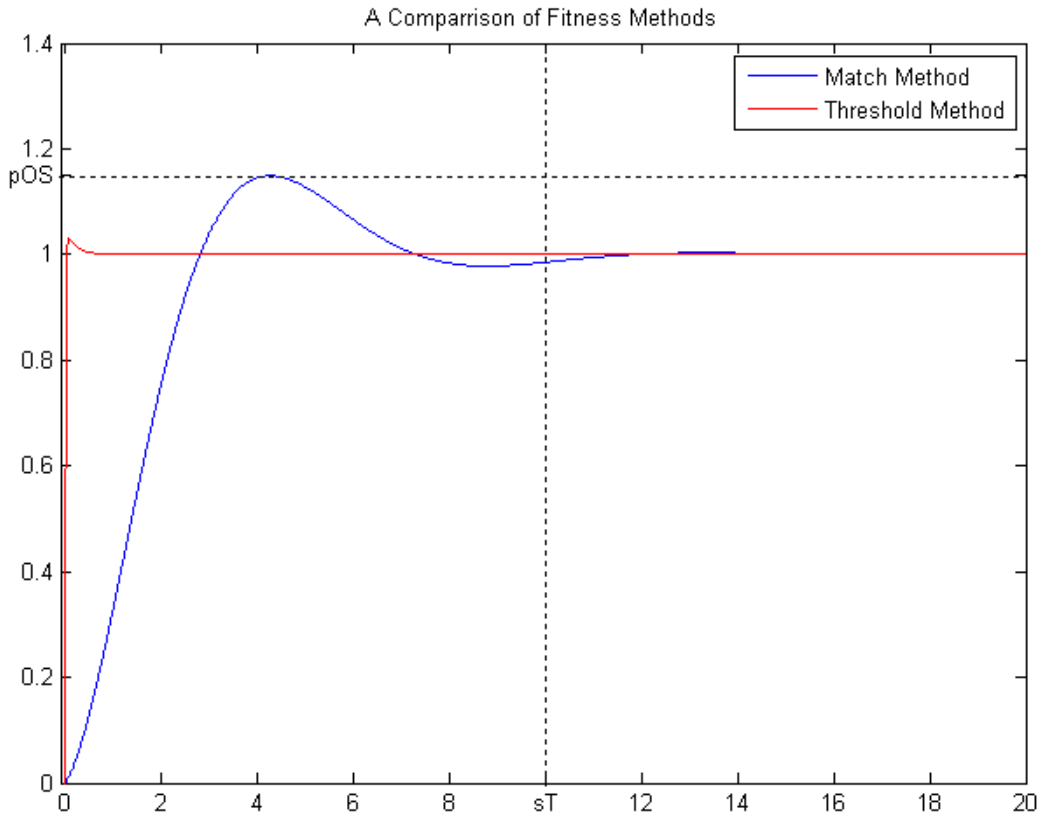

Figure 17: Match Fitness Result versus Threshold Fitness Result

One of the systems was run in both Match and Threshold mode, and the results are displayed above in figure 17. Match mode is shown to seek a result that exactly reproduces the user given criteria, with a maximum value touching the desired "pOS" value, and settling in to a steady output just at the time "sT." Threshold mode seeks the smallest settling time and percent overshoot available. The Threshold result has a returned significantly lower overshoot and settling time than the Match result.

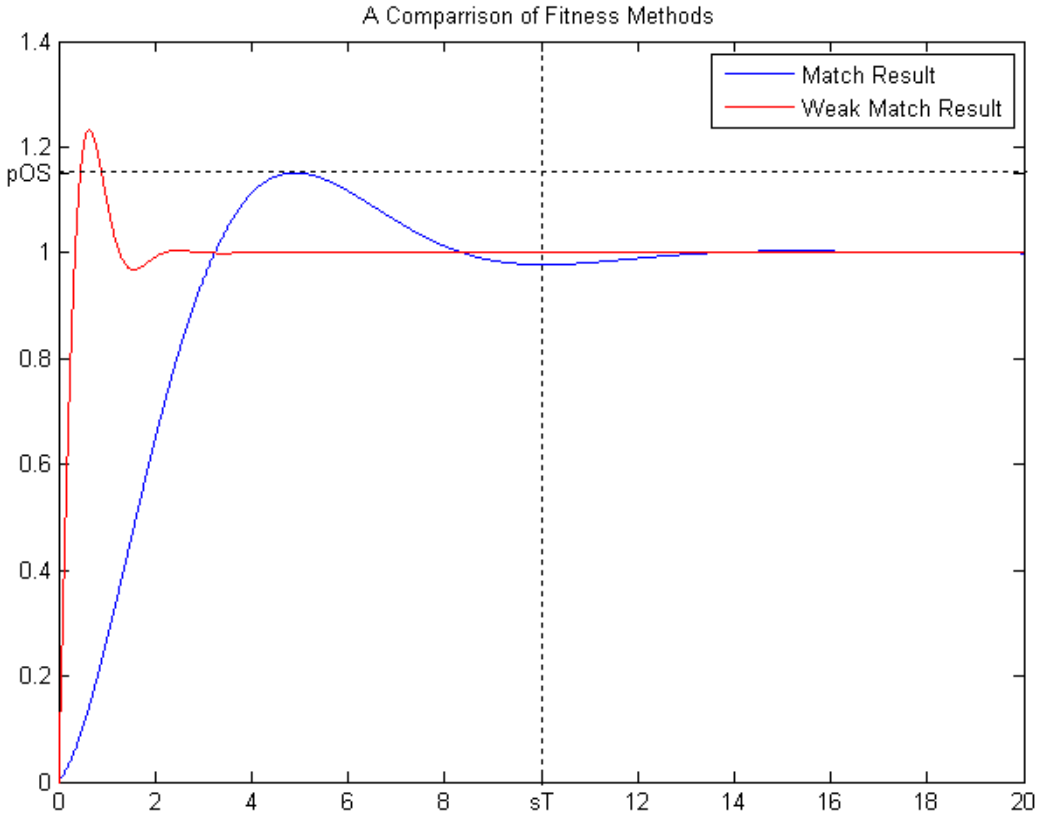

Figure 18: Match Fitness versus Weak Match Fitness

Similarly, figure 18 compares Match and Weak match results. Again, Match mode strives to reproduce both user specified criteria. Weak Match mode will exit when the sum of the two observed behaviors is equal to the sum of the two desired behaviors. Weak Match will try to meet both criteria, but will allow for a tradeoff. In the above example, the percent overshoot is slightly higher than desired, but the settling time is significantly lower. This is a good result for Weak Match mode.

Each of the four methods above will combine different performance criteria into a single fitness score. Several novel methods have been proposed to incorporate multiple dimensions of fitness into a genetic model. [12,16,18] These are left for the reader to investigate, but in

summary, these are not appropriate for this genome search. The population must be organized in a linear fashion, and the four available fitness methods provided allow for enough diversity.

4.5 - Breeding

After a sufficient number of genomes have been created, population creation ends and the current population matrix is sorted by fitness. If the "best" genome already exceeds fitness expectations, the algorithm can exit right away. In the more likely case, the algorithm enters the breeding cycle next.

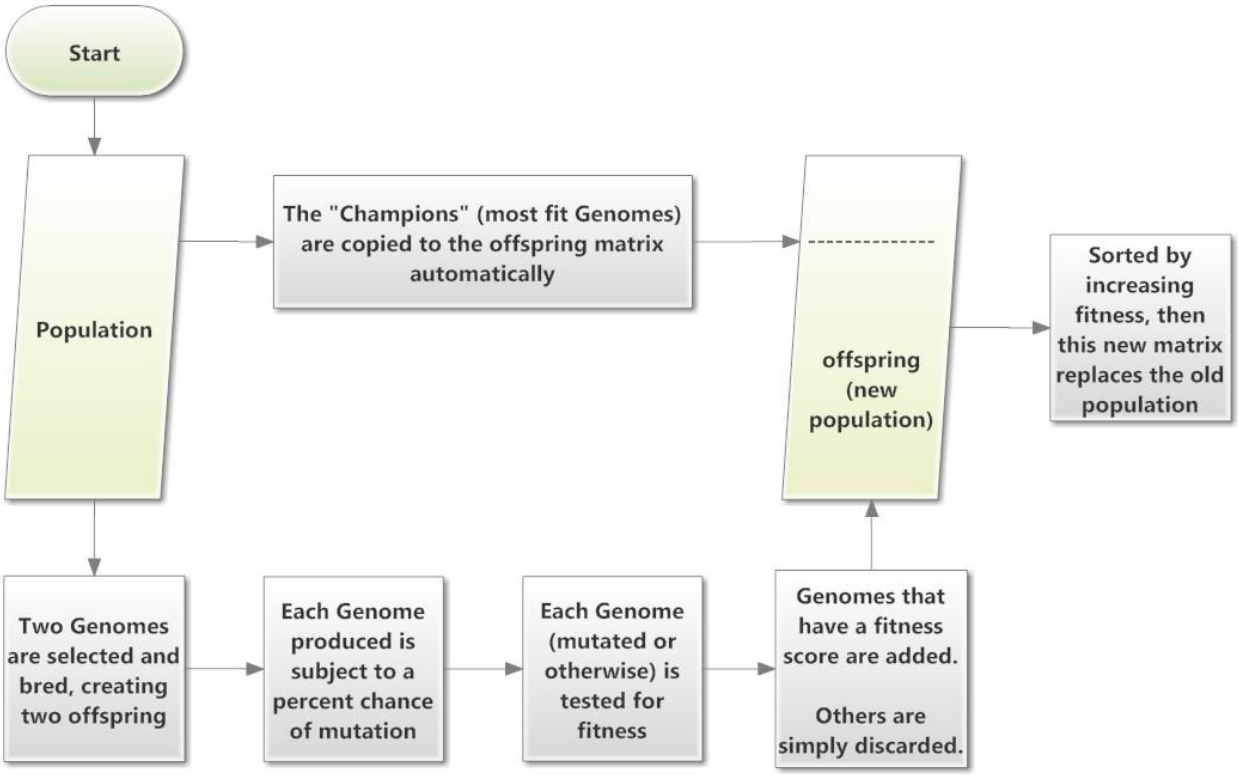

Figure 19: Breeding Cycle

The object of the breeding cycle is to create a new matrix of offspring based on the previous population. To ensure that the best fit solutions remain in the breeding pool, a certain percentage is copied over directly as "champions." They undergo no alteration or mutation.

To create the rest of the offspring, two parents are chosen at random from the entirety of the old population. This includes the "champions," because a copy of each of them remains in the old generation. The selected parents are bred through "random crossover" in which randomly selected elements from both parents are swapped to form two offspring.

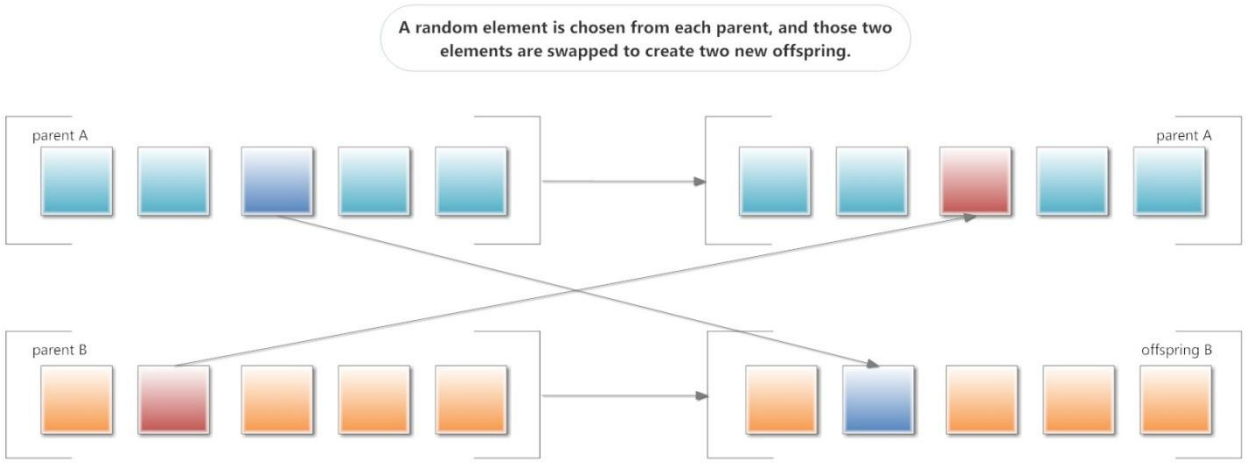

Figure 20: Random Crossover

This breeding method was chosen to preserve the same number of elements in the offspring genomes as the parent genomes, and also to produce two offspring genomes from each pair of parents.

As the algorithm moves from one generation to the next, preserving champions and breeding can drive a population towards stagnation, meaning that large portions of the population are roughly the same. This process would be sped up if the crossover method was not random.

If two parents were near to each other, swapping the third element from one parent with the third element of the other would produce two children that are virtually identical to their parents. However, the correlation between different elements in a genome is much weaker. [10] Swapping unrelated elements in two genomes is more likely to produce new results and add back some variety. [16]

4.6 - Mutation

Each offspring is individually subject to a percent chance of mutation. Mutation seeks to prevent the search from settling around local minima by injecting some randomness. Even though the genomes are represented as digital data, MatLab© stores them as floating point values. Bitwise mutation (randomly toggling the binary code that represents the elements of the genome) is outside the scope of this paper.

For this reason, a normally distributed value is added to each element of the genome instead. The mean of the distribution is zero, so it is equally likely that the mutation will add or subtract from the initial value.

Whether the offspring genome is mutated or not, it is tested next for fitness before being added to the offspring matrix. If the calculation fails for any reason, the offspring is discarded. This breeding cycle is repeated, randomly selecting two new parents each time, until the offspring matrix is as large as the old population matrix. Then the offspring are re-sorted in order of fitness, and that matrix entirely replaces the old population.

4.7 - Algorithm Exit Conditions

There are three exit conditions for the algorithm. If, at the end of a fitness sort, the best

solution has a fitness score less than one, the algorithm will exit and return that genome, that K vector to the user.

Additionally, if 200 generations have passed and such a solution has not been found, the algorithm will exit and return the most fit in the current population.

Lastly, each generation after 10, the algorithm will check the best fitness score from previous generations. If the current generation doesn't show at least a 7% improvement from the previous 10, the algorithm exits and returns the best fit K vector to the user.

4.7.1 - Exit Reporting

Although the K vector and the fitnessMatrix are the only two objects returned to the user, several pieces of information are displayed to the screen. The final K vector is provided, along with the data collected from the fitness test for that genome. This includes the measured percent overshoot, measured settling time, rise time, and undershoot. These are provided alongside the user criteria for comparison.

Additionally, the initially calculated complex conjugate pair is provided alongside the final system poles.

A tally of the number of genomes that failed fitness tests and were discarded is provided to the user, unless a variable within the program (called "verbose") is set to zero.

```
Best Fit K Vector is:  [ 25.1033 22.5265 4.9489 ]

Fitness is:            0.229554

Complex Conjugate Pair: [-0.869338445650699+i*1.40351150111699

-0.869338445650699-i*1.40351150111699]

System Poles          [-7.84957039485506;

-1.54966454075136+i*0.892519343170043;-1.54966454075136-i*0.892519343170043]

Percent Overshoot     = 0.413358

Desired Overshoot     = 7

Settling Time         = 2.58024

Desired Settling Time = 4.5

Undershoot            = -0

Rise Time             = 1.56389

--- --- --- ---

46  unstable solutions discarded for instability

0  unstable solutions discarded for poor Step results
```

Figure 21: Example Exit Reporting

Lastly, two figures are created and saved. One of them tracks the fitness value for the most fit genome in each generation. The other displays the step response of the transformed system.

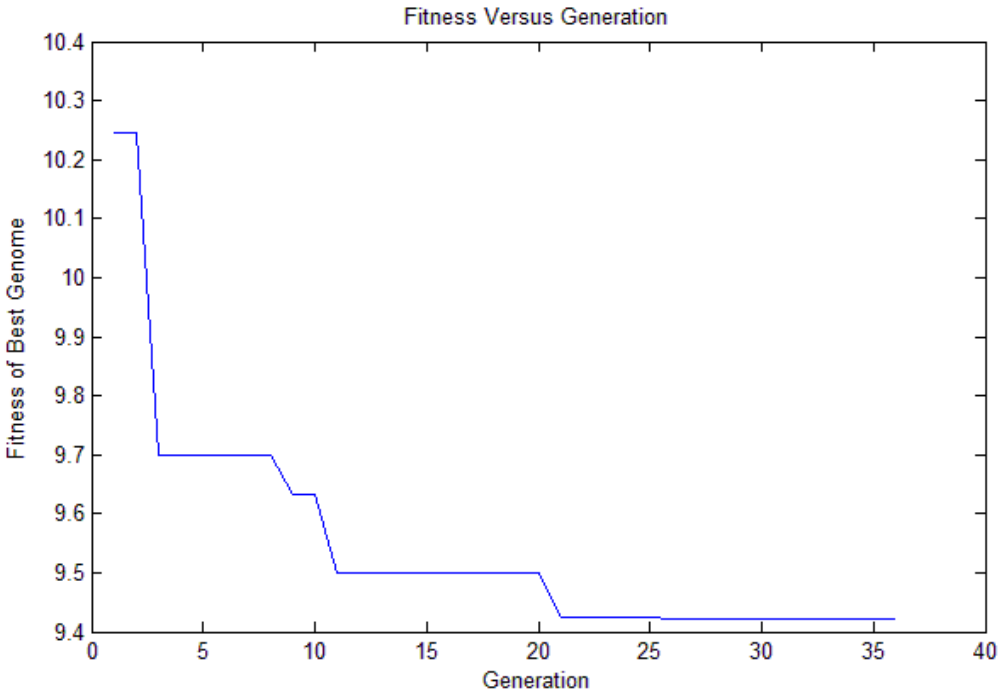

Figure 22: Example graph of "most fit" genome per generation

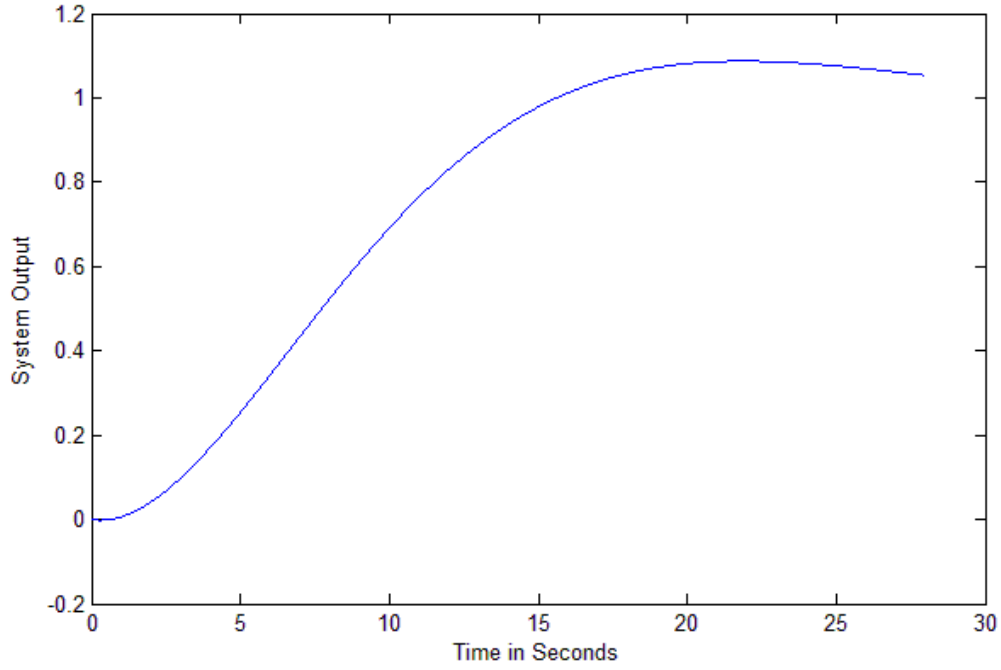

Figure 23: Example graph of transformed system performance

4.7.2 - Fitness Matrix

For research, at the end of each breeding cycle the fitness scores of each generation are stored in a matrix. Each row of the fitness matrix stores the sorted fitness scores of the entire generation. The final dimensions for the fitness matrix are (number of generations x population size). This is returned to the user as fitnessMatrix. The genomes themselves are not preserved in this matrix, only the fitness scores they produced, but the mapping can provide insight into the algorithm's progress.

An example is given below.

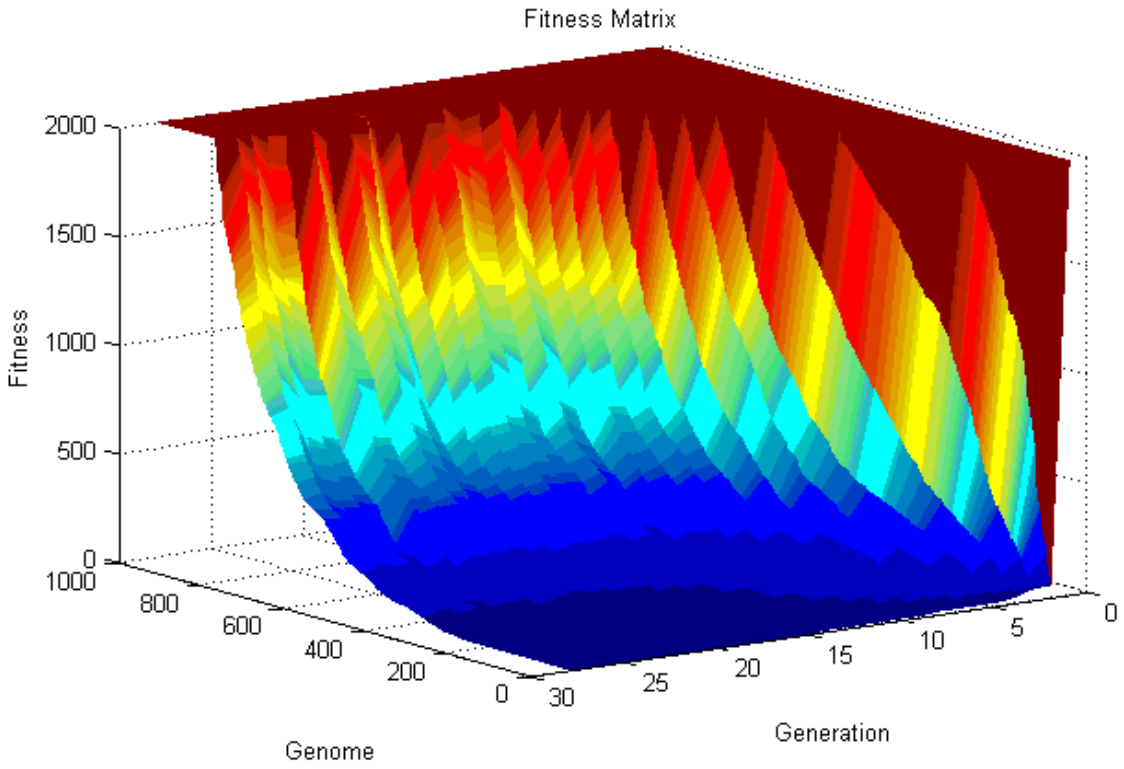

Figure 24: Example of the Fitness Matrix

This fitness matrix demonstrates that for the 28 generations that the algorithm searched,

each successive generation had more members who scored 500 fitness or lower. A steady decrease in the number of large fitness valued genomes indicates that the population is converging around more fit solutions.

CHAPTER 5: Island Approach and Improvements

5.1 - Overview of Changes

The island approach is largely similar to the approach described and implemented above. It is a separate MatLab© function with the same input arguments and the same types of data returned.

```
function [ kVector, fitnessMatrix ] =
      GeneticKIsland( systemA, percentOS, settlingTime, modeOption )
```

The purpose of implementing the island approach is to decrease the runtime of the algorithm. Breeding and fitness calculations are computationally intensive, and must be done a large number of times in each generation. In an island approach, the genetic population is divided up into sub-populations, called islands, which breed locally.

If this task is distributed to multiple computers, each computer can manage all of the breeding, mutation, and fitness calculations of one island at the same time several other computers manage the other islands. Though the islands will maintain separate breeding pools, they are not completely isolated from one another. Between breeding cycles the best solutions from each island are forced to migrate to neighboring islands so that good genomes continue to be favored.

The basic block diagram is as follows:

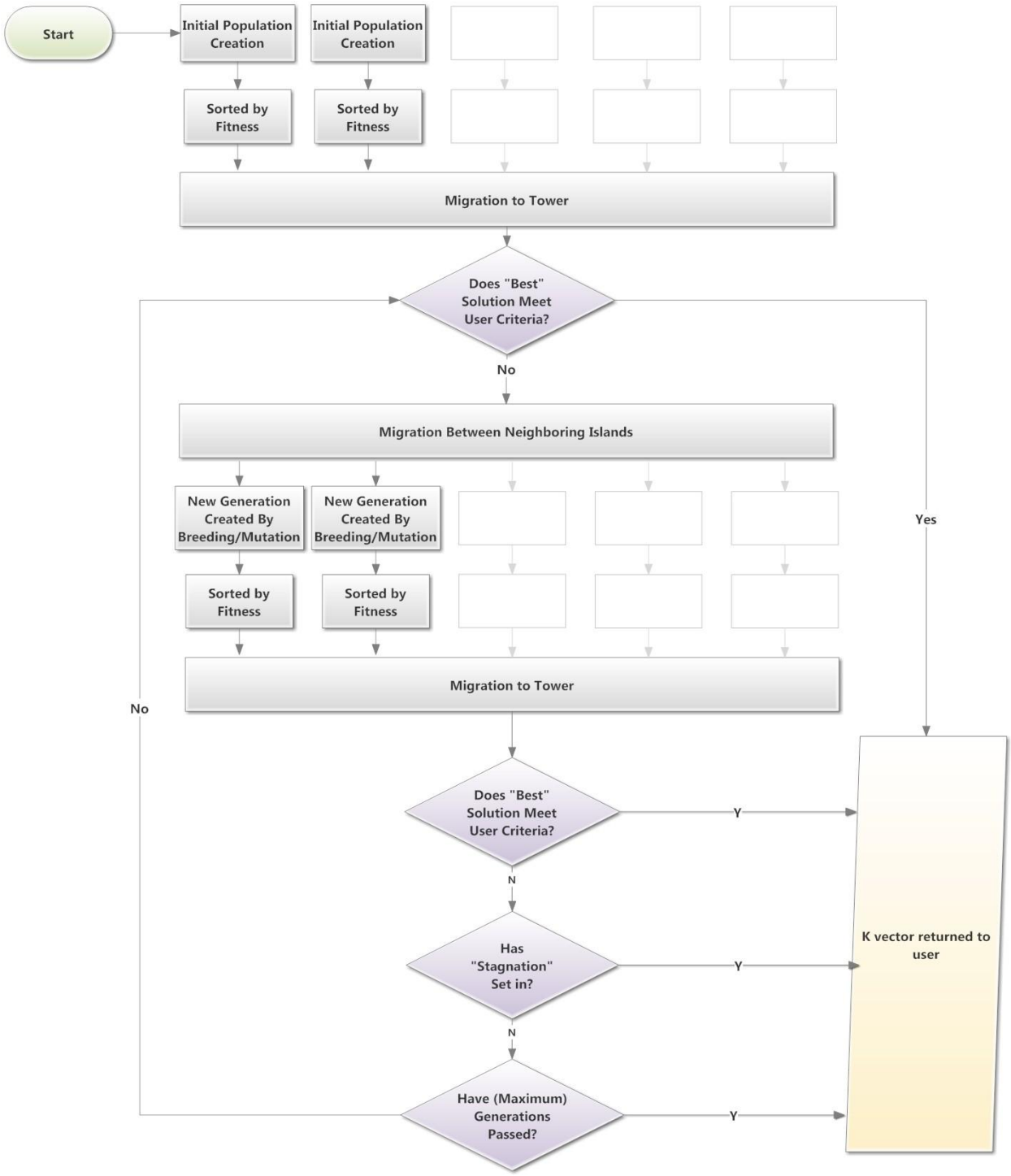

Figure 25: Island Approach Block Diagram

Because each island breeds with only a sub-set of the total population, that island's population experience some specialization, breeding results that differ from other island

populations more and more as each generation passes. Migrating the best solutions between islands (discussed the section 5.3) will keep a concentrated supply of the best solutions flowing, but for this problem, an effort has been made to capitalize on each island's potential for specialization.

5.2 - Orthogonal Mutation

One island is created for each element in the (K vector) genome. Whenever a mutation occurs, each island will handle that mutation process slightly differently. Each island is numbered and corresponds to a single element in the K vector. The numbered islands will mutate the corresponding element in the genome to a much greater degree than the rest.

In essence, it is believed that each island will be specialized to search along one axis of the problem space, one element of the genome. Solutions which are fit will migrate to other islands, specialized islands, where they might become more fit through that specialized mutation.

This specialization begins with the initial population creation. The total population is divided evenly between each island. The creation process for each genome is the same as discussed earlier in the regular genetic algorithm, except that if mutation occurs, it is tailored according to which island the genome is being created for.

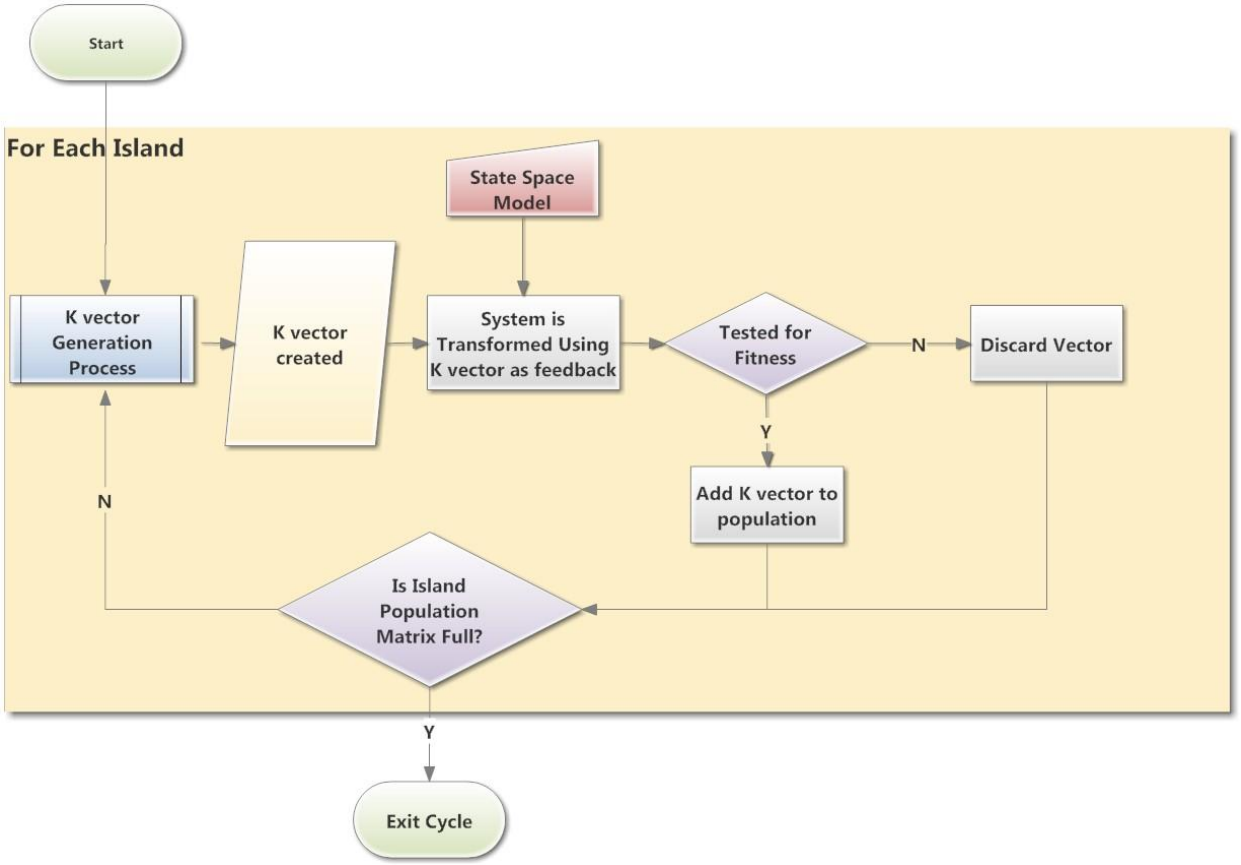

Figure 26: Island Population Creation

The initial genomes are subject to a random chance of mutation, unless the system is of size two. In that case each genome is subject to mandatory mutation to assure diversity within the genome population.

To facilitate specialization, each island mutates differently than the rest. An example is given here:

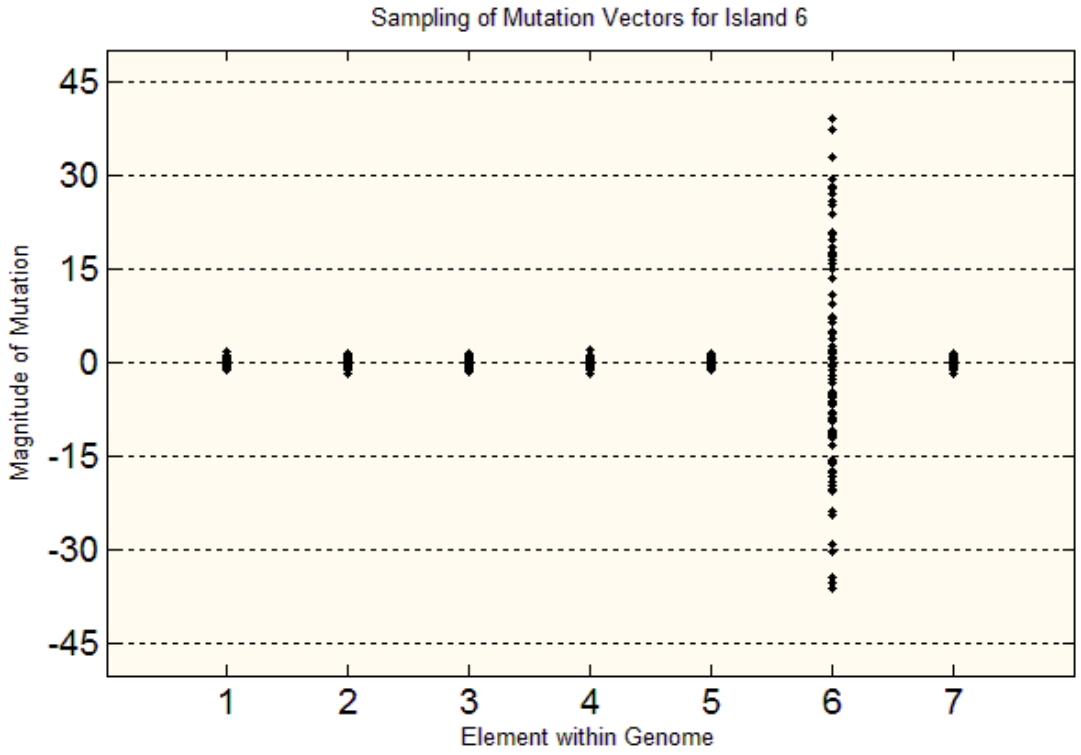

Figure 27: Collection of 100 mutation vectors, demonstrating range

For this figure, 100 mutation vectors are generated for a 7 element case, where the genome is in the population for island 6. The data shows that the sixth element (corresponding to the island number) is mutated over a wide spectrum of values, whereas the other six values do not change as much during mutation.

This same mutation process is used throughout the algorithm whenever mutation is necessary, including during the breeding process. This difference in mutation, and the effects caused by this inclusion, are the fundamental purposes of this thesis.

After creation, the sub-population at each island is sorted by fitness. The method of fitness calculation is unchanged from the normal genetic algorithm explained earlier. Lower fitness scores still equate to more fit solutions.

5.3 – The Addition of Migration

One of the major differences in the island approach is the migration of "most fit" genomes. The first migration occurs just after island creation, when a small sub-set of genomes from each island is sent to a central tower. After this migration process, the tower is checked for a genome that meets the criteria. If it exists, the algorithm exits. In the more likely case, the algorithm moves on to inter-island migration.

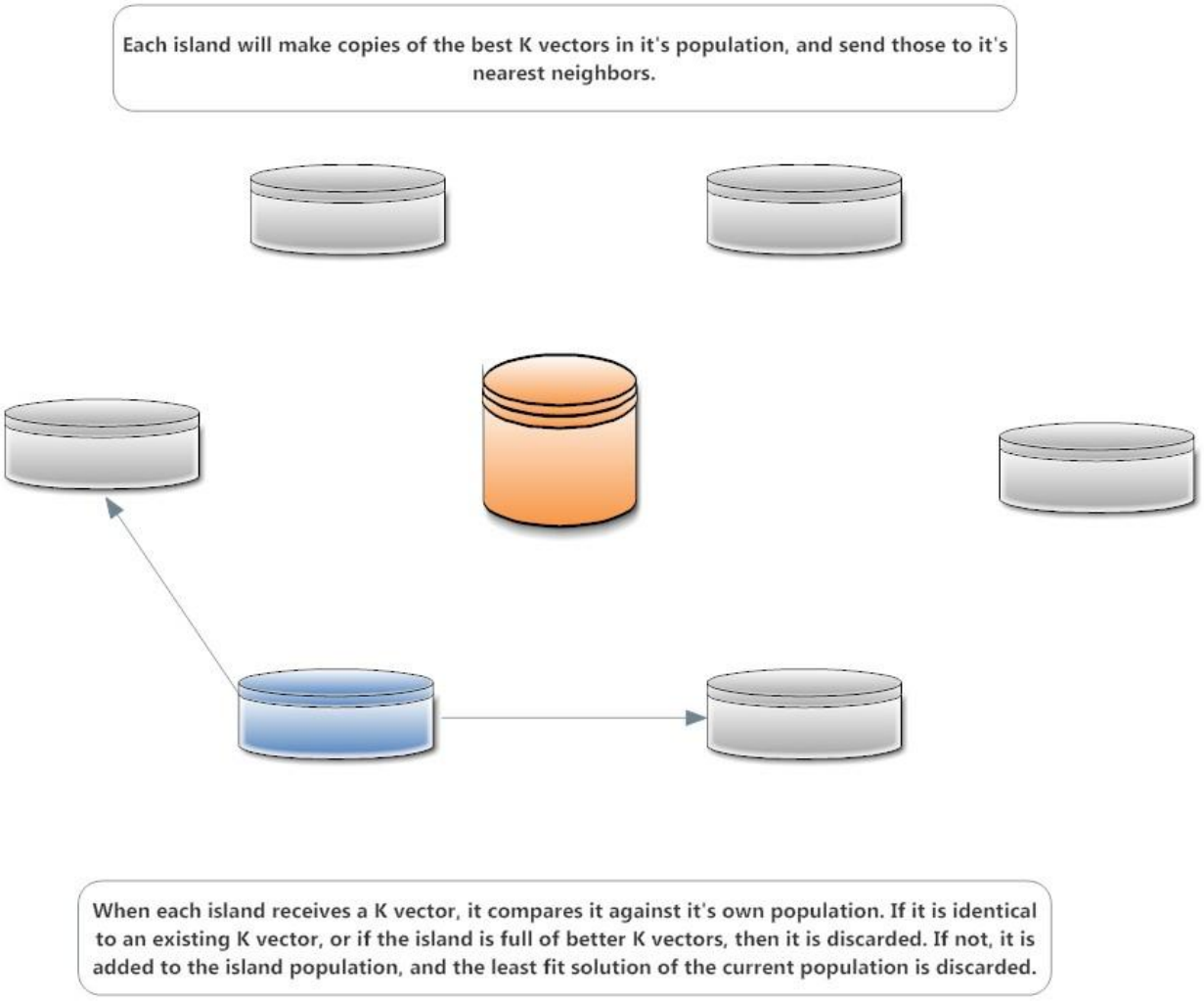

Figure 28: Migration from one island to its neighbors

Only the best few genomes are migrated to the nearest neighbors. Migration is limited to

nearest neighbors to save time. If a genome performs well, when it migrates to a near neighbor, it is very likely it will be in the top percent of that new population, where it will qualify for migration to the next nearest neighbor in the next cycle and so forth. The very best genomes will make their way to all of the other islands in just a few generations. [19]

To keep a constant, centralized list of fit genomes, migration also occurs to a central tower. This migration is one way, and occurs at the end of every breeding cycle.

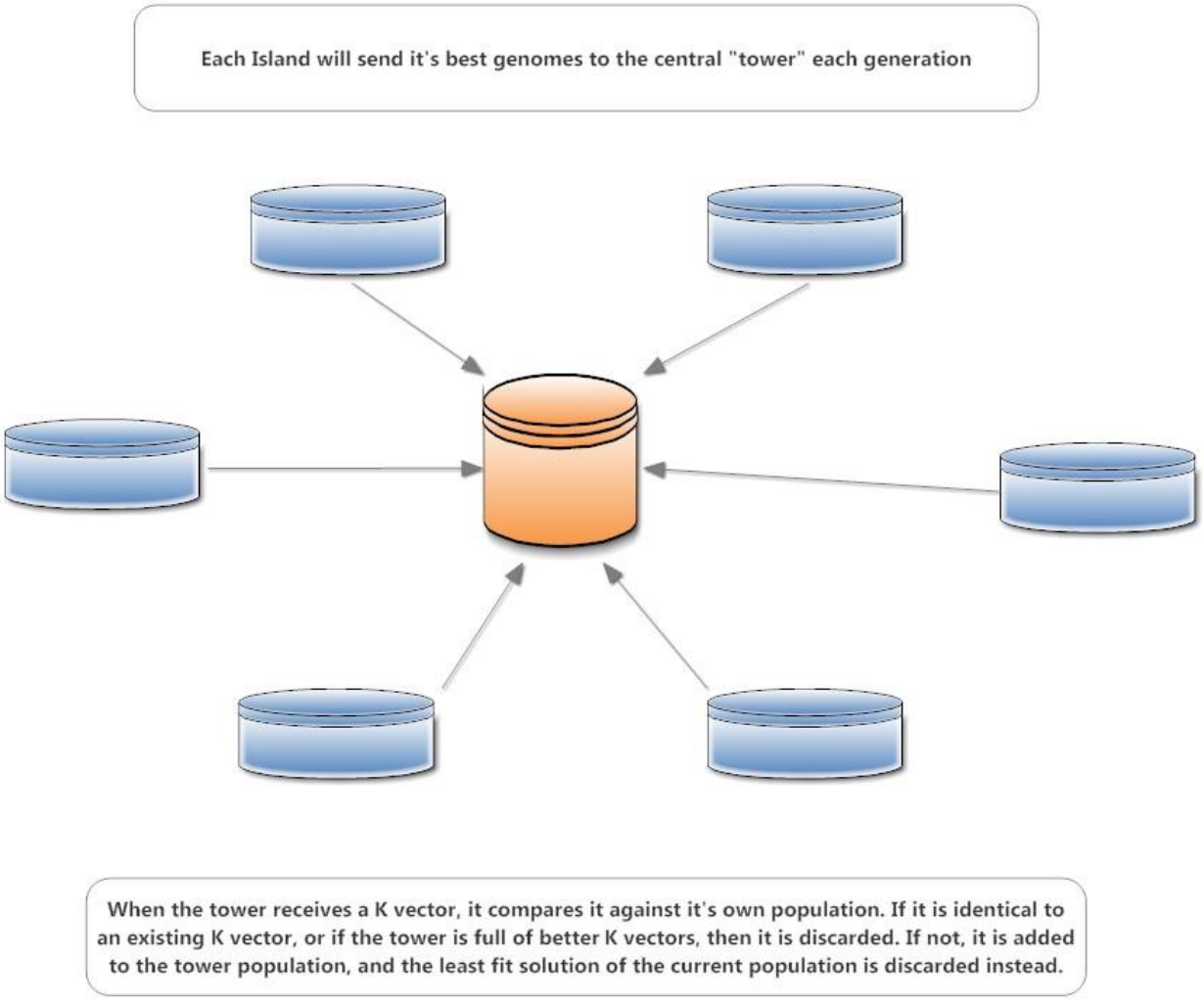

Figure 29: Migration to a central tower

The process of receiving migrating genomes is fairly straightforward. Incoming genomes

are compared with existing genomes, starting with the least fit genomes in the target population. The incoming genome moves up the target population until a genome is found within the target population that has the same fitness score or better. If the incoming genome is not identical to the target genome, then it is added to the population.

For this implementation, no breeding or mutation takes place in the tower. Many potential improvements to the tower are discussed in the future work section.

These migration patterns were chosen to decrease the amount of time taken to handle migration. With this approach, each island only needs to send genomes three times and receive genomes twice, rather than sending and receiving n times, where n is the number of islands in the population.

CHAPTER 6: Testing

6.1 - Test Procedure

An automated test was chosen to compare the regular genetic approach and the island genetic approach. Multiple systems were tested.

For each system, a set of desired performance criteria were chosen. Then each algorithm was run 80 times. For the first twenty, "Weak Threshold" performance was chosen. For the next twenty, "Threshold," then "Weak Match," and finally "Match."

All of the resulting K vectors were recorded, along with their performance criteria, fitness scores, graphs of the transformed system response, and the fitness matrix of each run.

6.2 - Descriptions of Systems Tested

The systems used ranged from two variable systems to seven variable systems. All of the systems, along with their pole-zero maps, are shown in the following pages.

Case Study 1: System 2g

$$A = \begin{bmatrix} 3 & -5 \\ 0 & -1 \end{bmatrix}$$

State Space Model for System 2g

$$B = \begin{bmatrix} 2 \\ 0.1 \end{bmatrix}$$

$$C = [0 \quad 1]$$

$$D = [0]$$

Transfer Function for System 2g: $H(s) = \frac{-1}{s+1}$

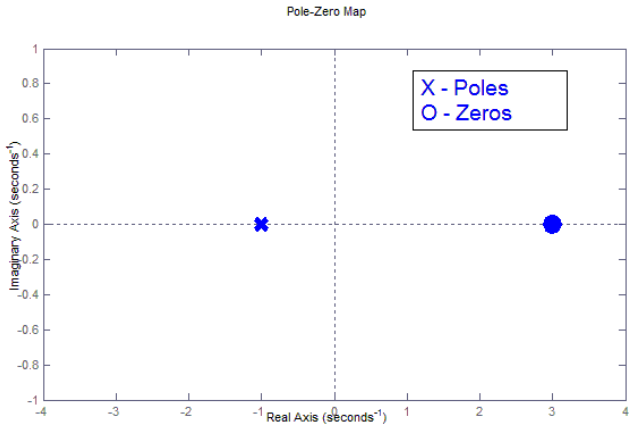

Figure 30: Pole Zero Map for System 2g

System 2g has two poles, one in the right half plane. There is also one zero which is concurrent with that pole. That pole and zero cancel one another out in the transfer function. For all test cases involving 2g, the Desired Overshoot was 7.5%, and the Desired Settling Time was set to 14s.

Case Study 2: System 2l

$$A = \begin{bmatrix} 1.024 & -2.318 \\ 4.287 & 1.185 \end{bmatrix}$$

State Space Model for System 2l

$$B = \begin{bmatrix} 6 \\ 0 \end{bmatrix}$$

$$C = [0 \quad -1]$$

$$D = [0]$$

Transfer Function for System 2l:

$$H(s) = \frac{-25.72}{s^2 - 2.209s + 11.15}$$

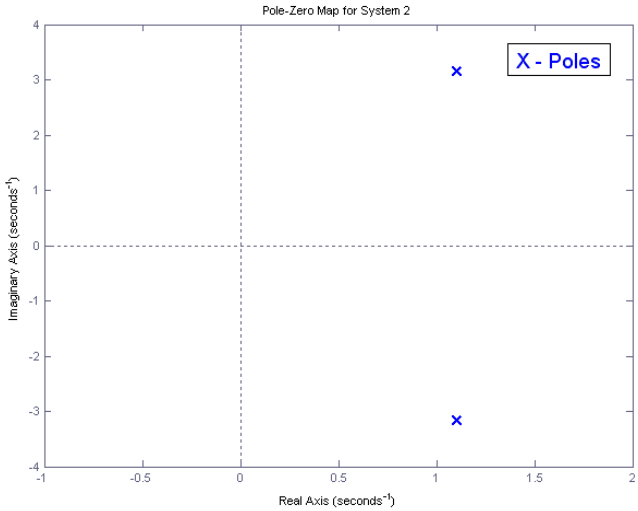

Figure 31: Pole Zero map for System 2l

System 2l has two poles, both in the right half plane, but has no zeros. For these tests, Desired Overshoot was set to 6 % and Desired Settling Time to 1.4s. It is anticipate that system 2l will provide less of a challenge than 2g. The directly calculated complex conjugate pair will likely be enough to drive the system.

Case Study 3: System 3

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -5 & -6 \end{bmatrix}$$ State Space Model for System 3

$$B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$C = [12 \quad 0 \quad 0]$$

$$D = [0]$$

Transfer Function for System 3: $$H(s) = \frac{12}{s^3+6s^2+5s}$$

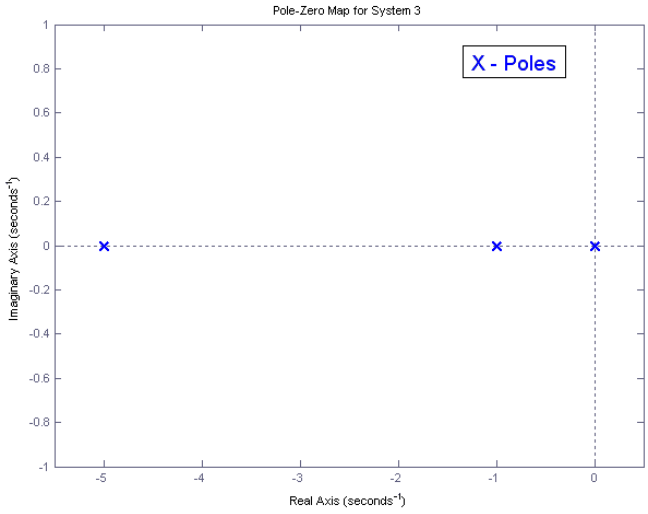

Figure 32: Pole Zero map for System 3

System 3 has three poles and no zeros. It is likely that two directly calculated complex conjugate poles will be enough to dominate, so long as the third pole is sufficiently far from the origin. The desired criteria for tests involving System 3 were Desired Overshoot 7% and Desired Settling Time 4.5s.

Case Study 4: System 3a

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -5 & -6 \end{bmatrix}$$

State Space Model for System 3a

$$B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$C = [12 \quad 0 \quad 0]$$

$$D = [0]$$

Transfer Function for System 3a: $$H(s) = \frac{12}{s^3+6s^2+5s}$$

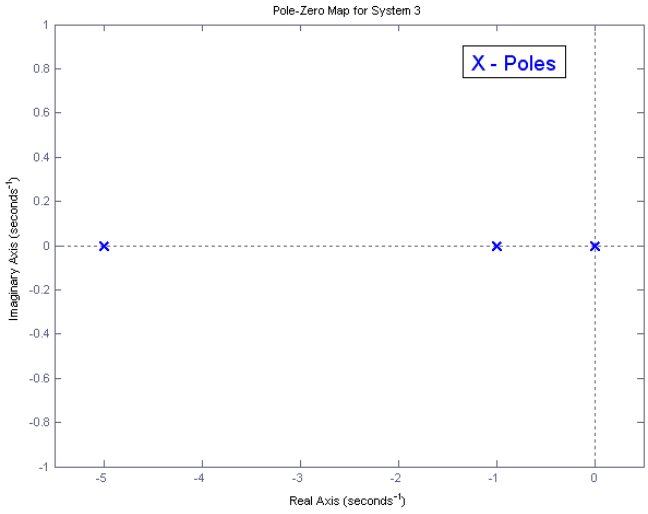

Figure 33: Pole Zero map for System 3a

System 3a is identical to system 3; however the performance criteria were set to a stricter Desired Overshoot of 4 % and Desired Settling Time of 6.1s.

Case Study 5: System 4

$$A = \begin{bmatrix} -9 & -33 & -51 & -26 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$ State Space Model for System 4

$$B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$C = [0 \quad 25 \quad 52.5 \quad 5]$$

$$D = [0]$$

Transfer Function for System 4: $$H(s) = \frac{25s^2 + 52.5s + 5}{s^4 + 9s^3 + 33s^2 + 51s + 26}$$

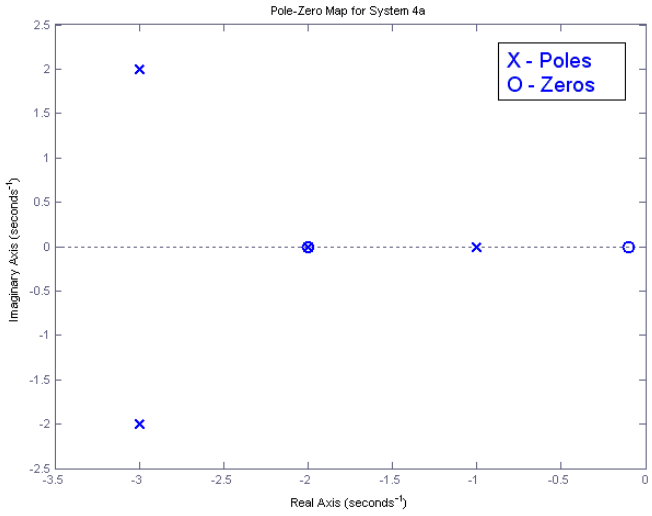

Figure 34: Pole Zero map for System 4

All the poles and zeros of this system are in the left half plane. There is a pole with a concurrent zero at -2. The Desired Overshoot was 10%, and the Desired Settling Time was 3.5s. Some difficulty is definitely expected, as both zeros will push performance away from directly calculated expectations.

Case Study 6: System 4a

$$A = \begin{bmatrix} 1.024 & -2.318 & 6.895 & -3.074 \\ 4.287 & 1.185 & 5.656 & 4.959 \\ 2.059 & 1.702 & -0.5937 & 0.7037 \\ -0.8556 & 7.26 & 5.142 & 4.874 \end{bmatrix}$$ State Space Model for System 4a

$$B = \begin{bmatrix} 2.629 \\ 2.552 \\ 4.42 \\ 3.24 \end{bmatrix}$$

$$C = [-0.2788 \quad -0.1028 \quad -0.8029 \quad 1.334]$$

$$D = [0]$$

Transfer Function for System 4a: $$H(s) = \frac{-0.222s^3 + 56.47s^2 + 472.1s + 2058}{s^4 - 6.489s^3 - 48.36s^2 + 153.4s + 940.5}$$

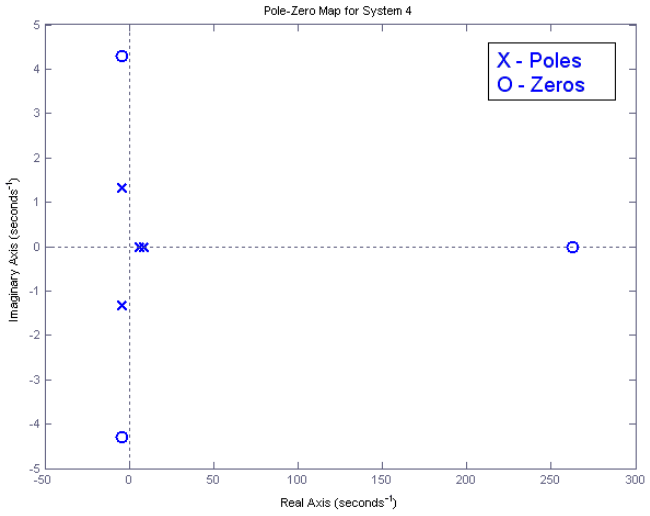

Figure 35: Pole Zero map for System 4a

Most of the poles and zeros are gathered around the origin, but one zero lies firmly in the right half plane. The Desired Overshoot was set to 2.85% and the Desired Settling Time to 5.6s.

Case Study 7: System 5

$$A = \begin{bmatrix} -9 & -33 & -51 & -26 & -6 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

State Space Model for System 5

$$B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$C = [0 \quad 25 \quad 52.5 \quad 5 \quad -3]$$

$$D = [0]$$

Transfer Function for System 5: $$H(s) = \frac{25s^3+52.5s^2+5s-3}{s^5+9s^4+33s^3+51s^2+26s+6}$$

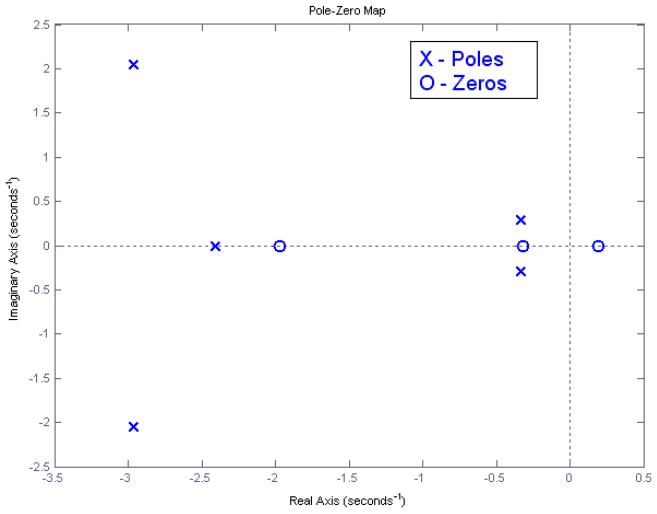

Figure 36: Pole Zero map for System 5

There are so many poles and zeros in this system it is impossible to predict what kind of trouble it will present. The relationship of vector B and C is worth mentioning as well. While B limits the effect of the system input to only the first state variable, the output of the system is the result of all of the state variables except the first state variable. The Desired Overshoot for 5 was 4.8% and the Desired Settling Time 12s.

Case Study 8: System 5b

$$A = \begin{bmatrix} -9 & -33 & -51 & -26 & -6 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

State Space Model for System 5b

$$B = \begin{bmatrix} 1 \\ 0 \\ -10 \\ 7 \\ 0 \end{bmatrix}$$

$$C = [0 \quad 25 \quad 52.5 \quad 5 \quad -3]$$

$$D = [0]$$

Transfer Function for System 5b: $$H(s) = \frac{-490s^4-4456s^3-8526s^2-4388s-789}{s^5+9s^4+33s^3+51s^2+26s+6}$$

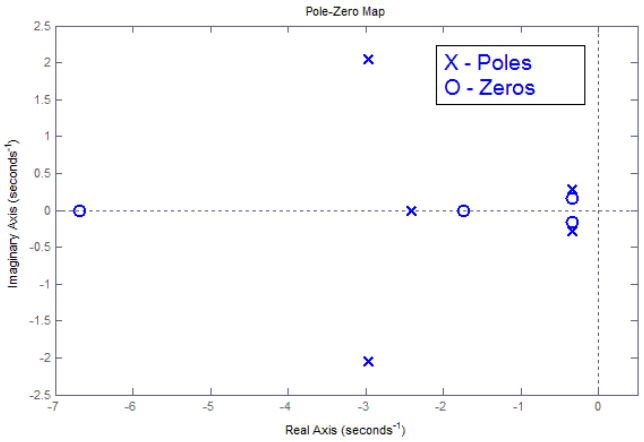

Figure 37: Pole Zero map for System 5b

System 5b and System 5 differ only in the B vector. This new B vector allows the system input to have a more direct effect on several of the state variables. It will be interested to note if this has any impact on the results of these algorithms. The Desired Overshoot for 5b was 4.8% and the Desired Settling Time 12s.

Case Study 9: System 6

$$A = \begin{bmatrix} 1.317 & 2.685 & 0.426 & 0.1132 & -0.7561 & -1.005 \\ 1.014 & -1.39 & 0.2217 & -.6557 & -.2623 & 2.045 \\ 2.106 & 1.308 & 1.061 & -1.282 & -.3894 & -.6528 \\ -.7979 & -1.724 & 1.007 & 2.787 & -.6319 & .9221 \\ -1.765 & -.6232 & 1.483 & 1.739 & 1.704 & -.9541 \\ -1.969 & -.01664 & 1.785 & -.6964 & -1.387 & -1.12 \end{bmatrix}$$

$$B = \begin{bmatrix} .06983 \\ .4422 \\ .327 \\ .9519 \\ .8347 \\ .3271 \end{bmatrix}$$

State Space Model for System 6

$$C = [.4761 \quad .7605 \quad .0737 \quad .4446 \quad .06584 \quad .7573]$$

$$D = [0]$$

Transfer Function for System 6:

$$H(s) = \frac{1.12s^5 - 5.236s^4 - 8.911s^3 + 63.88s^2 - 89.8s + 45.29}{s^6 - 4.359s^5 - 3.8s^4 + 46.84s^3 - 103.4s^2 + 160.1s - 157.2}$$

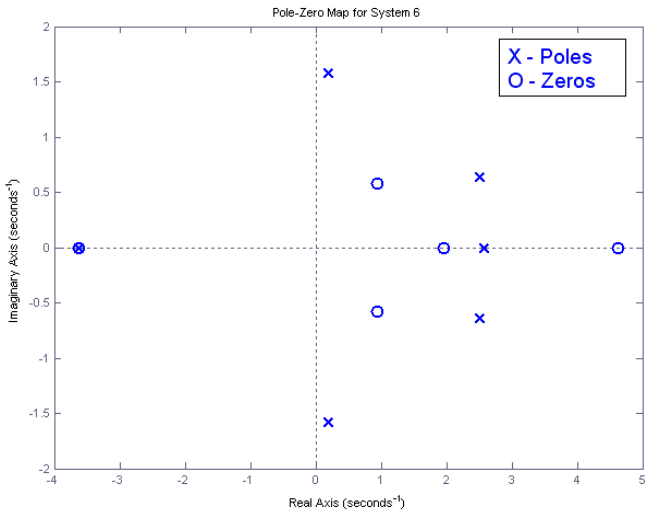

Figure 38: Pole Zero map for System 6

Many of the poles and zeros of System 6 are in the right half plane. The K vector can place them back in the left half plane, but there's no telling what effect the zeros will have on the dynamic response of the system without testing. The final poles of the system will likely end up very far from the starting guess. The Desired Overshoot was set to 15.5% and the Desired Settling Time to 11.04s.

Case Study 10: System 7

$$A = \begin{bmatrix} .7931 & -3.729 & -2.26 & .1303 & .8011 & -2.007 & 1.246 \\ -.3111 & 1.769 & -3.367 & -3.991 & .6095 & .6174 & -2.608 \\ -1.35 & -1.186 & -.4029 & .4683 & 2.507 & -3.533 & -2.165 \\ -1.369 & -2.472 & -.6618 & 1.651 & -3.489 & 2.521 & -1.791 \\ -4.054 & -3.987 & -1.981 & 1.84 & 2.233 & -3.179 & -1.428 \\ -4.996 & .3794 & -4.311 & -.1916 & 2.388 & -.1886 & 1.791 \\ 2.234 & 2.321 & -1.646 & 1.425 & .798 & -4.787 & 2.217 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.4237 \\ 0.2718 \\ 0.7051 \\ 0.6786 \\ 0.777 \\ 0.3956 \\ 0.0251 \end{bmatrix}$$

State Space Model for System 7

$$C = [0.9595 \quad 0.4729 \quad 0.8848 \quad 0.8675 \quad 0.8614 \quad 0.3789 \quad 0.3759]$$

$$D = [0]$$

Transfer Function for System 7:

$$H(s) = \frac{2.576s^6 - 30.73s^5 + 196.4s^4 - 1001s^3 + 3795s^2 - 9253s + 10330}{s^7 - 8.072s^6 = 21.17s^5 - 54.69s^4 + 59.71s^3 + 2303s^2 - 5976s - 9917}$$

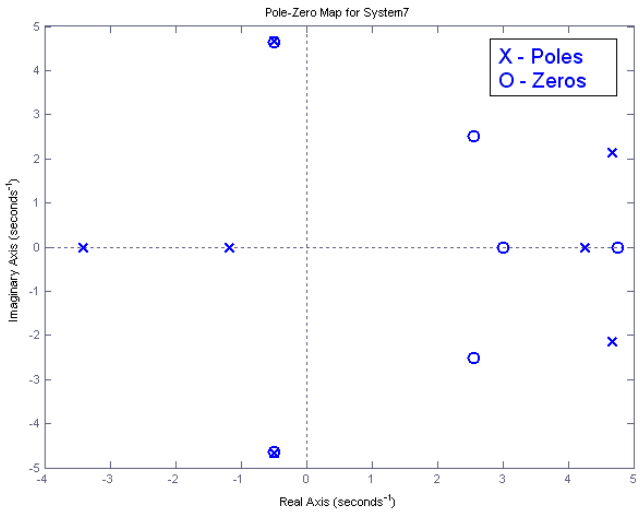

Figure 39: Pole Zero map for System 7

The Desired Overshoot was 12% and the Desire Settling Time 9s. The system is so large

with so many poles and zeros that it is sure to test the limits of the genetic approach. The initial calculations can only generate 2 of the 7 desired poles based on Pole Placement method, and the remaining 5 (the majority) are left to chance.

CHAPTER 7: Analysis, Conclusions and Future Works

7.1 - Results and Analysis

The island approach, using specialized mutation vectors, demonstrates better performance than the regular genetic approach in a variety of conditions. The island approach generally found a solution in a smaller number of generations, and the final fitness scores of the island approach were better than those using the regular genetic approach.

A comparison of total computation time is also made. While the island approach concludes within fewer generations, it is shown that those generations can take more time to conclude themselves under certain conditions.

An overview of the total results for a few systems will be presented first, and then specific examples will be explored in further detail.

First, the results for system 4r are presented here. As shown earlier, this system had two poles in the right half plane, and three zeros, one at 261. The desired percent overshoot was set at 10%, and the desired settling time was set at 3.5s.

Both approaches were able to find fit results in most cases. Their performance in each fitness mode is shown here.

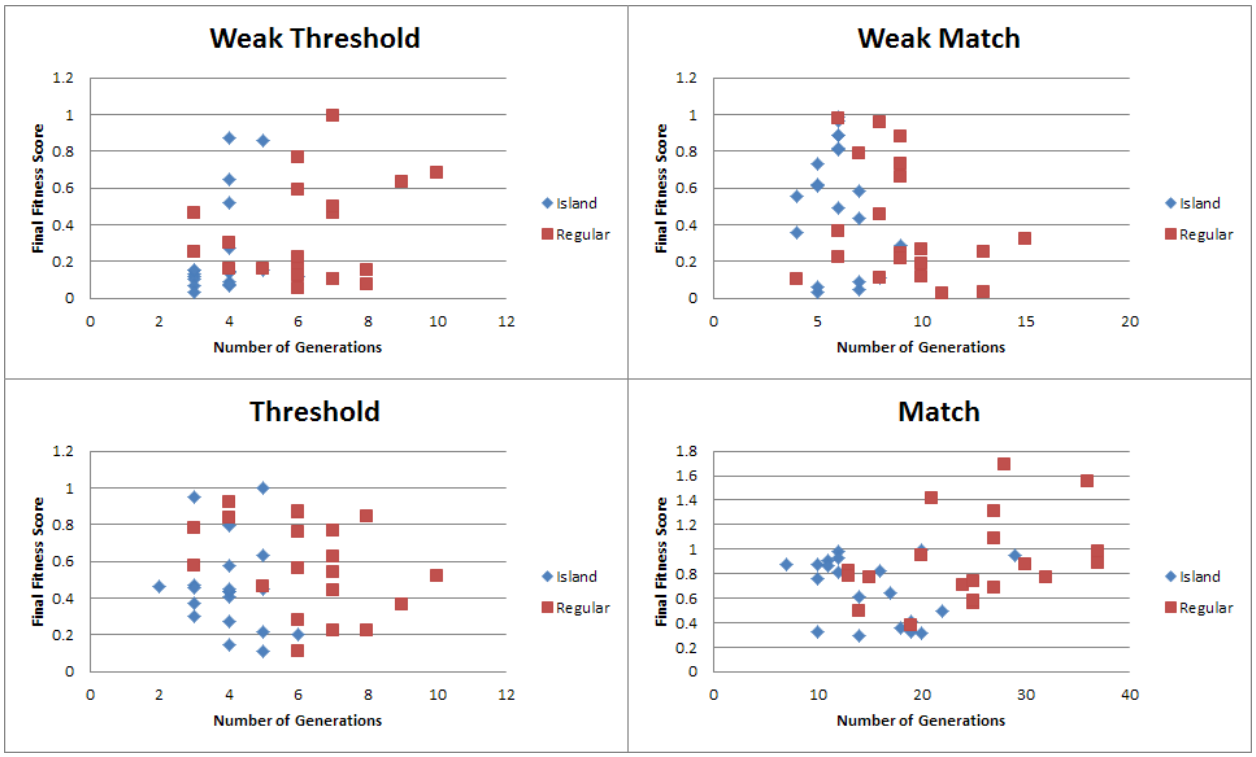

Figure 40: Results from System 4

The island approach consistently shows results in fewer generations, and the final fitness scores are lower (better) than those of the regular genetic algorithm. It is also worth pointing out that when Match fitness was used, 25% of the results for the regular fitness method were above 1, meaning that the normal genetic algorithm exited due to stagnation and did not return a successful result.

The next case to compare is that of system 3. This system had no zeros, and was fairly easy to control. Both implementations settled on fit results quickly.

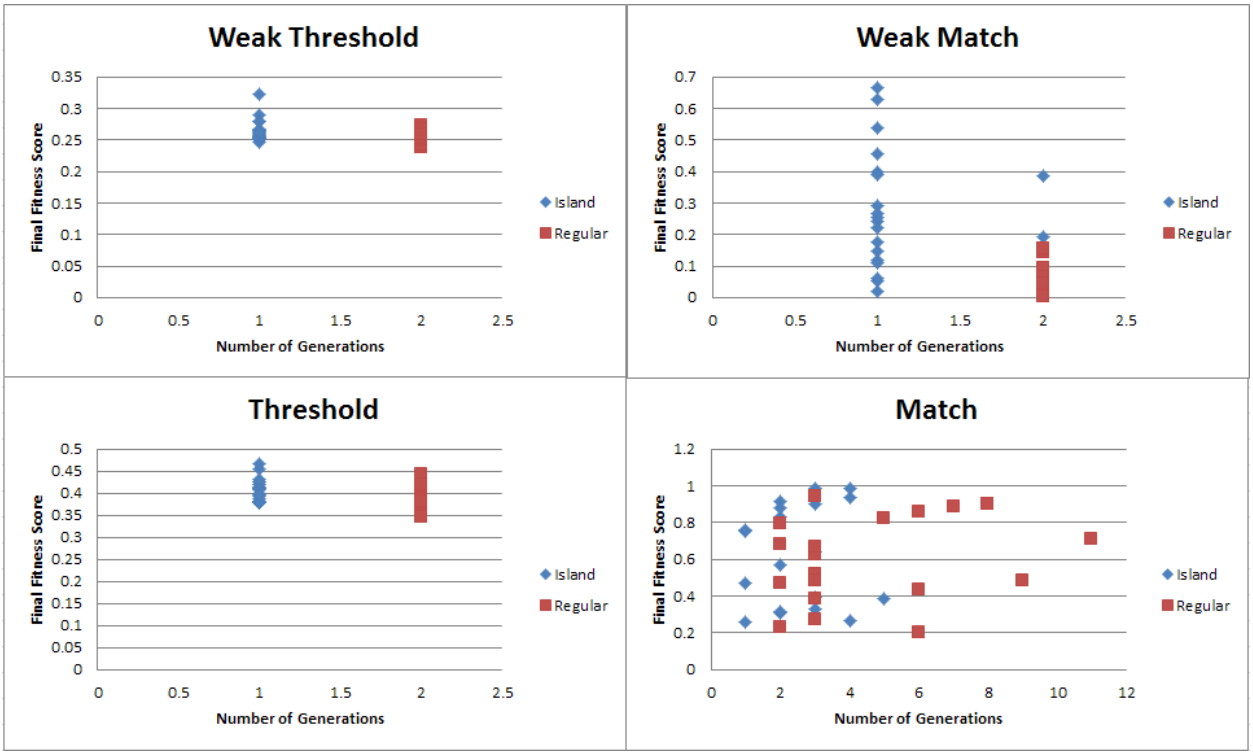

Figure 41: Results from System 3

System 3 did not present much of a challenge to either system in Weak Threshold, Threshold, or Weak Match mode. Both approaches settled quickly and with remarkably fit results.

The largest difference between the two implementations can be seen again in Match mode. The island approach consistently settled in fewer generations, but it would be difficult to assert that the island approach was strictly better overall. There are several cases where the regular approach took up to ten generations longer, but returned the same result as the island approach.

Interesting comparisons can also be drawn when the final systems produced by the different implementations are analyzed. The following is a comparison based on the Weak Match mode on System 4:

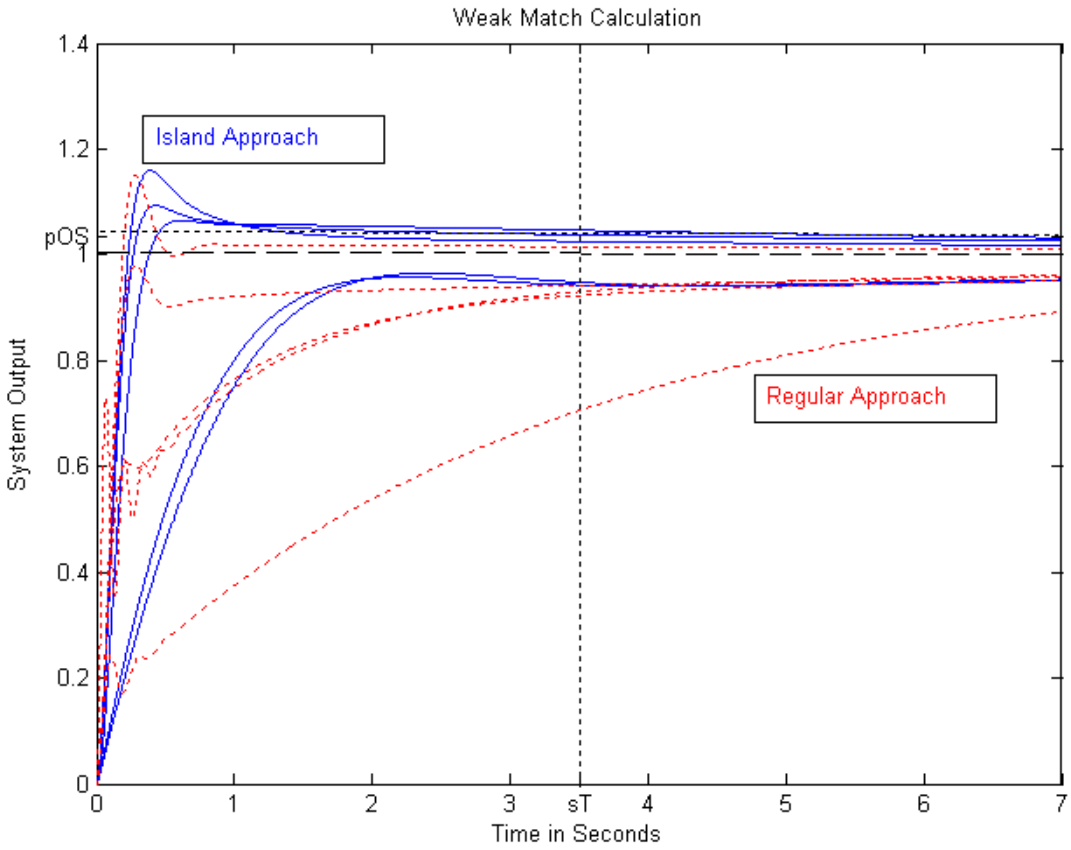

Figure 42: Sample of Island and Regular Implementation Solutions

These final solutions show significant ringing in the regular approach, as well as several severely overdamped solutions. Weak Match attempts to satisfy at least one of the user specified conditions, even if it cannot satisfy both.

Both algorithms were running under the same fitness calculation with the same system, but it is my belief that the specialization in the island populations led the algorithm towards more diverse areas, resulting in more fit solutions.

Let us return to system 3 and look at Threshold results for both implementations:

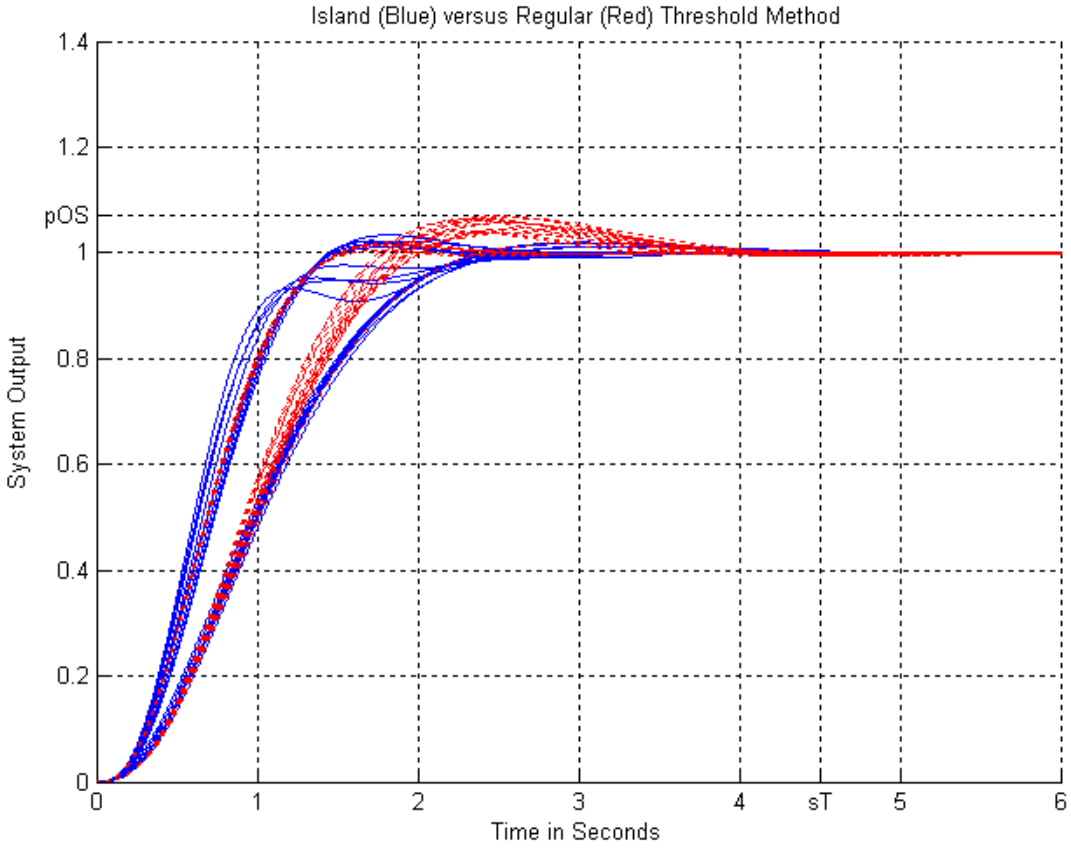

Figure 43: Threshold Mode Comparison

The object of Threshold method is to minimize settling time and overshoot. While the regular genetic algorithm (shown in red dashed lines) does perform better than absolutely required, the island approach (shown in blue) consistently has lower percent overshoot and settling time than the regular genetic approach. This is likely due to the increased diversity in the island population.

Even when this system was run again with different overshoot and settling time demands, the island approach found better results.

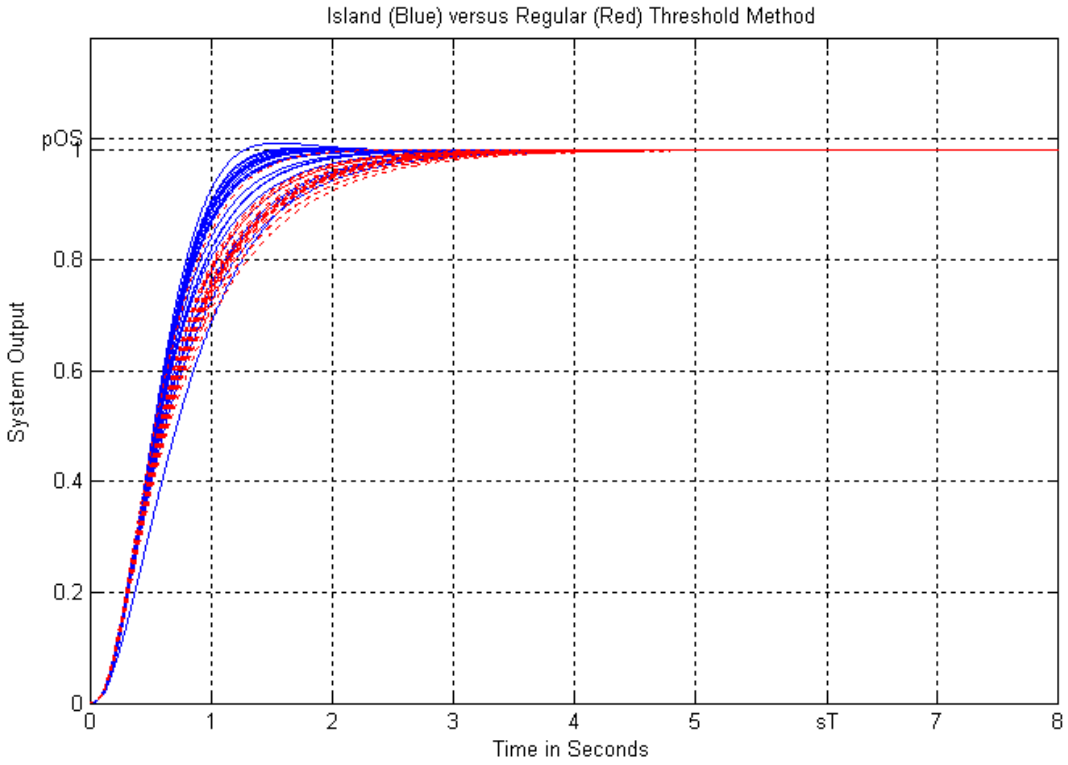

Figure 44: System 3a in Threshold Mode

The gathering of blue lines display consistently shorter settling times, and are all under the tighter 4% overshoot requirement.

Even the complexities of System 7 could be overcome by the genetic algorithms, as can be seen here.

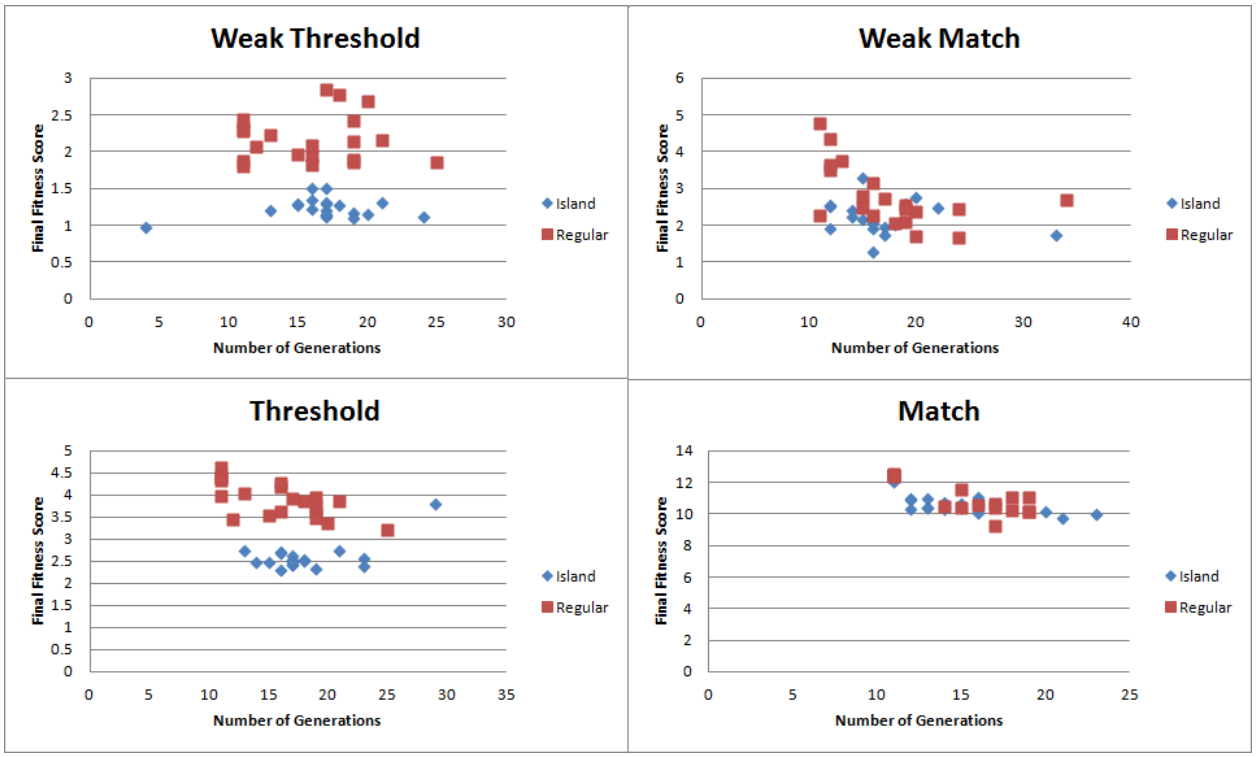

Figure 45: Results from System 7

The Match method, which traditionally presented the toughest challenges, reveals strikingly similar results from both algorithms.

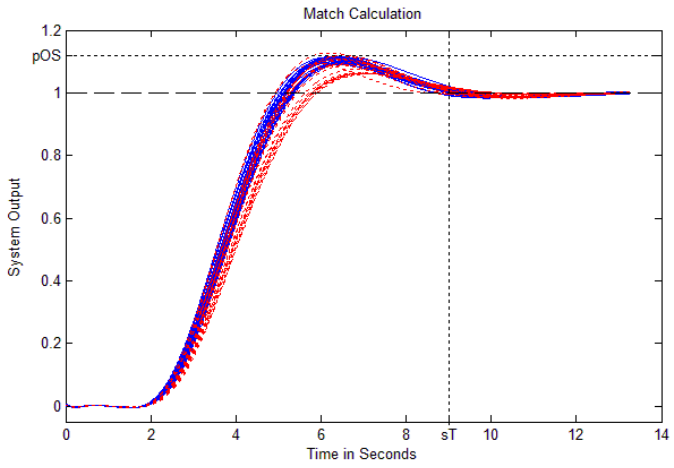

Figure 46: Match Mode comparison

The island approach, shown in blue comes somewhat closer, but both are very close to the desired performance criteria. The main contributor to their poor fitness scores (averaging

about 10) is the undershoot that is measured as the system oscillates near zero in the first two seconds. It is disappointing that the island approach isn't clearly better than the regular algorithm in this difficult case, but some theories as to why are presented in the following section.

Island approach has been demonstrated to produce results which are as good, if not better than the regular genetic implementation. Let us now turn to other measures of performance.

The amount of time taken to reach the final result was calculated for each instance above. This was simply the time between when the log file was created (just a few moments after the function begins) and when the final write to the log file took place.

The time taken overall was then divided by the number of generations that the algorithm searched. For the island approach, the time was further divided by the number of islands in the population, as the island approach is intended to be distributed to multiple computers or processors within the same computer.

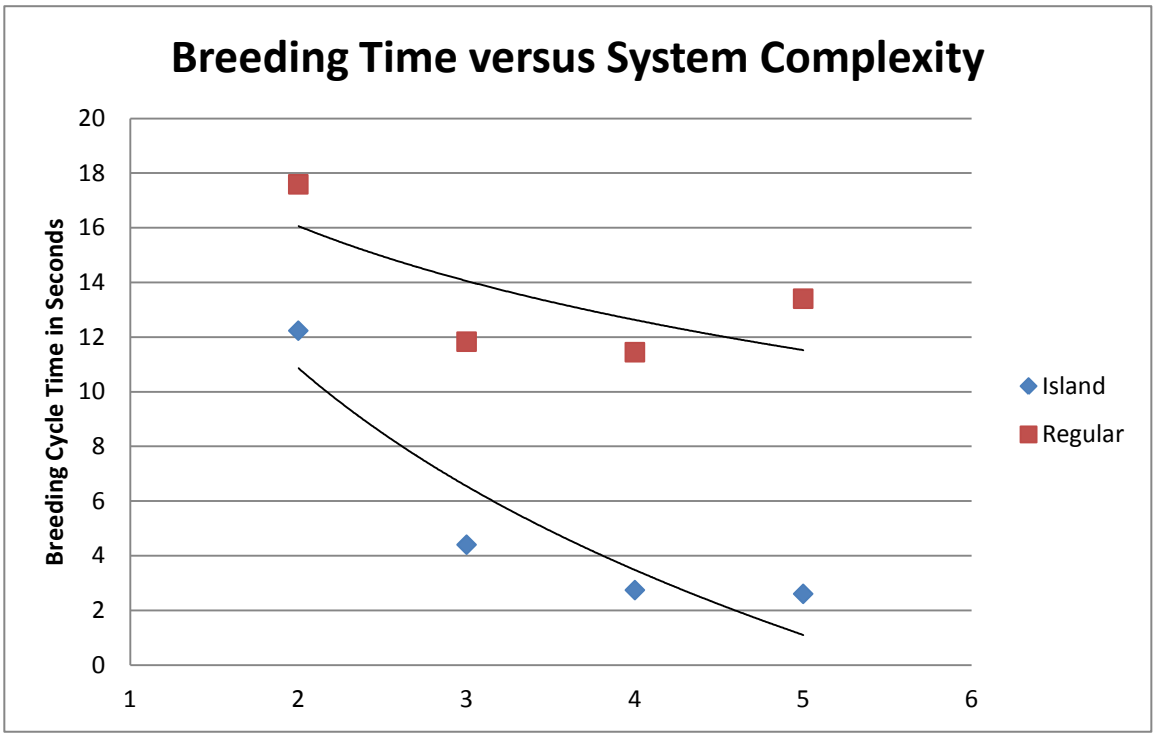

Figure 47: Seconds per Generation as system complexity increases

7.2 - Conclusions

A genetic approach has been shown to be a successful method of guiding a system's impulse response towards user generated criteria such as percent overshoot and settling time. Several fitness methods are available to generate solutions that meet or solutions that exceed those criteria.

The addition of orthogonal mutation vectors used in the island approach has a positive effect on the final solutions, as well as generally guiding the algorithm to a final solution in fewer generations.

As with any stochastic method, it is difficult to exhaustively study the effect of these changes, but these initial results show some promise.

7.2.1 - Weaknesses

Some of the systems chosen to test proved too difficult to transform for either algorithm. This is likely due to the amount of zeroes in the B vector, which will limit the effect a gain vector can have on the system's performance. Recall the pole placement method transforms the system by effectively replacing the A matrix with (A-BK).

For instance, with System 5b, every instance of the algorithm exited with a fitness score below 1 (a success.) For System 5, which is the same except for the B vector, no instance was able to successfully complete, and all attempts ended with fitness scores well above the 1 threshold. The results for each system are displayed on the following page for comparison.

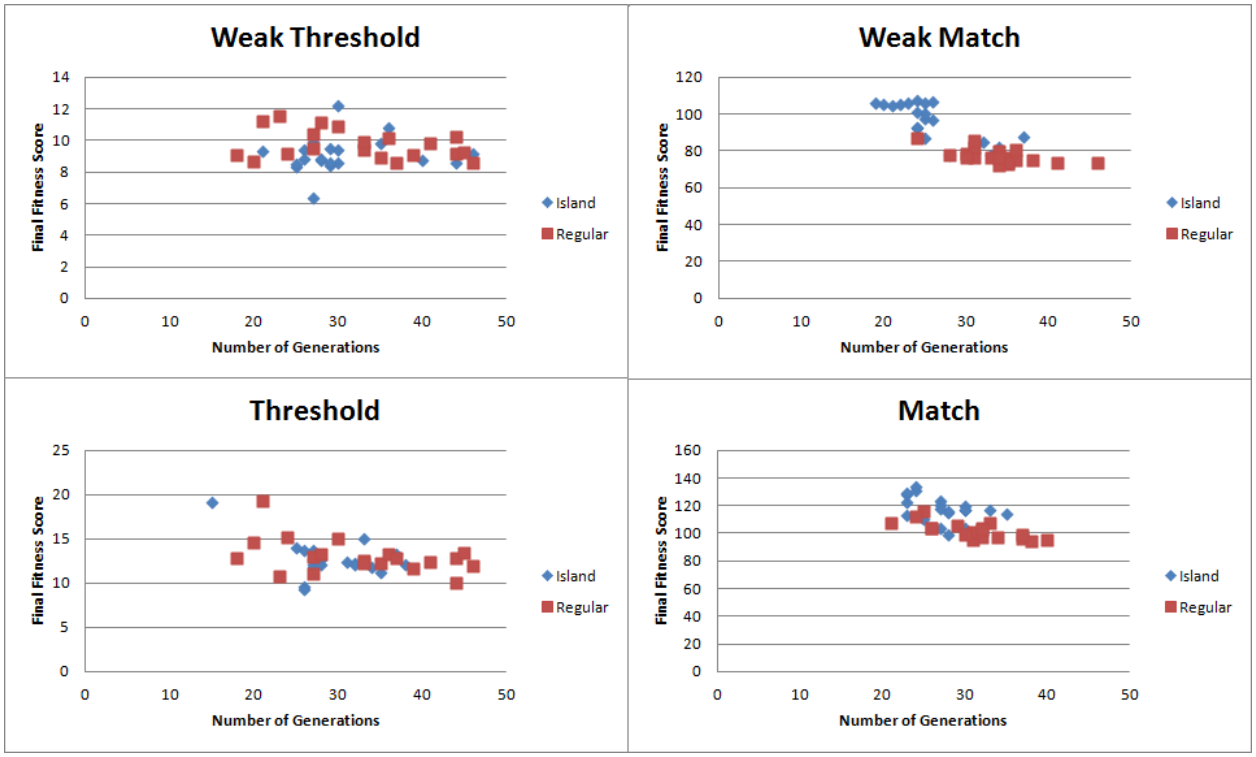

Figure 48: Results for System 5

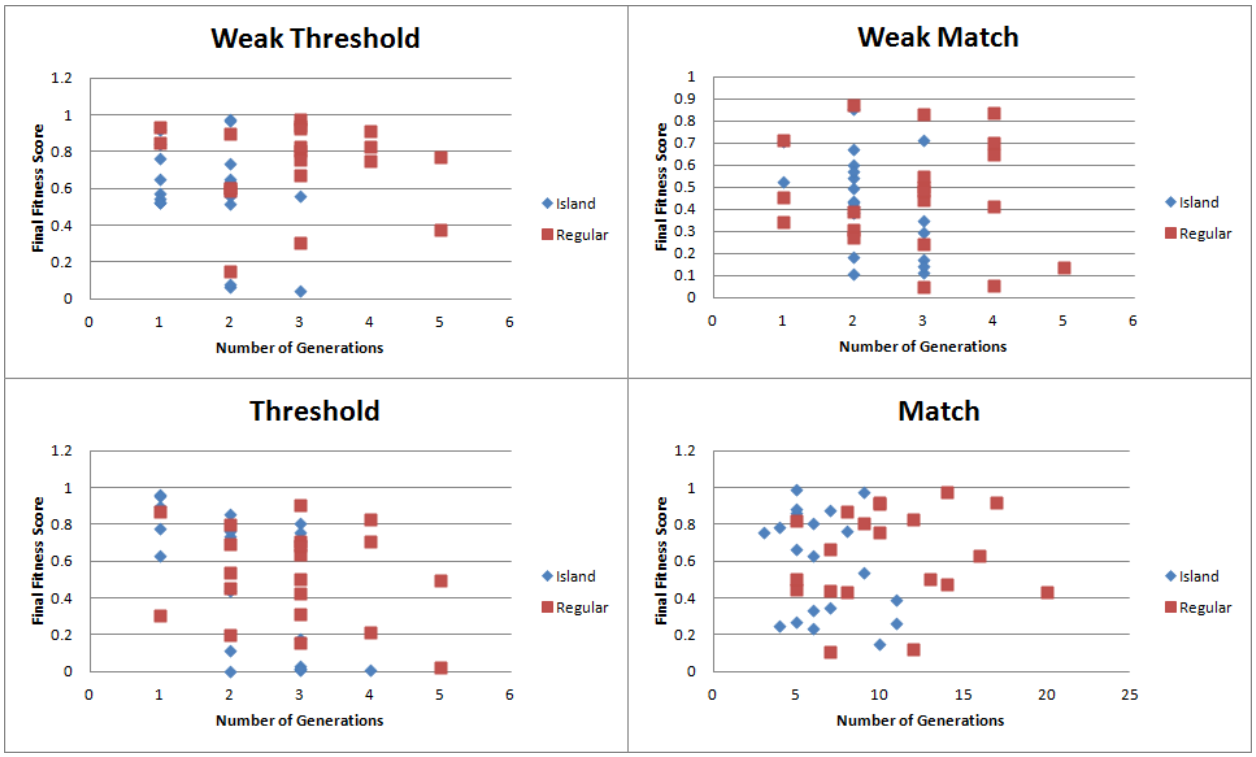

Figure 49: Results for System 5b

Currently, the total number of genomes does not change in the island approach based on the size of the system. This means that a system that has 6 islands will have half the population per island than a 3 island system. There is performance degradation in systems over 5 islands, which become more noticeable at 7. No systems beyond seven variables were considered in this test.

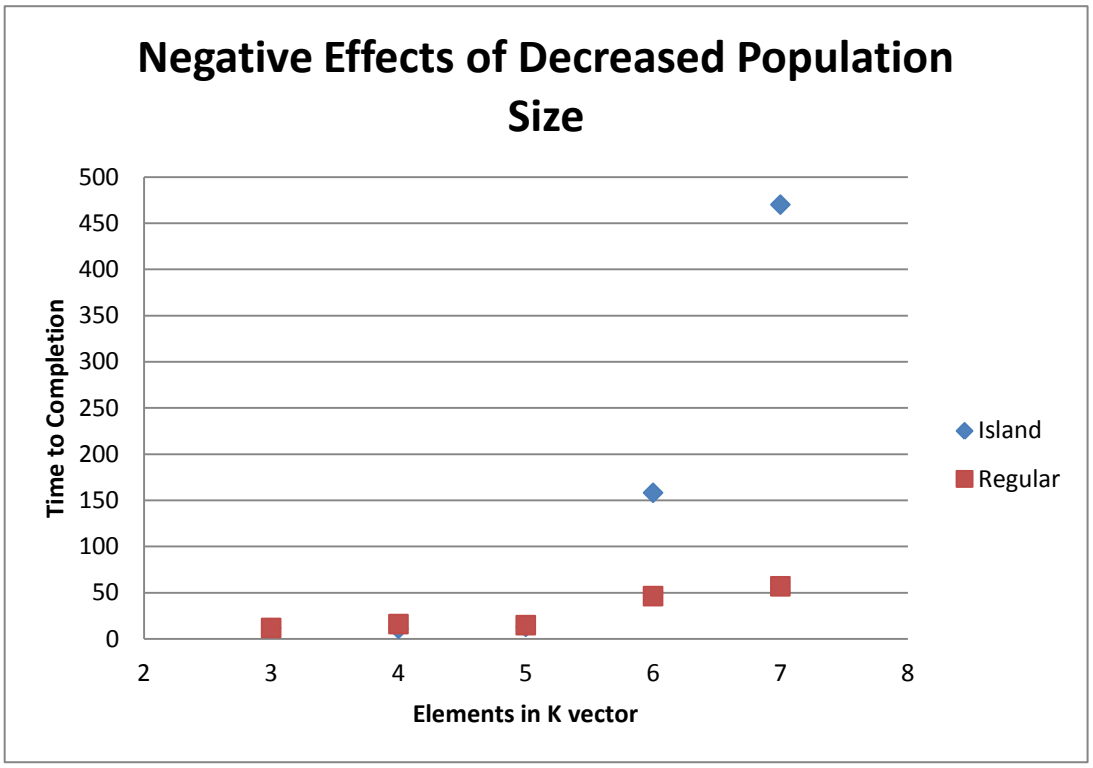

Figure 50: System Size versus Time to Completion

The regular approach has a significant advantage in total run-time in those larger systems. It is proposed that the smaller islands become too specialized with so little genetic material available. When the breeding cycle occurs, there are too few parents to choose from, and the selections are not diverse enough to create stable genomes. This can be observed in the dramatic increase in unstable solutions that the breeding and mutation process create, shown here.

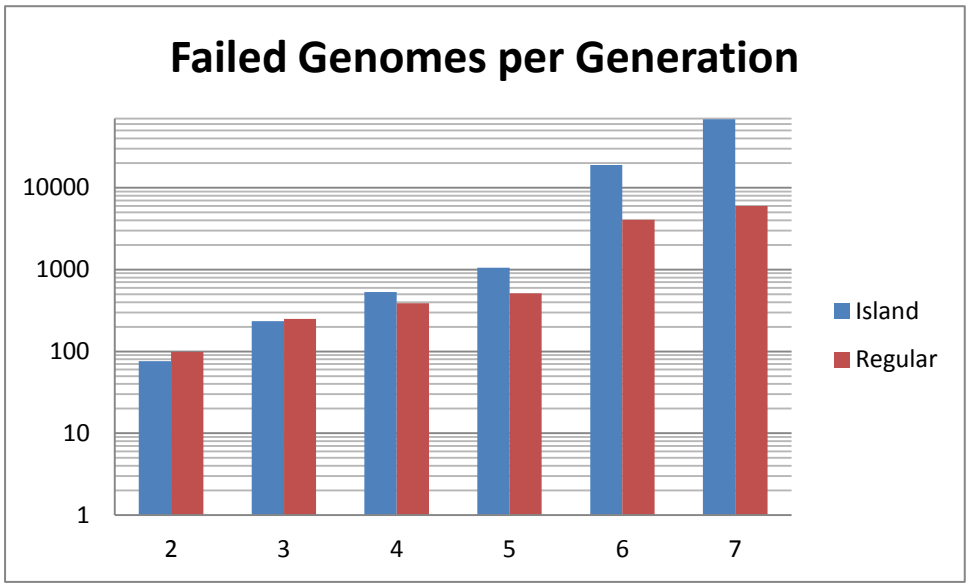

Figure 51: Comparison of Failed Genomes

When the system size is smaller than 5, it can be seen that the regular approach and the island approach produce roughly similar amounts of unstable (failed) genomes, but the trend begins at six and seven variables. The vertical axis is logarithmic, showing a dramatic increase in failed genomes in the island approach with six or more islands. At this point, each island has around 150 or fewer genomes.

The island approach does not fail at higher systems. It can be seen from the results printed earlier that the island approach finds good solutions in the end, and in most cases completes in fewer generations. However, the total time spent per generation suffers severely when the population becomes too small.

7.3 - Future Work

The fitness modes provided in the implementations are generic, skeletal fitness calculations that could be replaced for task-specific applications. Weights might be given by the user as to which of the two criteria were more important, so that fitness calculations could favor

it over the other.

If the genomes were represented within the implementation as objects with additional traits, it might be possible to track a limited view of the ancestry of a genome. When an offspring is produced that has significantly lower fitness than either of the parent genomes, then the changes which produced that offspring (in either breeding or in mutation) could be recorded so that similar changes could be favored in the future. Parents might gain more frequent access to breeding based on the performance of their offspring, and so forth.

The largest improvement to be made to the island approach is true parallelization. A central computer could house the tower, and control when each of the islands enters a breeding and mutation cycle. None of the islands need to communicate with one another except during mutation, which could easily be handled by the centralized (tower) computer in a spoke and hub topology. Communication times are negligible in comparison to breeding and mutation cycles, as only a handful of genomes are passed at a time.

The island approach currently focuses mutation along one element in the genome. Further experimentation might include setting those mutation vectors to be orthogonal diagonal combinations elements. One island, for instance, might focus on mutation that simultaneously increases element 3 twice as much as element 4, or decreases both by that same ratio. The direction of this vector could be set at the onset of the algorithm, or it could dynamically evolve based on the limited ancestral feedback proposed above.

As mentioned in the section on weaknesses, the population of each island could scale up when the system size increases, or perhaps each island could always be set to have the same population, regardless of the number of islands. This would offset the gains made in parallelizing the population, but a study could be done to find the benefits of such a trade-off.

Finally, the best fit genomes are collected in the tower, but are not currently bred or experimented on. Some efforts might be made to explore more specialized mutation, such as weighted sums of the genomes in the tower, to see if even more fit solutions can be produced. It is possible that the tower could use the makeup of the most fit solutions in the tower to provide mutation guidance to the islands.

References

[1] Chiu H. Choi, "Step Response Improvement by Pole Placement with Observer," Proceedings of the 40th Southeastern Symposium on System Theory, pp.7-12, Mar. 2008.

[2] Nise, N.S., "Control Systems Engineering," Hoboken, NJ: Wiley, 2004.

[3] Kautsky, J. and N.K. Nichols, "Robust Pole Assignment in Linear State Feedback," *Int. J. Control*, 41 (1985), pp. 1129-1155

[4] Guo, Pengfei, Xuezhi Wang, and Yingshi Han. "The Enhanced Genetic Algorithms for the Optimization Design." *2010 3rd International Conference on Biomedical Engineering and Informatics (BMEI 2010)* (2010): pp.2990-2994.

[5] Goldberg, David E. "Genetic Algorithms In Search, Optimization, And Machine Learning," n.p.: Reading, Mass.: Addison-Wesley Pub. Co., 1989.

[6] Schalkoff, Robert J., "Intelligent Systems: Principles, Paradigms, and Pragmatics." Sudbury, MA: Jones and Bartlett, 2011.

[7] Srinivas, M. and L.M. Patnaik, "Adaptive Probabilities Of Crossover And Mutation In Genetic Algorithms." Ieee Transactions On Systems Man And Cybernetics 24.4 (n.d.): 656-667. Science Citation Index.

[8] Varsek, A., T. Urbancic, and B. Filipic, "Genetic algorithms in controller design and tuning," *IEEE Trans. on System, Man, and Cybernetics,* vol. 23, no. 5, 1993, pp.1330-1339.

[9] Godefroid, Patrice, and Sarfraz Khurshid. "Exploring Very Large State Spaces Using Genetic Algorithms." International Journal On Software Tools For Technology Transfer 6.2 (2004): 117-127.

[10] Alpay, Daniel, and I. Gohberg. "State Space Method : Generalizations And Applications." n.p.: Birkhäuser Verlag, 2006.

[11] Hangos, K. M., J. Bokor, and G. Szederkényi. "Analysis And Control Of Nonlinear Process Systems." n.p.: Springer, 2004.

[12] C. M. Fonseca and P. J. Fleming, "Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization", Proc. ICGA 5, pp. 416-423, 1993.

[13] D Kobler, et al. "Parallel Island-Based Genetic Algorithm For Radio Network Design." Journal Of Parallel And Distributed Computing 47.1 (n.d.): 86-90.

[14] M. Munetomo, Y. Takai, Y. Sato, “A migration scheme for the genetic adaptive routing algorithm”, in Proc. IEEE International Conference on Systems, Man and Cybernetics, vol. 3, pp. 2774 – 2779, October 1998.

[15] F. de Toro, J. Ortega, J. Fernandez, A. Diaz, “PSFGA: a parallel genetic algorithm for multiobjective optimization”, in Proc. 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, pp. 384 – 391, January 2002.

[16] K. Deb, “Multi-objective genetic algorithms: problem difficulties and construction of test problems,” Evolutionary Computation, Vol.7, No.3, 205-230, 1999.

[17] Sun Yang, et al. "An Improved Parallel Genetic Algorithm Based On Injection Island Approach And K1 Triangulation For The Optimal Design Of The Flexible Multi-Body Model Vehicle Suspensions." 2009 ISECS International Colloquium On Computing, Communication, Control & Management (2009): 30.

[18] Patel, R., M.M. Raghuwanshi, and L.G. Malik. "An Improved Ranking Scheme For Selection Of Parents In Multi-Objective Genetic Algorithm." 2011 International Conference On Communication Systems & Network Technologies (CSNT) (2011): 734.

[19] M. Rucin'ski, D. Izzo, and F. Biscani, "On the impact of the migration topology on the Island Model," Elsevier, 2010.

```
function [ kVector, fitnessMatrix ] = GeneticK( systemA, percentOS,
settlingTime, modeOption )

%  This function seeks to find a transform vector K.  If the system is
%  transformed using the vector K, the resultant system will exibit a
%  specified percent overshoot and settling time or better.

%  Several variables are set here for continued use.
% ================================================================

% INPUT REQUIREMENTS:
% [systemA]         must be a StateSpaceMode
% [percentOS]       is the user's desited percent overshoot
% [settlingTime]    is the user's desired settling time
% [modeOption]      is an optional argument which must be a string.  This
%                   will determine how fitness is calculated


%   Reading the modeOption string, if it exists
%   --- See the calcFit method for more details ---

if nargin == 4
   if ischar(modeOption)
       if strcmp(modeOption, 'weakThreshold')
           % in weakThreshold fitness mode, the algorithm will use the
           % percentOS and the settlingTime arguments as a threshold, and
           % return the first kVector that beats the average of these two
           % criteria
           fitMode = 3;
       elseif strcmp(modeOption, 'threshold')
           % in threshold fitness mode, the algorithm will use the
           % percentOS and the settlingTime arguments as a threshold, and
           % return the first kVector that beats both of these two criteria
           fitMode = 2;
       elseif strcmp(modeOption, 'weakMatch')
           % in weakMatch fitness mode, the algorithm will use the
           % percentOS and the settlintTime arguments as goals, and return
           % the first kVector to come close to the average of these two
           % criteria
           fitMode = 1;
       elseif strcmp(modeOption, 'match')
           % in match fitness mode, the algorithm will use the percentOS
           % and the settlingTime arguments as goals, and return the first
           % kVector to come close to both of these two criteria
           fitMode = 0;
       else
           fitMode = 0;
           fprintf('%s is not a valid modeOption.  Using ''match''
instead.\n', modeOption);
       end
   else
       fitMode = 0;
       fprintf('modeOption is not a string.  Using ''match'' instead.\n');
   end
```

```
end

if nargin < 4
    fitMode = 0;
end

if nargin < 3
    fprintf('   ----------   ERROR   ----------\n');
    fprintf(' Insufficient arguments.  A call to this function must
include:\n');
    fprintf(' a state space system, a desired percent overshoot, a desired
settling time,\n');
    fprintf(' and may optionally include a mode option string.\n');
    fprintf('\n GeneticK( systemA, percentOS, settlingTime, modeOption )\n');
    return
end

%  The size of the system is calculated
  systemSize = size(systemA.a, 1);
%  The number of possible solutions in the population each generation
  populationSize = 1000;
%  Population is pre-allocated for speed.
  population = zeros(populationSize, systemSize);
%  The number of generations before the algorithm returns the "best result"
  generations = 200;
%  The percent chance that any given element will mutate
  mutationChance = 0.20;
%  A watchdog for stagnation
  stagnation = false;

%  If detailed performance information is desired, set this equal to one
verbose = 1;
if verbose == 1
    failedMatrix = zeros(1, 2);
end
    fitnessMatrix = zeros(1, populationSize);
%  The percent of possible solutions that carry unchanged from one
%  generation to the next.  These are always chosen as the "most fit"
  champions = populationSize * 0.2;

%  Log File for tracking the algorithm's progress, dynamically created.
fileStub = datestr(now,'yyyymmdd_HHMMSS');
logFileName = sprintf('%s.txt',fileStub);
logFile = fopen(logFileName, 'w');

% ================================================================
fwrite(logFile, sprintf('System \nA: = \n%s\nB: = \n%s\nC: = \n%s\nD: =
\n%s\n', mat2str(systemA.a), mat2str(systemA.b), mat2str(systemA.c),
mat2str(systemA.d)));
[y,t,x] = step(systemA);
systemTest = stepinfo(y,t);
fwrite(logFile, sprintf('Percent Overshoot: %f\nSettling Time: %f\nRise Time:
%f\nUndershoot: %f\n', systemTest.Overshoot, systemTest.SettlingTime,
systemTest.RiseTime, systemTest.Undershoot));
if isnan (calcFit(systemA, percentOS, settlingTime, fitMode))
    fwrite(logFile, sprintf('Current Fitness: Undeterminable \n'));
else
```

```
    fwrite(logFile, sprintf('Current Fitness: %f \n', calcFit(systemA,
percentOS, settlingTime, fitMode)));
end


%% CALCULATION OF IDEAL COMPLEX CONJUGATE PAIR %%
polesVector = calculatePoles(percentOS, settlingTime);
fwrite(logFile, sprintf('Pole Vector: %s \n', mat2str(polesVector)));

%% INITIAL POPULATION CREATION %%

%  The current generation is set to one
  g = 1;

%  An initial population of educated guesses need to be made.
%  These are stored within several matrices.  Each row is a whole k vector
%  candidate.  There is an additional column at the end of the row to store
%  the calculated fitness of that k vector.
%  Each population is also sorted so that the "most fit" vectors are on top
%  (index 1) and the least fit are on bottom.

  [population, failedVector] = createPopulations(systemA, polesVector,
populationSize, mutationChance, percentOS, settlingTime, fitMode);

%  Status for Generation One is presented to the user
  fitnessMatrix = [transpose(population(1:populationSize, (systemSize+1) ))];
  writeGeneration(g,population, logFile);
  displayGeneration(g,population);

  if verbose == 1
      failedMatrix(g, 1:2) = failedVector;
      fprintf('Unstable: %d    Failed Lsim: %d \n', failedMatrix(g,1),
failedMatrix(g,2));
      fwrite(logFile, sprintf('Unstable: %d    Failed Lsim: %d \n',
failedMatrix(g,1), failedMatrix(g,2)));
  end

   %% BREEDING CYCLE BEGINS %%

%  The algorithm then launches into a cycle.  Rows of the population matrix
%  are "bred" together to produce new "offspring", and then each
%  "offspring" is subject to a percent chance of mutation.  If that new
%  result is capable of trasforming the system into a stable system, and a
%  value for fitness can be calculated, then that result is allowed into
%  the population for the next generation.

  while (population(1, systemSize + 1) > 1 && g < generations && ~stagnation)
%  The generation count is incremented once at the begining of each loop.
      g = g + 1;
      failedMatrix(g, 1:2) = 0;

%  offSpring is the generation created in this loop.  It is created from
%  the last generation.  All members of the current "island{i}" matrix
%  are available for breeding.
      offSpring = zeros(populationSize, systemSize+1);
      offSpringCount = 1;
```

```
%  The best fit, the "champions" will be carried to the next generation
%  automatically.  This is their privelege for being the best.
      for (i = 1:champions)
          offSpring(i, 1:systemSize+1) = population(i, 1:systemSize+1);
          offSpringCount = offSpringCount + 1;
      end

%  We continue to breed new offSpring until the # of champions added to the
%  # of offspring generates a new population of the same size as the last
%  generation.

      while (offSpringCount <= populationSize)
%  Potential offspring are created using this method.
          offSpringVector = breedingRandomCrossover(population);

% offSpringVector may return multiple Children.  Each must be
% tested individually.
          for c = 1:size(offSpringVector, 1)

% Mutation seeks to randomly alter elements within a solution vector.
% This seeks to prevent the solution pool from gathering around a local
% minima, by allowing the algorith to explore some new territory.

% The offspring are subject to a percent chance of mutation first.  Then we
% determine the effects that this new possible K vector would have on the
% system, before deciding whether to add it into the population or to
% discard it.
              if rand < mutationChance
% We mutate all elements in a vector a little bit...
                  for i=1:systemSize
                      offSpringVector(c,i) =
subtleMutate(offSpringVector(c,i));
                  end
              end

% Transform the system using the possible new vector
              kc = 1/(systemA.c * (-inv(systemA.a - systemA.b *
offSpringVector(c, 1:systemSize))) * systemA.b);
              sys = ss(systemA.a - systemA.b * offSpringVector(c,
1:systemSize), systemA.b, systemA.c * kc, systemA.d);

% Calculate fitnes score, or discard if a bad result is returned.
              fitness = calcFit ( sys, percentOS, settlingTime, fitMode );
              if isnan(fitness)
                  if verbose == 1
                      failedMatrix(g,2) = failedMatrix(g,2) + 1;
                  end
              elseif fitness == -1
                  if verbose == 1
                      failedMatrix(g,1) = failedMatrix(g,1) + 1;
                  end
              else

% The resulting "fitness" is added to the end of the vector for use in
% further calculations. newK is added to the population matrix
                  offSpring(offSpringCount, 1:(systemSize + 1)) = [
offSpringVector(c, 1:systemSize) fitness];
```

```
% The index is incremented.
                  offSpringCount = offSpringCount + 1;
              end % fitness calculation and handling
          end % each potential solution in the offSpringVector
      end % while loop for breeding cycle

% The new offspring are re-sorted by fitness
      population = sortrows(offSpring, (systemSize+1));

  %%  Reporting %%

      fitnessMatrix = [fitnessMatrix ; transpose(population(1:populationSize,
(systemSize+1) ))];
      writeGeneration(g,population, logFile);
      displayGeneration(g,population);

      if verbose == 1
          fprintf('Unstable: %d    Failed Lsim: %d \n', failedMatrix(g,1),
failedMatrix(g,2));
          fwrite(logFile, sprintf('Unstable: %d    Failed Lsim: %d \n',
failedMatrix(g,1), failedMatrix(g,2)));
      end

   % We check to see if we've had improvement in the past 10 generations
      if g > 10
          improvement = fitnessMatrix(g-10,1) - population(1,systemSize+1);
          if improvement < fitnessMatrix(g-10,1) * .07
              stagnation = true;
          end
      end

  end % each generation
  %%  COMPLETION AND RESULTS %%

%  Once a solution exceeds the user supplied criteria, or a sufficient
%  amount of generations have passed, the algorithm returns the best
%  calculated solution.  This is represented in the form of a vector of
%  desired poles for the system.

  fprintf('Generation:            %d \n', g);
  fwrite(logFile, sprintf('Generation:            %d \n', g));
  fprintf('Best Fit K Vector is:  ');
  fwrite(logFile, sprintf('Best Fit K Vector is:  '));
  fprintf('[');
  fwrite(logFile, sprintf('['));
  fprintf(' %g', population(1, 1:systemSize));
  fwrite(logFile, sprintf(' %g', population(1, 1:systemSize)));
  fprintf(' ] \n')
  fwrite(logFile, sprintf(' ] \n'));
  fprintf('Fitness is:            %f \n', population(1,systemSize+1));
  fwrite(logFile, sprintf('Fitness is:            %f \n',
population(1,systemSize+1)));

  kVector = population(1, 1:systemSize);

% Transform the system
```

```
  kc = 1/(systemA.c * (-inv(systemA.a - systemA.b * kVector)) * systemA.b);
  sys = ss(systemA.a - systemA.b * kVector, systemA.b, systemA.c * kc,
systemA.d);
% Test the system
[y,t,x] = step(sys);
testResult = stepinfo(y,t);

  poles = sort(eig(sys.a),1,'descend');
  fprintf('Complex Conjugate Pair: %s \n', mat2str(polesVector));
  fwrite(logFile, sprintf('Complex Conjugate Pair: %s \n',
mat2str(polesVector)));
  fprintf('System Poles            %s \n', mat2str(poles));
  fwrite(logFile, sprintf('System Poles            %s \n', mat2str(poles)));
  fprintf('Percent Overshoot     = %g \n', testResult.Overshoot);
  fwrite(logFile, sprintf('Percent Overshoot     = %g \n',
testResult.Overshoot));
  fprintf('Desired Overshoot     = %g \n', percentOS);
  fwrite(logFile, sprintf('Desired Overshoot     = %g \n', percentOS));
  fprintf('Settling Time         = %g \n', testResult.SettlingTime);
  fwrite(logFile, sprintf('Settling Time         = %g \n',
testResult.SettlingTime));
  fprintf('Desired Settling Time = %g \n', settlingTime);
  fwrite(logFile, sprintf('Desired Settling Time = %g \n', settlingTime));
  fprintf('Undershoot            = %g \n', testResult.Undershoot);
  fwrite(logFile, sprintf('Undershoot            = %g \n',
testResult.Undershoot));
  fprintf('Rise Time             = %g \n', testResult.RiseTime);
  fwrite(logFile, sprintf('Rise Time             = %g \n',
testResult.RiseTime));
  solPlot = plot(t,y, 'DisplayName', sprintf('%s fitness', fileStub));
  saveas( solPlot, [fileStub '_solution.fig']);

   fitPlot = plot(1:g, fitnessMatrix (1:g, 1), 'DisplayName',
sprintf('Fitness'));
   title('Fitness Versus Generation');
   saveas ( fitPlot, [fileStub '_fitness.fig']);

   if verbose == 1
      disp('--- --- --- ---');
      fwrite(logFile, sprintf('--- --- --- ---\n'));
      fprintf('%d  unstable solutions discarded for instability \n',
sum(failedMatrix(1:g,1)));
      fwrite(logFile, sprintf('%d  unstable solutions discarded for
instability \n', sum(failedMatrix(1:g,1))));
      fprintf('%d  unstable solutions discarded for poor Lsim results \n',
sum(failedMatrix(1:g,2)));
      fwrite(logFile, sprintf('%d  unstable solutions discarded for poor Lsim
results \n', sum(failedMatrix(1:g,2))));
   end

   fclose(logFile);

end

function [ polesVector ] = calculatePoles (percentOS, settlingTime)
% Determining the complex conjugate pair.
```

```
% Percent overshoot and settling time can be calculated from a complex
% conjugate pair.  The angle and distance from the origin are important to
% that calculation.

% This algorithm works in reverse, calculating the position of the complex
% conjugate pair from the percentOS and settlingTime.

  iV = log(percentOS) ^2;
  z = sqrt(iV / (pi^2 + iV));
  w = log(0.02) / (settlingTime * z);

  x = (z * w);
  y = w * sqrt(z^2 - 1);

% X is the real component of the pole, and Y is the complex component.  The
% two values are used to create a complex conjugate pair

  polesVector(1,1) = x-y;
  polesVector(1,2) = x+y;
end

function [ population, failedVector ] = createPopulations (systemA,
polesVector, populationSize, mutationChance, percentOS, settlingTime,
fitMode)
% The size of the system is calculated
  systemSize = size(systemA.a, 1);

  population = zeros(populationSize, systemSize+1);
  populationIndex = 1;

% FailedVector is for recordkeeping.
  failedVector = [0 0];

% An initial population is created
  while populationIndex <= populationSize
% Randomly seed other values in the poles vector, assuring all are negative
    newPoles = polesVector;

      if systemSize > 2
          newPoles = [ newPoles  -abs(4 * (abs(polesVector(1,1) + randn(1,
systemSize - 2)))) ];
      end
% Find the kVector that will create those poles, using Matlab's place
% command
          kVector = place(systemA.a, systemA.b, newPoles);

% Mutation is introduced
      if rand < mutationChance || systemSize == 2
          for j = 1:systemSize
              kVector(1,j) = subtleMutate(kVector(1,j));
          end
      end

% Transform the system using that vector to see how it performs
      kc = 1/(systemA.c * (-inv(systemA.a - systemA.b * kVector)) *
systemA.b);
      sys = ss(systemA.a - systemA.b * kVector, systemA.b, systemA.c * kc,
```

```
systemA.d);

% Calculate performance, or discard if function fails
      fitness = calcFit ( sys, percentOS, settlingTime, fitMode );
      if isnan(fitness)
          failedVector(1,2) = failedVector(1,2) + 1;
      elseif fitness == -1
          failedVector(1,1) = failedVector(1,1) + 1;
      else

% The resulting "fitness" is added to the end of the vector for use
% in further calculations.
          kVector(1, systemSize + 1) = fitness;

% kVector has passed and is therefore added to the population matrix
          population(populationIndex, 1:systemSize + 1) = kVector;

% The index is incremented.
          populationIndex = populationIndex + 1;
      end % each tested kVector
  end % while the island can hold more kVectors

  population = sortrows(population, (systemSize+1));
end

function [ success ] = displayGeneration ( g, popMatrix )
   success = false;
   [populationSize, vectorSize] = size(popMatrix);

   fprintf('    Generation: %d    Best Fitness: %f    Mean Fitness: %f \n',
g, popMatrix(1,vectorSize), mean(popMatrix(1:populationSize,vectorSize)));
   fprintf('[');
   fprintf(' %g', (popMatrix(1,1:vectorSize-1)));
   fprintf('] \n');
   success = true;
end

function [success] = writeGeneration(g, popMatrix, logFileID)
   success = false;
   [populationSize, vectorSize] = size(popMatrix);

   fwrite(logFileID, sprintf('    Generation: %d    Best Fitness: %f    Mean
Fitness: %f \n', g, popMatrix(1,vectorSize),
mean(popMatrix(1:populationSize,vectorSize))));
   fwrite(logFileID, sprintf('['));
   fwrite(logFileID, sprintf(' %g', (popMatrix(1,1:vectorSize-1))));
   fwrite(logFileID, sprintf('] \n'));
   success = true;
end

function [ mutatedElement ] = subtleMutate ( originalElement )
   mutatedElement = originalElement + 0.6 * randn(1,1);
end % subtleMutate

function [ mutatedElement ] = mutateElement ( originalElement )
   mutatedElement = originalElement + 15 * randn(1,1);
end % mutateVector
```

```
function [ fitness ] = calcFit (systemS, percentOS, settlingTime, fitMode)
%  A complete system is passed in, along with desired criteria.

%  They are then compared to percentOS and settlingTime.  The following
%  calculation is then made:

%  The transformed system is tested for stability.  If any of the poles of
%  A are positive, return a "-1" and exit.

   poles = esort(eig(systemS.a));
   if poles(1,1) > 0
       fitness = -1;
       return
   end % if any of the poles of A are positive, discard K and start over

%  Calculate Settling Time and Percent Overshoot
   [y,t,x] = step(systemS);
   testResult = stepinfo(y,t);

%  Determine the "fitness" of that element of the population

    switch fitMode
        case 3 % weak threshold mode
            % The algorithm is searching for the first kVector that
            % outperforms the criteria.  Since this is a 'weak' search, the
            % algorithm will give a good fitness score if the kVector
            % performs really well on one criterion, even if it doesn't
            % perform well on the other criterion.
            if (testResult.RiseTime < testResult.SettlingTime)
                fitness = (testResult.Overshoot + testResult.SettlingTime) /
(percentOS + settlingTime) + 10 * testResult.Undershoot;
            else
                fitness = (testResult.SettlingTime + testResult.RiseTime) /
(settlingTime + settlingTime) + 10 * testResult.Undershoot;
            end
        case 2 % threshold mode
            % The algorithm is searching for the first kVector that
            % outperforms the criteria.  A kVector must pefrom well in both
            % criteria to achieve a good fitness score.  Only one criterion
            % is insufficient.
            if (testResult.RiseTime < testResult.SettlingTime)
                fitness = max(testResult.Overshoot / percentOS,
testResult.SettlingTime / settlingTime) + 10 * testResult.Undershoot;
            else
                fitness = max(testResult.RiseTime / settlingTime,
testResult.SettlingTime / settlingTime) + 10 * testResult.Undershoot;
            end
        case 1 % weak match mode
            % The algorithm is searching for a kVector that gets within 5%
            % of the user selected criteria.  Since this is a 'weak'
            % search, getting close in one criterion will compensate for
            % poor performance in the other.
            fitness = abs((testResult.Overshoot + testResult.SettlingTime) -
(percentOS + settlingTime))/((percentOS + settlingTime) * 0.05) + 10 *
testResult.Undershoot;
        case 0 % match mode
```

```
            % The default mode.  The algorithm is looking for a kVector
            % that performs within 5% of the user selected criteria.
            % Performance must meet or exceed both criteria to get a good
            % fitness score.
            fitness = max ( abs(testResult.Overshoot - percentOS)/(percentOS
* 0.05) , abs(testResult.SettlingTime - settlingTime)/(settlingTime * 0.05) )
+ 10 * testResult.Undershoot;
        otherwise
    end

%  Return "fitness."  It is possible to return NaN (not a number) if the
%  stepInfo calculation fails.
end

function [ offSpringVector ] = breedingRandomCrossover (populationMatrix)
% Pick two parents at random from the population, and randomly
% determine a number of elements to swap

   [populationSize, systemSize] = size(populationMatrix);

   parentA = populationMatrix(ceil(rand*populationSize),1:systemSize);
   parentB = populationMatrix(ceil(rand*populationSize),1:systemSize);

   parentAmark = ceil(rand*systemSize);
   parentBmark = ceil(rand*systemSize);

   offSpringVector = zeros(2, systemSize);

   offSpringVector(1, 1:systemSize) = parentA;
   offSpringVector(2, 1:systemSize) = parentB;

   offSpringVector(1,parentAmark) = parentB(1,parentBmark);
   offSpringVector(2,parentBmark) = parentA(1,parentAmark);
% Experimentation has proven that some "clean-up" is necessary.
   offSpringVector = offSpringVector(1:2,1:systemSize);
end
```

```
function [ kVector, fitnessMatrix ] = GeneticKIsland( systemA, percentOS,
settlingTime, modeOption )

%  This function seeks to find a transform vector K.  If the system is
%  transformed using the vector K, the resultant system will exibit a
%  specified percent overshoot and settling time or better.

%  Several variables are set here for continued use.
% ================================================================

% INPUT REQUIREMENTS:
% [systemA]         must be a StateSpaceMode
% [percentOS]       is the user's desited percent overshoot
% [settlingTime]    is the user's desired settling time
% [modeOption]      is an optional argument which must be a string.  This
%                   will determine how fitness is calculated


%   Reading the modeOption string, if it exists
%   --- See the calcFit method for more details ---

if nargin == 4
   if ischar(modeOption)
       if strcmp(modeOption, 'weakThreshold')
           % in weakThreshold fitness mode, the algorithm will use the
           % percentOS and the settlingTime arguments as a threshold, and
           % return the first kVector that beats the average of these two
           % criteria
           fitMode = 3;
       elseif strcmp(modeOption, 'threshold')
           % in threshold fitness mode, the algorithm will use the
           % percentOS and the settlingTime arguments as a threshold, and
           % return the first kVector that beats both of these two criteria
           fitMode = 2;
       elseif strcmp(modeOption, 'weakMatch')
           % in weakMatch fitness mode, the algorithm will use the
           % percentOS and the settlintTime arguments as goals, and return
           % the first kVector to come close to the average of these two
           % criteria
           fitMode = 1;
       elseif strcmp(modeOption, 'match')
           % in match fitness mode, the algorithm will use the percentOS
           % and the settlingTime arguments as goals, and return the first
           % kVector to come close to both of these two criteria
           fitMode = 0;
       else
           fitMode = 0;
           fprintf('%s is not a valid modeOption.  Using ''match''
instead.\n', modeOption);
       end
   else
       fitMode = 0;
       fprintf('modeOption is not a string.  Using ''match'' instead.\n');
   end
```

```
end

if nargin < 4
    fitMode = 0;
end

if nargin < 3
    fprintf('   ----------   ERROR   ----------\n');
    fprintf(' Insufficient arguments.  A call to this function must
include:\n');
    fprintf(' a state space system, a desired percent overshoot, a desired
settling time,\n');
    fprintf(' and may optionally include a mode option string.\n');
    fprintf('\n GeneticKIsland( systemA, percentOS, settlingTime, modeOption
)\n');
    return
end

%  The size of the system is calculated
  systemSize = size(systemA.a, 1);
%  The number of ISLANDS is set equal to the systemSize for now.
%  Each ISLAND is a sub-population that has it's own mutation procedure
  islands = systemSize;
%  The number of possible solutions in the total population each generation
  populationSize = 1000;
%  Each ISLAND represents a portion of that total population
  islandSize = ceil(populationSize / systemSize);
%  A Tower is created to hold the best solutions, and serve as a selective
%  breeding ground.
  towerSize = 4*islands;
  tower = 9999 * ones(towerSize, systemSize+1);
%  This is the number of "best" solutions that will migrate to nearby
%  islands.
  migrateCount = ceil(islandSize * 0.05);
%  The number of generations before the algorithm returns the "best result"
  generations = 200;
%  The percent chance that any given new offspring will mutate
  mutationChance = 0.20;
%  If detailed performance information is desired, set this equal to one
verbose = 1;
if verbose == 1
    failedMatrix = zeros(1, 2);
end
    fitnessMatrix = zeros(1, populationSize);
%  The percent of possible solutions that carry unchanged from one
%  generation to the next.  These are always chosen as the "most fit"
  champions = islandSize * 0.2;
%  A watchdog for stagnation
  stagnation = false;

%  Log File for tracking the algorithm's progress, dynamically created.
fileStub = datestr(now,'yyyymmdd_HHMMSS');
logFileName = sprintf('%s.txt',fileStub);
logFile = fopen(logFileName, 'w');

% ============================================================
fwrite(logFile, sprintf('System \nA: = \n%s\nB: = \n%s\nC: = \n%s\nD: =
```

```
\n%s\n', mat2str(systemA.a), mat2str(systemA.b), mat2str(systemA.c),
mat2str(systemA.d)));

[y,t,x] = step(systemA);
systemTest = stepinfo(y,t);

fwrite(logFile, sprintf('Percent Overshoot: %f\nSettling Time: %f\nRise Time:
%f\nUndershoot: %f\n', systemTest.Overshoot, systemTest.SettlingTime,
systemTest.RiseTime, systemTest.Undershoot));
if isnan (calcFit(systemA, percentOS, settlingTime, fitMode))
    fwrite(logFile, sprintf('Current Fitness: Undeterminable \n'));
else
    fwrite(logFile, sprintf('Current Fitness: %f \n', calcFit(systemA,
percentOS, settlingTime, fitMode)));
end


%% CALCULATION OF IDEAL COMPLEX CONJUGATE PAIR %%
polesVector = calculatePoles(percentOS, settlingTime);
fwrite(logFile, sprintf('Pole Vector: %s \n', mat2str(polesVector)));

%% INITIAL POPULATION CREATION %%

%  The current generation is set to one
  g = 1;

%  An initial population of educated guesses need to be made.
%  These are stored within several matrices.  Each row is a whole k vector
%  candidate.  There is an additional column at the end of the row to store
%  the calculated fitness of that k vector.
%  Each population is also sorted so that the "most fit" vectors are on top
%  (index 1) and the least fit are on bottom.

  [island, failedVector] = createIslandPopulations(systemA, polesVector,
populationSize, mutationChance, percentOS, settlingTime, fitMode);

%  Status for Generation One is presented to the user.
  for islandIndex = 1:islands
      fitnessMatrix (g, ((islandIndex-1) * islandSize + 1):(islandIndex *
islandSize))= transpose(island{islandIndex}(1:islandSize, (systemSize+1) ));
      fwrite(logFile, sprintf('Island %d \n', islandIndex));
      fprintf('Island %d \n', islandIndex);
      writeGeneration(g,island{islandIndex}, logFile);
      displayGeneration(g,island{islandIndex});
      tower = migrate(island{islandIndex}, tower, towerSize);
  end
      fwrite(logFile, sprintf('Tower \n'));
      fprintf('Tower \n');
      writeGeneration(g,tower, logFile);
      displayGeneration(g,tower);

  if verbose == 1
      failedMatrix(g, 1:2) = failedVector;
      fprintf('Unstable: %d    Failed Lsim: %d \n', failedMatrix(g,1),
failedMatrix(g,2));
      fwrite(logFile, sprintf('Unstable: %d    Failed Lsim: %d \n',
failedMatrix(g,1), failedMatrix(g,2)));
```

```
  end

   %% BREEDING CYCLE BEGINS %%

%  The algorithm then launches into a cycle.  Rows of the population matrix
%  are "bred" together to produce new "offspring", and then each
%  "offspring" is subject to a percent chance of mutation.  If that new
%  result is capable of trasforming the system into a stable system, and a
%  value for fitness can be calculated, then that result is allowed into
%  the population for the next generation.

  while (tower(1, systemSize + 1) > 1 && g < generations && ~stagnation)
%  The generation count is incremented once at the begining of each loop.
     g = g + 1;
     failedMatrix(g, 1:2) = 0;

%  Each island is handled in series in this program.  Some efforts could be
%  made in future to parallelise this process.

     for p = 1:islands
%  offSpring is the generation created in this loop.  It is created from
%  the last generation.  All members of the current "island{i}" matrix
%  are available for breeding.
        offSpring = zeros(islandSize, systemSize+1);
        offSpringCount = 1;

%  The best fit, the "champions" will be carried to the next generation
%  automatically.  This is their privelege for being the best.
        for (i = 1:champions)
            offSpring(i, 1:systemSize+1) = island{p}(i, 1:systemSize+1);
            offSpringCount = offSpringCount + 1;
        end

%  We continue to breed new offSpring until the # of champions added to the
%  # of offspring generates a new population of the same size as the last
%  generation.

        while (offSpringCount <= islandSize)
%  Potential offspring are created using this method.
            offSpringVector =
breedingRandomCrossover(island{p}(1:islandSize, 1:systemSize));

% offSpringVector may return multiple Children.  Each must be
% tested individually.
            for c = 1:size(offSpringVector, 1)

% Mutation seeks to randomly alter elements within a solution vector.
% This seeks to prevent the solution pool from gathering around a local
% minima, by allowing the algorith to explore some new territory.

% The offspring are subject to a percent chance of mutation first.  Then we
% determine the effects that this new possible K vector would have on the
% system, before deciding whether to add it into the population or to
% discard it.
                if rand < mutationChance
% To keep mutation orthogonal in each island, we mutate all elements in a
% vector a little bit...
```

```
                    for i=1:systemSize
                        offSpringVector(c,i) =
subtleMutate(offSpringVector(c,i));
                    end
% ... then mutate one element a lot.
                    offSpringVector(c,p) =
mutateElement(offSpringVector(c,p));
                end

% Transform the system using the possible new vector
                kc = 1/(systemA.c * (-inv(systemA.a - systemA.b *
offSpringVector(c, 1:systemSize))) * systemA.b);
                sys = ss(systemA.a - systemA.b * offSpringVector(c,
1:systemSize), systemA.b, systemA.c * kc, systemA.d);

% Calculate fitnes score, or discard if a bad result is returned.
                fitness = calcFit ( sys, percentOS, settlingTime, fitMode
);
                if isnan(fitness)
                    if verbose == 1
                        failedMatrix(g,2) = failedMatrix(g,2) + 1;
                    end
                elseif fitness == -1
                    if verbose == 1
                        failedMatrix(g,1) = failedMatrix(g,1) + 1;
                    end
                else

% The resulting "fitness" is added to the end of the vector for use in
% further calculations. newK is added to the population matrix
                    offSpring(offSpringCount, 1:(systemSize + 1)) = [
offSpringVector(c, 1:systemSize) fitness];

% The index is incremented.
                    offSpringCount = offSpringCount + 1;
                end % fitness calculation and handling
            end % each potential solution in the offSpringVector
        end % while loop for breeding cycle
% The new offspring are re-sorted by fitness, and then assigned to the
% island
        island{p} = sortrows(offSpring, (systemSize+1));
    end

% Report Generation's progress to user
    for islandIndex = 1:islands
        fitnessMatrix (g, ((islandIndex-1) * islandSize + 1):(islandIndex *
islandSize))= transpose(island{islandIndex}(1:islandSize, (systemSize+1) ));
        fwrite(logFile, sprintf('Island %d \n', islandIndex));
        fprintf('Island %d \n', islandIndex);
        writeGeneration(g,island{islandIndex}, logFile);
        displayGeneration(g,island{islandIndex});
    end
        fwrite(logFile, sprintf('Tower \n'));
        fprintf('Tower \n');
        writeGeneration(g,tower, logFile);
        displayGeneration(g,tower);
```

```
      if verbose == 1
          fprintf('Unstable: %d    Failed Lsim: %d \n', failedMatrix(g,1),
failedMatrix(g,2));
          fwrite(logFile, sprintf('Unstable: %d    Failed Lsim: %d \n',
failedMatrix(g,1), failedMatrix(g,2)));
      end

      %% MIGRATION %%

%  The most fit solutions from each island are sent to neighboring islands.
%  The receiving island will discard any solutions that are not better than
%  what it currently holds, and also will discard any solutions that are
%  identical to those already in it's population.
      newIsland{1} = migrate ( island{islands}, island{1}, migrateCount );
      newIsland{1} = migrate ( island{2}, newIsland{1}, migrateCount );
%  A very few are also sent to the tower, which keeps the best fit of the
%  total population archived.
      tower = migrate(newIsland{1}, tower, towerSize);

      for i = 2:(islands-1)
          newIsland{i} = migrate ( island{i-1}, island{i}, migrateCount );
          newIsland{i} = migrate ( island{i+1}, newIsland{i}, migrateCount );
          tower = migrate(newIsland{i}, tower, towerSize);
      end

      newIsland{islands} = migrate ( island{islands-1}, island{islands},
migrateCount );
      newIsland{islands} = migrate ( island{1}, newIsland{islands},
migrateCount );
      tower = migrate(newIsland{islands}, tower, towerSize);

      for islandIndex = 1:islands
          island{islandIndex} = newIsland{islandIndex};
      end

   % We check to see if we've had improvement in the past 10 generations
      if g > 10
          improvement = fitnessMatrix(g-10,1) - tower(1,systemSize+1);
          if improvement < fitnessMatrix(g-10,1) * .07
              stagnation = true;
          end
      end

  end % each generation
  %%  COMPLETION AND RESULTS %%

%  Once a solution exceeds the user supplied criteria, or a sufficient
%  amount of generations have passed, the algorithm returns the best
%  calculated solution.  This is represented in the form of a vector of
%  desired poles for the system.

  fprintf('Generation:            %d \n', g);
  fwrite(logFile, sprintf('Generation:            %d \n', g));
  fprintf('Best Fit K Vector is:  ');
  fwrite(logFile, sprintf('Best Fit K Vector is:  '));
  fprintf('[');
  fwrite(logFile, sprintf('['));
```

```
  fprintf(' %g', tower(1, 1:systemSize));
  fwrite(logFile, sprintf(' %g', tower(1, 1:systemSize)));
  fprintf(' ] \n')
  fwrite(logFile, sprintf(' ] \n'));
  fprintf('Fitness is:            %f \n', tower(1,systemSize+1));
  fwrite(logFile, sprintf('Fitness is:            %f \n',
tower(1,systemSize+1)));

  kVector = tower(1, 1:systemSize);

% Transform the system
  kc = 1/(systemA.c * (-inv(systemA.a - systemA.b * kVector)) * systemA.b);
  sys = ss(systemA.a - systemA.b * kVector, systemA.b, systemA.c * kc,
systemA.d);
% Test the system
[y,t,x] = step(sys);
testResult = stepinfo(y,t);

  poles = sort(eig(sys.a),1,'descend');
  fprintf('Complex Conjugate Pair: %s \n', mat2str(polesVector));
  fwrite(logFile, sprintf('Complex Conjugate Pair: %s \n',
mat2str(polesVector)));
  fprintf('System Poles            %s \n', mat2str(poles));
  fwrite(logFile, sprintf('System Poles            %s \n', mat2str(poles)));
  fprintf('Percent Overshoot     = %g \n', testResult.Overshoot);
  fwrite(logFile, sprintf('Percent Overshoot     = %g \n',
testResult.Overshoot));
  fprintf('Desired Overshoot     = %g \n', percentOS);
  fwrite(logFile, sprintf('Desired Overshoot     = %g \n', percentOS));
  fprintf('Settling Time         = %g \n', testResult.SettlingTime);
  fwrite(logFile, sprintf('Settling Time         = %g \n',
testResult.SettlingTime));
  fprintf('Desired Settling Time = %g \n', settlingTime);
  fwrite(logFile, sprintf('Desired Settling Time = %g \n', settlingTime));
  fprintf('Undershoot            = %g \n', testResult.Undershoot);
  fwrite(logFile, sprintf('Undershoot            = %g \n',
testResult.Undershoot));
  fprintf('Rise Time             = %g \n', testResult.RiseTime);
  fwrite(logFile, sprintf('Rise Time             = %g \n',
testResult.RiseTime));
  solPlot = plot(t,y, 'DisplayName', sprintf('%s fitness', fileStub));
  saveas( solPlot, [fileStub '_solution.fig']);

   fitPlot = plot(1:g, fitnessMatrix (1:g, 1), 'DisplayName', sprintf('Island
1'));
   hold on
   for i = 2:islands
      fitPlot = plot(1:g, fitnessMatrix (1:g, ((i-1) * islandSize + 1)),
'DisplayName', sprintf('Island %d', i));
   end
   title('Fitness Per Island');
   hold off
   saveas ( fitPlot, [fileStub '_fitness.fig']);

   if verbose == 1
      disp('--- --- --- ---');
      fwrite(logFile, sprintf('--- --- --- ---\n'));
```

```
      fprintf('%d  unstable solutions discarded for instability \n',
sum(failedMatrix(1:g,1)));
      fwrite(logFile, sprintf('%d  unstable solutions discarded for
instability \n', sum(failedMatrix(1:g,1))));
      fprintf('%d  unstable solutions discarded for poor Lsim results \n',
sum(failedMatrix(1:g,2)));
      fwrite(logFile, sprintf('%d  unstable solutions discarded for poor Lsim
results \n', sum(failedMatrix(1:g,2))));
   end

   fclose(logFile);

end

function [ polesVector ] = calculatePoles (percentOS, settlingTime)
% Determining the complex conjugate pair.

% Percent overshoot and settling time can be calculated from a complex
% conjugate pair.  The angle and distance from the origin are important to
% that calculation.

% This algorithm works in reverse, calculating the position of the complex
% conjugate pair from the percentOS and settlingTime.

  iV = log(percentOS) ^2;
  z = sqrt(iV / (pi^2 + iV));
  w = log(0.02) / (settlingTime * z);

  x = (z * w);
  y = w * sqrt(z^2 - 1);

% X is the real component of the pole, and Y is the complex component.  The
% two values are used to create a complex conjugate pair

  polesVector(1,1) = x-y;
  polesVector(1,2) = x+y;
end

function [ islandMap, failedVector ] = createIslandPopulations (systemA,
polesVector, populationSize, mutationChance, percentOS, settlingTime,
fitMode)
% The size of the system is calculated
  systemSize = size(systemA.a, 1);
% The number of ISLANDS is set equal to the systemSize for now.
  islands = systemSize;
% The number of possible solutions in the population each generation
  islandSize = ceil(populationSize/islands);
% Population is pre-allocated for speed.
  for i = 1:islands
     islandMap{i} = zeros(islandSize, systemSize+1);
  end
% FailedVector is for recordkeeping.
  failedVector = [0 0];


% Each island will create and then mutate an initial population
  for islandIndex = 1:islands
```

```
      islandPopulationIndex = 1;
      while islandPopulationIndex <= islandSize
% Randomly seed other values in the poles vector, assuring all are negative
    newPoles = polesVector;

      if systemSize > 2
          newPoles = [ newPoles  -abs(4 * (abs(polesVector(1,1) + randn(1,
systemSize - 2)))) ];
      end
% Find the kVector that will create those poles, using Matlab's place
% command
          kVector = place(systemA.a, systemA.b, newPoles);

% The goal of dividing the population into islands is that each island will
% hold a population that has mutated along an axis which is orthogonal to
% the others.  This is reflected in the initial population as well as in
% future breeding cycles.

% Mutation is introduced along the axis specific to this island
          if rand < mutationChance || systemSize == 2
            kVector(1,islandIndex) = mutateElement(kVector(1,islandIndex));
          end

% Transform the system using that vector to see how it performs
          kc = 1/(systemA.c * (-inv(systemA.a - systemA.b * kVector)) *
systemA.b);
          sys = ss(systemA.a - systemA.b * kVector, systemA.b, systemA.c *
kc, systemA.d);

% Calculate performance, or discard if function fails
          fitness = calcFit ( sys, percentOS, settlingTime, fitMode );
          if isnan(fitness)
              failedVector(1,2) = failedVector(1,2) + 1;
          elseif fitness == -1
              failedVector(1,1) = failedVector(1,1) + 1;
          else

% The resulting "fitness" is added to the end of the vector for use
% in further calculations.
              kVector(1, systemSize + 1) = fitness;

% kVector has passed and is therefore added to the population matrix
              islandMap{islandIndex}(islandPopulationIndex, 1:systemSize + 1)
= kVector;

% The index is incremented.
              islandPopulationIndex = islandPopulationIndex + 1;
          end % each tested kVector
      end % while the island can hold more kVectors

%  Once the population has been created then all elements are sorted
%  according to their "fitness."  After this process, the matrix
%  "islandMap{i}" will have the "most fit" elements first, and the "least
%  fit" elements last.
      islandMap{islandIndex} = sortrows(islandMap{islandIndex},
(systemSize+1));
  end % for each island in the map
```

```
end

function [ success ] = displayGeneration ( g, popMatrix )
   success = false;
   [populationSize, vectorSize] = size(popMatrix);

   fprintf('    Generation: %d    Best Fitness: %f    Mean Fitness: %f \n',
g, popMatrix(1,vectorSize), mean(popMatrix(1:populationSize,vectorSize)));
   fprintf('[');
   fprintf(' %g', (popMatrix(1,1:vectorSize-1)));
   fprintf('] \n');
   success = true;
end

function [success] = writeGeneration(g, popMatrix, logFileID)
   success = false;
   [populationSize, vectorSize] = size(popMatrix);

   fwrite(logFileID, sprintf('    Generation: %d    Best Fitness: %f    Mean
Fitness: %f \n', g, popMatrix(1,vectorSize),
mean(popMatrix(1:populationSize,vectorSize))));
   fwrite(logFileID, sprintf('['));
   fwrite(logFileID, sprintf(' %g', (popMatrix(1,1:vectorSize-1))));
   fwrite(logFileID, sprintf('] \n'));
   success = true;
end

function [ mutatedElement ] = subtleMutate ( originalElement )
   mutatedElement = originalElement + 0.6 * randn(1,1);
end % subtleMutate

function [ mutatedElement ] = mutateElement ( originalElement )
   mutatedElement = originalElement + 15 * randn(1,1);
end % mutateVector

function [ fitness ] = calcFit (systemS, percentOS, settlingTime, fitMode)
%  A complete system is passed in, along with desired criteria.

%  They are then compared to percentOS and settlingTime.  The following
%  calculation is then made:

%  The transformed system is tested for stability.  If any of the poles of
%  A are positive, return a "-1" and exit.

   poles = esort(eig(systemS.a));
   if poles(1,1) > 0
       fitness = -1;
       return
   end % if any of the poles of A are positive, discard K and start over

%  Calculate Settling Time and Percent Overshoot
    [y,t,x] = step(systemS);
    testResult = stepinfo(y,t);

%  Determine the "fitness" of that element of the population

    switch fitMode
```

```
        case 3 % weak threshold mode
            % The algorithm is searching for the first kVector that
            % outperforms the criteria.  Since this is a 'weak' search, the
            % algorithm will give a good fitness score if the kVector
            % performs really well on one criterion, even if it doesn't
            % perform well on the other criterion.
            if (testResult.RiseTime < testResult.SettlingTime)
                fitness = (testResult.Overshoot + testResult.SettlingTime) /
(percentOS + settlingTime) + 10 * testResult.Undershoot;
            else
                fitness = (testResult.SettlingTime + testResult.RiseTime) /
(settlingTime + settlingTime) + 10 * testResult.Undershoot;
            end
        case 2 % threshold mode
            % The algorithm is searching for the first kVector that
            % outperforms the criteria.  A kVector must pefrom well in both
            % criteria to achieve a good fitness score.  Only one criterion
            % is insufficient.
            if (testResult.RiseTime < testResult.SettlingTime)
                fitness = max(testResult.Overshoot / percentOS,
testResult.SettlingTime / settlingTime) + 10 * testResult.Undershoot;
            else
                fitness = max(testResult.RiseTime / settlingTime,
testResult.SettlingTime / settlingTime) + 10 * testResult.Undershoot;
            end
        case 1 % weak match mode
            % The algorithm is searching for a kVector that gets within 5%
            % of the user selected criteria.  Since this is a 'weak'
            % search, getting close in one criterion will compensate for
            % poor performance in the other.
            fitness = abs((testResult.Overshoot + testResult.SettlingTime) -
(percentOS + settlingTime))/((percentOS + settlingTime) * 0.05) + 10 *
testResult.Undershoot;
        case 0 % match mode
            % The default mode.  The algorithm is looking for a kVector
            % that performs within 5% of the user selected criteria.
            % Performance must meet or exceed both criteria to get a good
            % fitness score.
            fitness = max ( abs(testResult.Overshoot - percentOS)/(percentOS
* 0.05) , abs(testResult.SettlingTime - settlingTime)/(settlingTime * 0.05) )
+ 10 * testResult.Undershoot;
        otherwise
    end

%  Return "fitness."  It is possible to return NaN (not a number) if the
%  stepInfo calculation fails.
end

function [ offSpringVector ] = breedingRandomCrossover (populationMatrix)
% Pick two parents at random from the population, and randomly
% determine a number of elements to swap

   [populationSize, systemSize] = size(populationMatrix);

   parentA = populationMatrix(ceil(rand*populationSize),1:systemSize);
   parentB = populationMatrix(ceil(rand*populationSize),1:systemSize);
```

```
   parentAmark = ceil(rand*systemSize);
   parentBmark = ceil(rand*systemSize);

   offSpringVector = zeros(2, systemSize);

   offSpringVector(1, 1:systemSize) = parentA;
   offSpringVector(2, 1:systemSize) = parentB;

   offSpringVector(1,parentAmark) = parentB(1,parentBmark);
   offSpringVector(2,parentBmark) = parentA(1,parentAmark);
% Experimentation has proven that some "clean-up" is necessary.
   offSpringVector = offSpringVector(1:2,1:systemSize);
end

function [ newTarget ] = migrate ( source, target, quantity )
   if size(source,2) ~= size(target,2)
      fprintf('Source Size %d ~= Target Size %d', size(source,2),
size(target,2));
      newTarget = target;
      return
   else
      newTarget = target;
      rightMost = size(source,2);
      bottom = size(target,1);
      for i = 1:quantity
         j = bottom;
         seek = true;
         destroy = false;
         while seek
            if j == 0
               seek = false;
            elseif newTarget(j, rightMost) > source(i, rightMost)
               j = j - 1;
            elseif newTarget(j, rightMost) <= source(i, rightMost)
               seek = false;
               if newTarget(j, 1:rightMost) == source(i, 1:rightMost)
                  destroy = true;
               end
            end
         end % while we are looking for the place to insert it
         if destroy || j == bottom
         elseif j == 0
            newTarget = [source(i, 1:rightMost) ; newTarget(1:bottom-1,
1:rightMost)];
         else
            newTarget = [newTarget(1:j, 1:rightMost) ; source(i, 1:rightMost)
; newTarget(j+1:bottom-1, 1:rightMost)];
         end % we have inserted or deleted it
      end % for each element in SOURCE
   end
end
```

Vita

Arnold Cassell graduates with a Masters in Electrical Engineering in December of 2012. His undergraduate degree was a B.S. in Computer Engineering from Clemson University in 2004. He has worked in manufacturing, in program management and software compliance for the Navy, and in signal programming for the railroad industry.

He hopes to return to manufacturing after having learned much about controls and robotics here at the University of North Florida.