SPACECRAFT TRAJECTORY OPTIMIZATION SUITE (STOPS): OPTIMIZATION OF MULTIPLE GRAVITY ASSIST SPACECRAFT TRAJECTORIES USING MODERN OPTIMIZATION TECHNIQUES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

by

Timothy J. Fitzgerald

December 2015

COMMITTEE MEMBERSHIP

TITLE:	Spacecraft Trajectory Optimization Suite (STOpS): Optimization of Multiple Gravity Assist Spacecraft Trajectories Using Modern Optimization Techniques
AUTHOR:	Timothy J. Fitzgerald
DATE SUBMITTED:	December 2015
COMMITTEE CHAIR:	Kira Abercromby, Ph.D. Associate Professor of Aerospace Engineering
COMMTTEE MEMBER:	Rob McDonald, Ph.D. Professor of Aerospace Engineering
COMMITTEE MEMBER:	Eric Mehiel, Ph.D. Professor of Aerospace Engineering
COMMITTEE MEMBER:	Gerald Shaw, Ph.D. SRI International, Inc.

ABSTRACT

Spacecraft Trajectory Optimization Suite (STOpS): Optimization of Multiple Gravity Assist Spacecraft Trajectories Using Modern Optimization Techniques

Timothy J. Fitzgerald

In trajectory optimization, a common objective is to minimize propellant mass via multiple gravity assist maneuvers (MGAs). Some computer programs have been developed to analyze MGA trajectories. One of these programs, Parallel Global Multiobjective Optimization (PaGMO), uses an interesting technique known as the Island Model Paradigm. This work provides the community with a MATLAB optimizer, STOpS, that utilizes this same Island Model Paradigm with five different optimization algorithms. STOpS allows optimization of a weighted combination of many parameters. This work contains a study on optimization algorithm performance and how each algorithm is affected by its available settings.

STOpS successfully found optimal trajectories for the Mariner 10 mission and the Voyager 2 mission that were similar to the actual missions flown. STOpS did not necessarily find better trajectories than those actually flown, but instead demonstrated the capability to quickly and successfully analyze/plan trajectories. The analysis for each of these missions took 2-3 days each. The final program is a robust tool that has taken existing techniques and applied them to the specific problem of trajectory optimization, so it can repeatedly and reliably solve these types of problems.

ACKNOWLEDGMENTS

I would like to thank Dr. Kira Abercromby for her continued support and guidance through this whole process. This work would not have been possible without her advice and patient assistance. I never would have been able to narrow my topic down without first drawing that line in the sand.

I would also like to thank Dr. Rob McDonald for his optimization advice and tutorage over the past year and a half, Dr. Eric Mehiel for his patience with me, and Dr. Gerald Shaw for his continual guidance, support, and encouragement over the past two years.

I would like to thank my parents and my sister for their continued support, encouragement, and forgiveness for my sporadic lapses of communication.

Lastly, I would like to thank my “AERO Grad Roomies” Michelle Haddock, Michael Strange, and Chris Barlog for helping make my graduate experience as enjoyable as it was, and also for putting up with my constant updates of “how awesome my GUI is”.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Statement of Problem

With many engineering problems, not just any solution will do. It is desirable and often required to find the best solution within reason: the ‘optimal’ solution. The field of optimization originated within the field of calculus; the objective was to find the minima or maxima (the optima) of calculus functions. These functions typically contain a small number of variables and have well-known derivatives. The derivatives can be used to easily identify the optima of the function, and the second derivatives can be used to identify whether these optima are maxima or minima. The absolute, most optimal solution is often difficult to find in real-world engineering problems, which typically have large quantities of variables and functions/derivatives that are not clearly defined. Additionally, once an optimal solution is found, it is even harder to determine if that is the absolute best (the global optimum) or just a very good solution (a local optimum).

Various methods have been developed to find global and local optima over a wide range of search spaces, and the best method to use for a particular problem is an optimization problem of its own. A particularly interesting method to tackle this conundrum is the Island Model Paradigm [7], where multiple methods run simultaneously and continuously compare solutions. This allows the different methods to play off each other’s strengths, building a more robust optimization tool. This work serves to utilize this model with other existing optimization algorithms in a new tool to optimize spacecraft trajectories.

1.2 Purpose of Study

Each spacecraft trajectory poses an incredibly difficult optimization problem to mission designers. The search space is often immense with an unknown landscape, and designing algorithms for particular problems is time consuming. Running these algorithms can be computationally expensive. As the exploration of space continues to grow, the desire to quickly and efficiently find the best available trajectories grows as well. The need for an effective and robust optimization algorithm that returns the global optimum within a reasonable amount of time is continually increasing. This thesis will provide an open source solution to the problem, available to universities, industry, and individuals interested in the field of trajectory optimization. The work does not serve to develop new optimization methods, but instead to take existing techniques and develop a tool that can repeatedly and reliable solve the specific problem of spacecraft trajectory optimization.

1.3 Literature Review

Some other programs exist that perform a similar function to the suite presented in this work. Most existing programs are proprietary or expensive. Some examples of existing programs that come with a hefty price tag are BullsEye [21], COPERNICUS [22], Mission Analysis Environment for Heliocentric High-Thrust Missions [23], and Mixed Integer Distributed Ant Colony Optimization [24].

There are also some programs that are publicly available, free of charge. One is called Java Astrodynamics Toolkit [25]. This tool is written in Java, and is exactly what the name implies: multiple individual functions that can be used for mission

analysis. This includes orbits, ADC, optimization, etc. The program requires some digging and manipulation of existing code before it can be applied to any particular trajectory problem.

Another program is called Skipping Stone [26]. This program is a MATLAB user interface developed for a master's thesis that analyzed the possibility of a mission to the solar bow shock. The mission kept the time of flight under 15 years and tried to keep the spacecraft dry mass as close to 500 kg as possible. Skipping Stone utilized four stochastic methods, but the user's control over these algorithms is limited and the number of flybys is limited to four.

One last program is called Parallel Global Multiobjective Optimization (PaGMO) [27]. PaGMO was developed by Dario Izzo *et al.* of the European Space Agency. This program was supplemented by a Cal Poly Thesis by Jason Bryan, titled "Global Optimization of MGA-DSM Problems Using the Interplanetary Gravity Assist Trajectory Optimizer (IGATO)" [28]. PaGMO is a C++/Python program that is generic to optimization. It does not have an overarching interface, and like JAT it requires some manipulation and code building to use for trajectory optimization problems. This is what Jason did. He built a user interface around PaGMO specific to spacecraft trajectory problems, and added in dynamic restart capabilities, a pruning algorithm, and subdomain decomposition.

1.4 Structure of Paper

This paper first gives an introduction/refresher on orbital mechanics in Section 2. For readers with experience in this area, this section can be skipped. Section 3 then dives into the general field of optimization. This section also

includes more detail on each of the five algorithms utilized in the suite. Even the reader who is familiar with the algorithms presented here is encouraged to read this section, as this work employs particular aspects of each algorithm that may not be the reader's understood 'standard'. Following an explanation of optimization, Section 4 details how the Island Model Paradigm works. With the workings of STOpS explained, Section 5 shows the verification process used for each algorithm on known test functions. After the ideal default parameters for each algorithm have been presented, Section 6 shows the results of two specific test cases. Lastly, Section 7 briefly shows the GUI used in this work, followed by the conclusions drawn in Section 8.

All images, unless otherwise cited, were generated by the author using MATLAB 2015a or 2015b.

1.5 Acronyms

ACO – Ant Colony Optimization

ACS – Ant Colony System

AS – Ant System

DE – Differential Evolution

EAS – Elitist Ant System

GA – Genetic Algorithm

GUI – Graphical User Interface

LS – Local Search

MMAS – Min-Max Ant System

PSO – Particle Swarm Optimization

SOI – Sphere of Influence

STOpS – Spacecraft Trajectory Optimization Suite

TSP – Traveling Salesman Problem

2. ORBITAL MECHANICS REVIEW

Contrary to the common misconception, objects in orbit are not in ‘zero-g’. Gravity is actually the main force acting on an object in orbit, and without it, the orbit would not exist. Objects in orbit are simply traveling fast enough to escape the pull of gravity, but slow enough that they do not leave the planet entirely. Consider the cannonball example, proposed by Isaac Newton [2] and shown in Figure 2.1 [1].

If a cannonball is shot from the North Pole, it will eventually hit the ground (trajectory A). If it is shot faster, it will go further before it hits the ground (trajectory B). Eventually, the ball will be shot fast enough that by the time it falls to where it would have hit the ground, it is beyond that point horizontally (trajectory C). It then continues its motion; it is in orbit. If the ball is shot faster, then on the other side of the Earth it gets even further away, but it is still within Earth’s gravity so it still gets pulled back (trajectory D). All trajectories described so far return back to the original height of the cannon. If the ball is shot fast enough, it will no longer return. Instead, it has enough energy to get far enough away from the Earth that it can escape Earth’s gravitational pull (trajectory E).

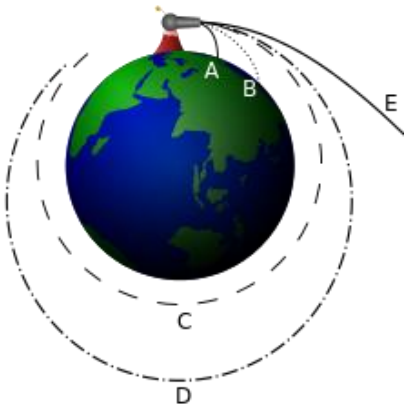


Figure 2.1. Newton’s Cannonball

2.1 Planetary Flybys & ΔV Maneuvers

Consider trajectory E from Newton's cannonball example. If time were propagated backwards from the shot out of the cannon, there would be a mirrored escape trajectory going in the opposite direction. If an object enters a celestial body's sphere of influence (SOI) with enough energy, then it will continue until it escapes at a different location. Within the celestial body's reference frame, the magnitude of the object's velocity will be the same at the entry and exit points of the SOI, but the direction will be different. When examining an interplanetary trajectory, this process takes place so quickly and in such a small area (relative to the entire trajectory/problem), that it can be reasonably approximated as an impulsive ΔV maneuver [5]. This is called "patched conics", and allows these trajectory problems to be broken up into individually solvable parts. The core of these problems, gravity assist maneuvers or "flybys", can be seen below in Figure 2.2 [5].

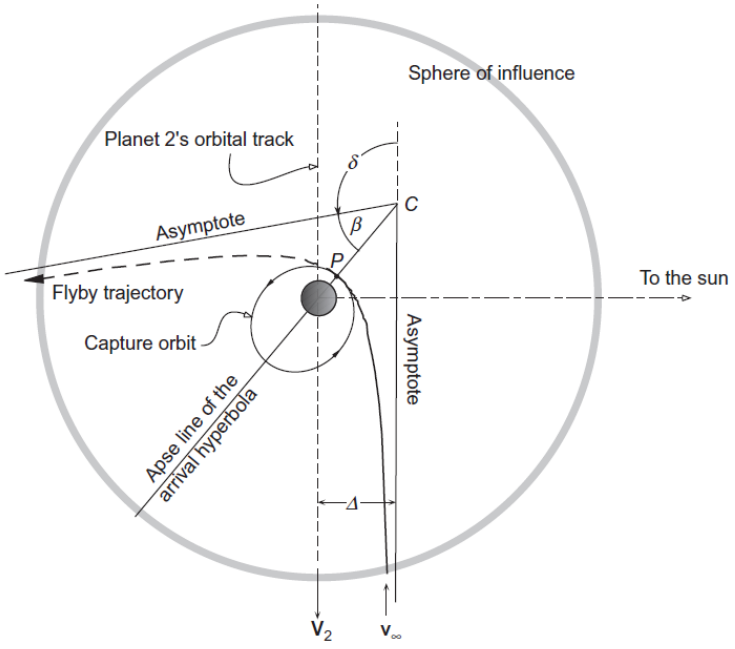


Figure 2.2. Planetary Flyby Maneuver

Keeping in mind that one of the main variables being optimized for these spacecraft trajectories is fuel mass, it is easy to understand why these ΔV maneuvers are valuable: they are essentially “free”. If the spacecraft is placed on the right flyby, the over-arching gravity assist trajectory can be altered while expending no fuel. Even though the magnitude of the craft’s velocity (relative to the celestial body), has not changed, the vector itself (its direction) has. In the larger reference frame (outside of the flyby body’s reference frame), the direction has also changed. Based on how the spacecraft has approached and leaves the body it is flying by, the magnitude of the spacecraft’s velocity can actually be changed as well.

3. OPTIMIZATION

Optimization can be simply stated as finding the best solution to a given problem. In general, the only difference between finding a maximum and minimum with the algorithms used in this work is applying a negative sign to the value of the cost function being optimized. For clarity, all explanations in this section refer to the minima of a function as the function's optima.

In real-world engineering problems, optimization can be a difficult process. These problems typically have many variables, and the bounds for these variables can span a large range of values; they have an immensely large search space. Trying to define what makes a particular solution the "best" is tough as well. For a problem with *n* variables, the search space spans *n-dimensional* space. This makes the search space impossible to visualize if *n* is greater than 3. For the sake of discussion this section will deal with only 2-dimensional problems in order to more effectively and visually explain the processes and algorithms. All processes and algorithms apply to these 2-dimensional problems in the same way they apply to problems with more dimensions without any loss of generality.

This section serves to explain the optimization algorithms and processes utilized in this work. All processes used were taken from literature. However, various implementations of the algorithms and processes have been found, so it is necessary to establish exactly which elements have been taken from literature and implemented here. The only optimization process that is unique to this work is the particular application of Ant Colony Optimization to spacecraft trajectory problems.

3.1 Local vs. Global

As mentioned earlier, when a good solution is found, it is sometimes difficult to determine if it is just a really good solution (a local optima) or the actual best solution (the global optimum). It is important to understand the difference between these two, as certain methods work better at finding one as opposed to the other. For example, local search optimization specializes in finding local optima, whereas genetic algorithms have the ability to move from optima to optima without always getting stuck in the first one they find. As a result, they have a chance to actually find the global optimum; local search methods will only find the global optimum if they are initially placed in that optimum's basin. Ackley's function [18], whose two-dimensional version is shown in Figure 3.1 below, serves as an excellent example of the two types of optima.

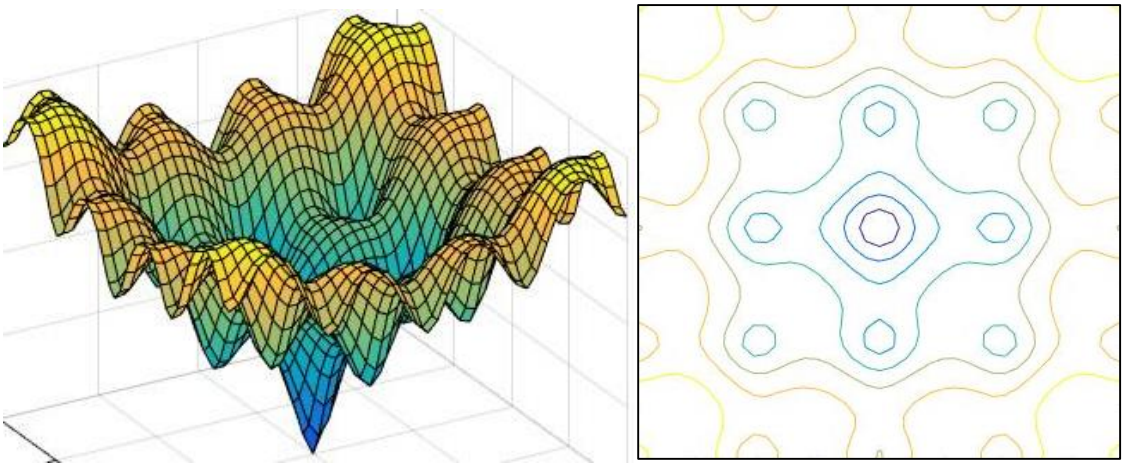


Figure 3.1. Ackley's Function (surface plot on left, contour plot on right)

Ackley's function can be evaluated in any number of dimensions, as can be seen in Eq. 3.1, where *d* is the number of dimensions. The two-dimensional version will be used as the example function when explaining the various optimization methods used in this work, with $a=20$, $b=0.2$, and $c=2\pi$.

$$f(\boldsymbol{x}) = -a * exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i{}^2}\right) - exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + exp(1) \quad (3.1)$$

A function like this is called multi-modal, because it contains many optima. Every bottom of a basin is a local optima; it is better than a group of points immediately around it. However, there is only one global optima, or point that is better than all the other points. The tricky part in telling the difference is that by definition, the global optimum is also a local optimum, so there is no clear cut difference (unless the search space is fully defined as it is in Ackley's function or most other 2-dimensional problems). This particular function has its global minimum value at 0 when all elements of $\boldsymbol{x}$ are 0.

3.2 Local Search Methods

A basic, but effective, set of methods often used in optimization can be generalized to be called local search methods. A gradient-based method is a local search method that uses the gradient (a.k.a. the slope or derivative) of a cost function. When dealing with minimization problems, this means that the optimizer will travel "down" until it cannot go down any further. In the case of Ackley's function, it is as if one were to place a drop of water randomly somewhere in the search space; it would slide down into the bottom of whichever basin it landed in. This is why local search methods are very good at finding local optima as opposed to global optima: they go to the nearest good solution and stop. It would be quite surprising if the water shot all the way back out the other side of the basin!

That particular path in the water droplet example reflects a specific gradient-based method called steepest descent [6]. Steepest descent is more efficient than

other local search methods, but it requires knowledge of the function's gradient to actually be used.

Another way to define a good search direction is to simply take one variable and see if changing it (either increasing or decreasing) improves the solution. Whatever change makes the cost decrease is the new search direction. The process is then repeated for the next variable. This method (holding all variables constant except for one at a time) is called univariate search, or one at a time search [6]. This method is not as effective as steepest descent since directly along any one axis is rarely the direction of most improvement [6], but it does allow some progress in functions with undefined derivatives. A benefit of this search is that since all sequential search directions are perpendicular, the function will not undo any of its progress on its next step.

Methods like steepest descent and univariate search are used to find the best direction to look for a minimum, but there is another step to actually finding the minimum. When a direction is chosen, the problem is essentially converted into a one-dimensional problem: the optimizer can travel along the chosen search direction until a minimum is found. Binary bracketing and golden sectioning techniques [6] are used to actually locate the minimum along the chosen search direction. Binary bracketing 'feels' out along the search direction until one point has a cost value lower than both the initial point and a third point (that is further away, along the search direction). This scenario guarantees that some minimum exists along that bracket. Golden sectioning then continually shortens the bracket

that contains the optima until the bracket has become smaller than some preset tolerance.

Once the minimum is found, the process to choose a new search direction is repeated. Once the optimizer cannot find a direction that will improve the solution, an optima has been found. It should be noted that in the univariate search in this work, when the direction of improvement is decided, the test point along that direction is used as the initial point in binary bracketing. This can be seen in the following example. Since Ackley's function's derivative is not well defined at all points, the example will use univariate search to define the direction and golden sectioning to find the minimum along each direction.

A point is randomly chosen in the search space: (-1.4,0.75). The first direction examined is the X direction. It can be seen that increasing X leads us to an improvement in the function's cost value. The optimizer continues to travel in that direction until it senses that it is moving "up". After golden sectioning has been applied, the minimum of that line has been found to lie at (-0.9392,0.75). Moving next in the Y direction, the process is repeated and the optimizer finds itself at the point (-0.9392,0.9682). Finally, one more improvement in the negative X direction places the optimizer on its final point, (-09695,0.9682). Increasing or decreasing either the X or Y value from this point increases the cost, instead of decreasing it. Therefore, an optima has been found. The path can be seen below in Figure 3.2.

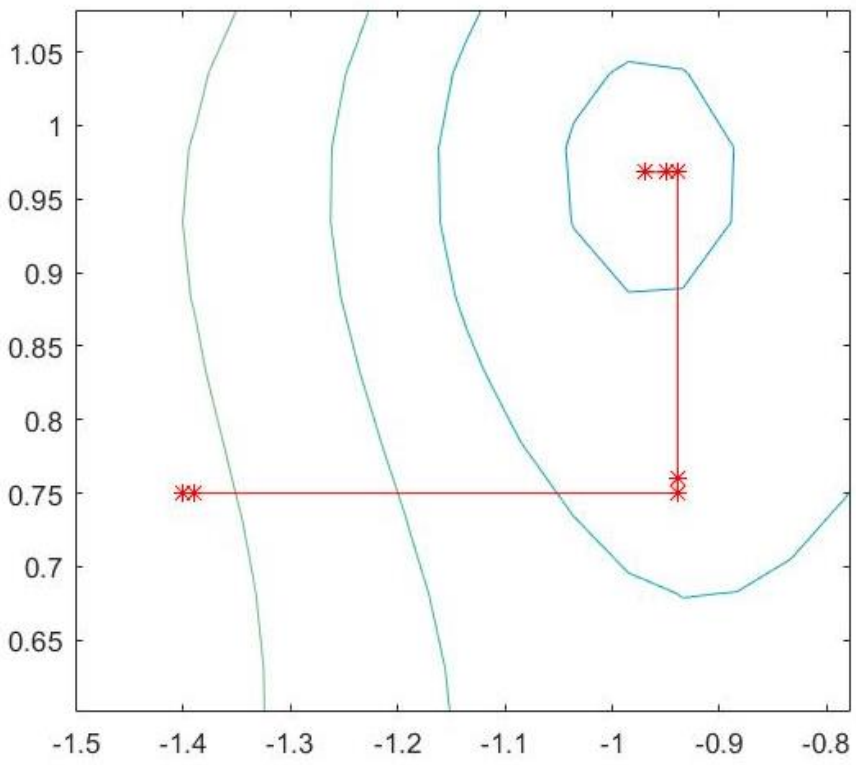


Figure 3.2. Univariate Search, Ackley's Function

Since the search space for Ackley's function is known, it is obvious that a local minimum has indeed been reached, and the global minimum has been completely ignored.

3.3 Genetic Algorithms

One of the global optimization methods used in this work is that of the Genetic Algorithm (GA). This work follows the methods found in two sources: "Genetic Algorithms in Search, Optimization, & Machine Learning" by David E. Goldberg [9], and "Practical Genetic Algorithms" by Randy L. Haupt and Sue Ellen Haupt [10]. This section serves as a summary of the techniques presented in those books, as well as the adaptations required for the particular problems tackled here. The interested reader is encouraged to reference these books for more detailed explanations.

Essentially, GAs (a subset of evolutionary algorithms) mimic biological optimization similar to the theory of Darwin and survival of the fittest. GAs start with a span of random solutions (a *population*) and use some *selection* method to decide which solutions to use in *mating* to create a new group of solutions (the next *generation*). This process continues until the best member of the current generation has a solution that meets some criteria, or the algorithm can run for a fixed number of generations.

GAs are a type of stochastic method; they rely heavily on randomness to effectively search the entire search space. This randomness, when combined with the survival of the fittest mentality, works well in optimization. As described by Goldberg, “[w]hile randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance” [9]. An example of GA’s progress can be seen in Figure 3.3. The processes will be explained in the following sections.

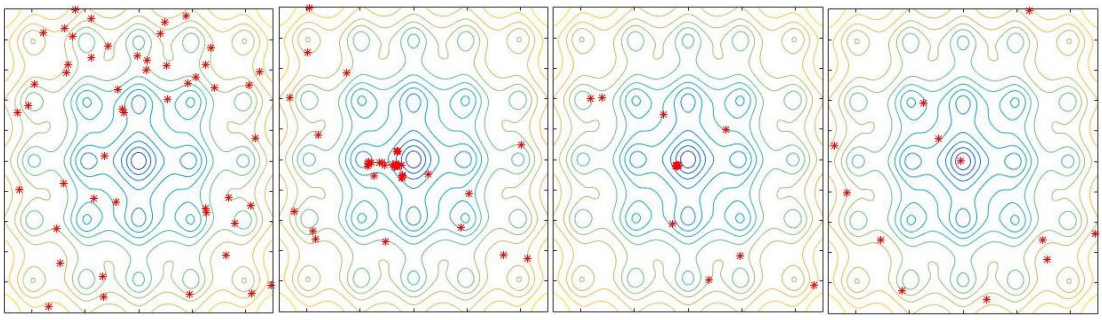


Figure 3.3. Progress of a GA (left to right: generations 1, 3, 6, 20)

3.3.1 Binary vs. Continuous

GAs can be divided into two distinct groups: *binary* and *continuous*. These terms deal directly with the variables used to solve the problem, or more directly, the cost function. Original GAs were binary. In binary GAs, population members

are strings, each made of binary values of all the variables in succession. For example, if the cost function has two variables that are integers between 0 (binary representation 000) and 7 (binary representation 111), a possible population member could be (2,5). This would be represented as the string 010101, where the 010 is the 2 and the 101 is the 5.

This type of variable representation poses some obvious concerns for modern optimization problems. The search space is discrete; the system above would only be able to look at integer values. In order to obtain the precision needed for many real world problems (whether that means a larger range of integers, or floating point decimal numbers), the length of the string must be quite large, especially when high precision is paired with a multi-dimensional cost function.

The solution to this conundrum is the continuous GA. In this type of variable representation, the variables are free to be any value between specified upper and lower bounds, with precision limited only by computational programming. Aside from how variables are represented, the main difference between these two methods is how population members are mated to create the next generation, which will be discussed later.

3.3.2 Selection Methods

The selection method in GAs has a noticeable effect on the algorithm's performance. A good selection method does not eliminate areas of the search space too quickly (avoiding premature convergence), yet accurately guides the algorithm to the global optimum in a reasonable amount of function evaluations. There are multiple choices for how to choose which members of a population to

keep and use for mating. Obviously, *random selection* is a choice. It does not guarantee any progress, but given a large enough population, it could potentially discover some local, or even the global, optima.

Another more useful method is called *natural selection* due to its similarity to the process in nature. In this method, the population is sorted and arranged from best to worst. The best members (exactly how many is up to the user) are kept and advance to the next generation. It is the same as the survival of the fittest, as if the weaker members of the population have died out; they were not strong enough to survive. After the best members have been selected, they continually mate and produce offspring until the population has been replenished. For example, if a population contains 100 members, and the best 20 are selected to advance, then these 20 members mate until 80 more offspring have been produced, thus giving the next generation the required 100 members. More details on how the mating process works will follow.

Another option for selection methods is known as *thresholding*. In this method, the number of population members that survive fluctuates; any member with a cost below the threshold survives, advances, and is used in mating. By itself, this solution can only guarantee a solution as good as the cost of the threshold. Eventually, members will all have costs below the threshold and the number of members selected to advance will be equal to the size of the population, which leaves no room for new offspring. The difficulty with this method is that the user must have some realistic idea of the cost of an optimal solution. If no members meet the threshold requirement, then the algorithm needs to start over with an

entirely new randomized population. A workaround for this problem is setting a variable threshold. In the case when some solutions actually meet the threshold, as soon as a certain percentage of the population meets the threshold, the threshold can be lowered. In the case when no population members meet the threshold, the threshold can be raised.

The last method for selection is called *roulette selection*. In this method, each member of the population is assigned a probability, with the better solutions receiving the higher probabilities. No population members are automatically eliminated in this method. While at first this may seem counterproductive, in reality this is the only way to ensure that no part of the solution space is ignored. This gives all areas at least a small chance for survival, which could be useful in functions that have a noisy search space and/or isolated optima. Each time the mating procedure is executed, two members are selected based on their probabilities, analogous to spinning a roulette wheel [9].

The roulette selection method has two options for assigning probabilities to the population members; weight based on rank, or weight based on cost. The former is a “blinder” approach; it allows solutions a better chance to survive, because the probabilities are the same whether the fitness values are similar or distinct. With the latter option, if there is a solution that is considerably better than the rest, in a cost based probability scenario it is highly probable that this solution will dominate the mating process, and could potentially lead the algorithm to premature convergence.

3.3.3 Mating Methods

After solutions have been selected and added to the mating pool, there are a few options as to how the mating will be performed. Just as in nature, it is possible for an offspring to inherent a trait identical to one of its parents, but it is also possible for its other traits to be mixtures of its parents.

When mating, the stark difference between a binary and continuous GA becomes even more apparent. Binary mating uses a method called *crossover*, which is similar to DNA exchange. Single bits, or chunks of the binary strings, can be exchanged to change the value of the variables. This allows the algorithm to explore new areas of the search space. With this method, the resulting new variables in the offspring may or may not be between its parents' values; it could also be a direct copy of its parents' values. To perform crossover, the number and location of 'break points' must be chosen to determine the crossover pattern. For example, consider two parents at (2,5), or 010101, and (1,0), or 001000. If two break points are desired, and their locations are between bits two & three and bits five & six, respectively, then the children would be 011001, or (3,1), and 000100, or (0,4).

Continuous GAs can also follow a crossover pattern, but since their variables are just numbers, not binary strings, the crossover only allows switching of whatever values were initially generated. This means that new information cannot be introduced (except via mutation, which will be discussed next), severely limiting the areas available for exploration. To circumvent this issue, typically continuous GAs use a different mating method. Instead of literally exchanging bits,

population members of these problems give values to their offspring that are between their own values. This process is accomplished via equation (3.2) [9], where β is a random number between 0 and 1. This equation is applied to each variable amongst the parents, not the vectors as a whole.

$$offspring = \beta * parent1 + (1 - \beta) * parent2 \quad (3.2)$$

Going back to the two parents used in the example above, the children could potentially have values (1.63,4) and (1,2.5). This way of mating is called blending, since traits of both parents are combined to form some new combination of the two [10]. One downside to this method is that future information is limited to values only between these two parents. To get around this issue, if the range of β is extended to be from –δ to 1+δ, the range of potential new information extends beyond the parents.

3.3.4 Mutation

No matter what type of GA, which selection method, or which mating method is chosen, all have the chance for mutation. Typically the probability for mutation is around 20%. Sometimes, an algorithm can converge prematurely. In this case, if a mutation is introduced into the system, a new area can be brought back into consideration. After the mating process is finished, each variable of each child has a chance to undergo mutation; it has a chance to become a new, totally random value somewhere between the previously defined variable bounds. One may assume that this could diminish the algorithm's performance. On the contrary, if the mutation creates a population member with a higher cost, then that population will end up not making it to the next generation; no negative progress

will have been made. The possibility for it to help outweighs the chance that it will hurt.

In the literature, the mutation method for continuous GAs gave each individual variable a chance for mutation [9]; that is to say that it is possible for only one or some of a population member's variables to be mutated. In this work, it was found that this mutation did not adequately explore the search space. In two-dimensional functions specifically, this only allowed exploration mostly in a cross (along the two dimensions) about the current main grouping of points. This problem is shown on the left side of Figure 3.4. To allow mutation to promote adequate exploration, if the mutation probability was met, then the population member's entire set of variables was allowed to mutate. The effect of this modification can be seen on the right side of Figure 3.4, and this work found this method to be more successful overall. In this particular example, the algorithm had already reached success so exploration was not necessary, but in harder functions, when the GA may temporarily lie in only a local optimum, the continued exploration is quite important to avoid premature convergence.

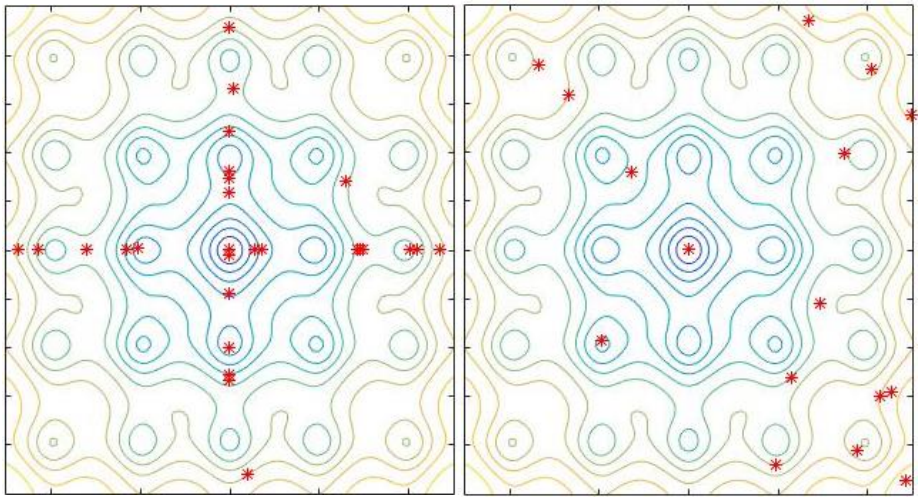


Figure 3.4. Mutation Methods (one variable on left, all variables on right)

This work also implements a new form of mutation that was not found in the literature. In many test cases, the GA arrived in the basin of the global optimum, but not directly on the value. A standard solution to this problem would be to apply a local search method at the final solution of the GA. However, when using local search methods on a function whose derivative is unknown, it is incredibly difficult to know which direction will lead to improvement. Furthermore, even if the correct direction is known, it is difficult to know how far to travel in that direction. The workaround presented in this work relies on the GA eventually having multiple members of the population as identical solutions. When this is the case, if two identical members are selected for mating, then both offspring just match them in traditional methods. Even with the extension of β mentioned in the previous section, there will be no new exploration if the parents are identical. So, in this work if the two parents are indeed the same, instead of them mating, one parent is slightly perturbed. Each variable has a chance to get perturbed between 0-δ% of the span of the original bounds. This mimics the process of a random march local search, and was found to improve the solution found by the GA in every test case.

3.4 Differential Evolution

The method of Differential Evolution (DE) used in this work was adapted from one source: “Differential Evolution: A Practical Approach to Global Optimization” by Kenneth Price, Rainer M. Storm, and Jouni A. Lampinen [12]. This section serves as a summary of the techniques presented in that book, as well as the adaptations required for the particular problems tackled here. The interested reader is encouraged to reference that book for more detailed explanations.

The process that DE follows is roughly similar to GAs; they both generate a random initial population, then from that population they choose certain members to change (create offspring), then they decide which of all these possibilities move on to the next generation. This process is repeated until either convergence is met or a certain number of function evaluations has been exceeded. The difference between the methods lies in how they select members (both before and after the "mating" process), and how they actually change ("mutate") those members.

3.4.1 Mating/Mutation

In DE, there is not really a mating process, or any process that is analogous to nature. Instead, it is referred to as mutation. The mutation in DE uses equation (3.3) [12]. In that equation, *V* is referred to as the mutant vector. All of the *x* vectors are members from the current population. The subscript *r0* is for the base vector, and the subscripts *r1* and *r2* are for the vectors used to create the difference vector. The variable *F* is the scaling factor that is applied to the difference vector.

$$\vec{V}_i = \vec{x}_{r0} + F(\vec{x}_{r1} - \vec{x}_{r2}) \tag{3.3}$$

In every generation of DE, the same number of mutant vectors are generated as there are original population members. These mutant vectors are not necessarily the vectors that are able to move on to the next generation. After each mutant vector has been formed, there is a chance for crossover; a probability that the trial vector (the one up for survival) will either take its trait (aka variable) from the i^{th} mutant vector, or from the i^{th} original population member. After all the trial vectors have been created, there is a group of vectors that is twice the size of the

original population. Half of these members survive until the next generation. The method of survivor selection will be discussed in a later section.

The other option DE offers for customization besides selection is the value chosen for *F*. In the literature it was found that a value of *F* between 0.4 and 1 should work for most functions [12]. There are also two other options (besides that of a constant *F*), which will be discussed later.

3.4.2 Selection of Base Vector & Difference Vector Contributors

The basic methodology and exploration strategy of DE is to get overall trends from the search space. For that reason, the selection of vectors to create the difference vector is always random. A vector from the original population can be used any number of times for any number of difference vectors. The only rule is that the base vector, difference vector contributors, and original population vector must all be unique.

The literature presented three methods for choosing the base vector. The first is random selection, where “[a]ll vectors serve as base vectors once and only once per generation” [12]. They are continually selected at random, with no “regard for their objective function value” [12] until none are left. The next option is to use only the best so far solution as the base vector for every mutant vector generated. The last option is a combination of the two. For each mutant vector created, the base vector is some vector on the line between the best so far vector and a random vector. Exactly where on this line the vector lines is a random chance between 0 and 1 (corresponding to 0% through 100% of the distance from the best so far vector to the randomly chosen vector).

3.4.3 Selection of Survivors

After all trial vectors have been created, the DE algorithm is left with a group of potential survivors that is twice as large as the original population. There are two ways to determine which members survive. The first and most obvious option is to only allow the best members to survive. That is to say that the half of the population that has the best objective function values will move on to the next generation, and the rest will be discarded.

The other option for selecting survivors is called tournament selection. In this method, each vector is paired up against *T* other vectors. If the current vector has a better objective function value, then it is assigned a win. After all vectors have finished their competitions, the half with the most wins moves on to the next generation. This method gives some chance for suboptimal solutions to move through. These selection methods both make DE an elitist method. Neither method can discard the best solution.

The literature had mentioned that any selection process used for parent selection in evolutionary algorithms and GAs could work in DE. However, when cost weighted random and rank weighted random selection of survivors was employed, no combination of parameters led the algorithm to higher than a 15% success rate, and so it was not employed later.

3.5 Particle Swarm Optimization

The method of Particle Swarm Optimization (PSO) used in this work was adapted from one source: “Particle Swarm Optimization” by Maurice Clerc [17]. This section serves as a summary of the techniques presented in that book, as

well as the adaptations required for the particular problems tackled here. The interested reader is encouraged to reference that book for more detailed explanations.

PSO can be imagined as the optimization of bees. The bees essentially begin their search for pollen by flying randomly. As bees find areas with flowers, they communicate these optimal areas to the other bees, until essentially all bees know where the best area to find flowers is. In this scenario, the bees are flying to and from a set location: their hive. In optimization, it does not add anything to the algorithm to have solutions move to and from a set location. Instead, in PSO, particles are given a random initial position and a random initial velocity, and from there they explore the workspace. The particles communicate with each other, and a few things influence a particle's velocity: its own velocity, the best solution it has found, and the best solution that a different particle has told it about.

3.5.1 Particle Motion

As soon as the algorithm starts, the particles all have a velocity. One of the inputs to the algorithm is the maximum velocity. When choosing the initial velocity values for the particles, a value between the negative and positive maximum velocity is chosen. A typical value for the maximum velocity is one half the length of the search space for each dimension. Given a particle's initial position and velocity (if acceleration is, for the time being, ignored), the particle will eventually leave the boundaries of the search space. When this happens, the particle's position is set to be the edge that was crossed, and the velocity's sign is changed. This keeps all particles within the set solution space.

However, the acceleration of the particles is what makes this algorithm interesting. There are three aspects that affect a particle's movement: its own current velocity (*v*), its knowledge of the best solution that it has seen (location denoted by *p*), and the knowledge of the best solution that an informant particle has seen (location denoted by *g*). With these three pieces of information, the particle's path is altered using Eq. 3.4 [17]. In this equation *x* corresponds to the particle's position and the three *c* terms correspond to the particle's confidence in each of the respective pieces of information it has.

$$v_{new} = c_1 v + c_2(p - x) + c_3(g - x) \tag{3.4}$$

The area that the user has control over in this algorithm is the confidence values. The confidence in the particle's velocity, c_1, is set to a value initially and then kept constant throughout the entire time span. This value is always kept less than 1, which signifies a deceleration (it is not desirable for the particle to continually gain speed, because this would make convergence impossible). The confidence values for the other two terms, however, vary. An input to the algorithm is c_{max}, which is the maximum confidence that can be placed on either piece of information. A random number is chosen between 0 and c_{max} for each iteration.

The equation to decide a particle's next velocity now has only two parameters that are up to the user. The literature found that making these two parameters dependent was beneficial [17]. The equation used is given below in Eq. (3.5), where φ is now the only parameter up to the user.

$$\begin{cases} c_1 = \frac{1}{\varphi - 1 + \sqrt{\varphi^2 - 2\varphi}} \\ c_{max} = \varphi c_1 \end{cases} \tag{3.5}$$

It is important to note that the value of φ cannot be less than 2, because then the parameter c_1 has imaginary components. This work found this method to be less effective than giving the user full control over both confidence values.

3.5.2 Informants

When looking at the entire PSO algorithm, and not just a single particle, it becomes apparent that the communication of the particles is another parameter that affects the performance of the algorithm. If all particles speak to each other on each iteration, then the current best overall solution found will dominate the choices made by all particles, which could potentially lead to premature convergence. Conversely, if not enough particles communicate, then each particle could be left to explore on its own, which eventually turns back into purely random search. The number of informants, *K*, is an important parameter for the user to set.

3.6 Ant Colony Optimization

Although ants can sometimes be pesky little creatures, they definitely excel at finding and exploiting optimal paths between their nests and food sources. Like many optimization methods that seek to model real world behavior, Ant Colony Optimization (ACO) algorithms seek to mimic the behavior of ants. All algorithms developed, equations used, and explanations presented stem from one source: "Ant Colony Optimization" by Marco Dorigo and Thomas Stützle [11]. This section serves as a summary of the techniques presented in that book, as well as the adaptations required for the particular problems tackled here. The interested reader is encouraged to reference that book for more detailed explanations.

Real ants communicate with each other indirectly, via *stigmergy*, to tell each other how well their choices have paid off. They lay down a chemical, called pheromone, and they lay more or less depending on how happy they are with their path/findings. For any ants that come that way later, their decisions are influenced by the amount of pheromone that has been deposited by previous ants; if there is a lot of pheromone on a certain path, then they will likely follow that path. As the pheromone from bad paths gradually evaporates (since less or no ants follow it), and pheromone builds up on better paths, eventually all the ants follow one path.

In ACO algorithms, artificial ants are generated that follow 'paths' through discrete or NP-hard optimization problems, continually updating the pheromone levels of their paths to influence the choices of later ants. Traditionally, ACO has been applied to 'round-trip' problems, like the Traveling Salesman Problem. In these problems, ants leave from a random node, and travel to every available node in the problem, finally ending up at the node from which they started. Based on the cost of their tour, they alter their pheromone deposit at every node. This is a direct analogy to the behavior of ants in real life: they leave the nest, find food, and return to the nest. This methodology does not map directly over to other NP-hard problems, such as the orbit optimization problems tackled in this work. Instead, this work models these problems as 'one-way' problems, where the same basic idea is applied: ants travel from the first planet to the last, and based on how much their trip costs, they alter the levels of pheromone deposited. The nodes are the bodies involved in the trajectory at different time steps. The initial body's time is

chosen randomly for the first iterations, then at some point the algorithm transitions to a probabilistic selection of the first node.

One of the original ACOs was called Ant System (AS). Although it worked well on 'shorter' problem instances, it has inspired many variations that have significantly improved the obtained results, especially for more complex, 'longer' problems. One part of AS that remains with all its variations used in this work is its "choice info" matrix. This matrix serves as a probability matrix (hence the variable *p*) that combines the amount of pheromone on the next available arcs with the cost to cross those arcs to arrive at the next available nodes. By placing more or less weight on either contributing part, the path the ants eventually end up on changes. This matrix is given by Eq. (3.6) below, where *i* and *j* represent the current node and the next potential node, respectively. The amount of pheromone on a particular arc is denoted by τ_{ij}, and the cost to get from node *i* to node *j* is η_{ij}. These two values are raised to the power of α and β, respectively, to vary the importance of either. The summation term in the denominator is the sum of all the weights of the possible arcs to use at that step. In this work, that means the sum of all the potential weights of the next set of nodes (the available transfers to the next celestial body).

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{available}[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}} \tag{3.6}$$

The variations/extensions of AS that have been included in this work are Elitist Ant System (EAS), Rank-Based Ant System (AS_{rank}), Min-Max Ant System (MMAS), and Ant Colony System (ACS). The differences between these specific

algorithms are explained in the sections below. A summary table for reference is provided in Table 3.1 [11] that shows the recommended values for the various parameters in these problems. The number of nodes is given by the variable *m*, the number of ants is given by *n*, and the initial pheromone level applied to each node is given by τ_0. The initial level of pheromone and the evaporation rate determine how much exploration can occur in the beginning of the optimization process. If the initial level is too low, then the search may prematurely converge on a suboptimal solution, and if it is too high, then it may take an unnecessarily long time to converge to any solution. The parameters *e* and C^{nn} will be described in the next sections.

Table 3.1. Parameter Settings for ACO Algorithms

ACO Algorithm	α	β	ρ	m	τ_0
AS	1	2 to 5	0.5	n	m/C^{nn}
EAS	1	2 to 5	0.5	n	$(e+m)/\rho C^{nn}$
AS_{rank}	1	2 to 5	0.1	n	$0.5r(r-1)/\rho C^{nn}$
MMAS	1	2 to 5	0.02	n	$1/\rho C^{nn}$
ACS	--	2 to 5	0.1	10	$1/nC^{nn}$

3.6.1 Tour Construction

In this work, a tour construction technique was developed that, to the best knowledge of the author, is unique to spacecraft trajectory optimization problems. In this technique, for every celestial body involved in the trajectory, there is a set of nodes. In an ant's tour, it will visit each *set* of nodes once, unlike traditional ACO problems where the ant visits each *node* once. There is also no return to the original node (as that would be simulating direct travel from the final body back to the launch body).

Before any ant is sent on a tour, the 'nearest neighbor tour' must be completed. The cost of this tour is denoted as C^{nn}, and is needed for each method's initial pheromone deposit calculation. To construct this tour, an ant is placed on a random initial node. While the choice info matrix has not yet been calculated since the pheromone levels are unknown, the cost to get from each node to another (all η_{ij}) is known. Using this information alone, the ant takes the cheapest path for each leg of the tour. This final cost is then used to give all nodes their initial pheromone level, and the artificial ants can begin their actual tours.

The user has input a range of launch dates, followed by a range of possible times of flight between the celestial bodies. So naturally, the possible locations of the body that the spacecraft is launching from serve as the first set of available nodes. Then, each subsequent body has a set of nodes defined by its locations that span the earliest possible time the spacecraft could arrive to the latest. As more legs are added to the trajectory, the window of time for each set of nodes increases, depending on how large the given TOF span is.

3.6.2 Pheromone Update

After the ants have all completed their tour (except in ACS, where the ants complete their tours and update pheromone levels in parallel), the pheromone levels are updated. First, evaporation is applied. The input parameter ρ is used to denote the amount of pheromone that evaporates after each iteration. For example, if ρ is set to 2% (as it is in MMAS), then after each iteration, 2% of the pheromone from every node is removed. Each of the variations on ACO has a value for ρ that gives the best results in most cases, but this value of course may

need to be changed per the problem. These recommended values can be seen in Table 3.1.

The amount of pheromone applied to each node, as well as how many ants are even allowed to deposit pheromone, is where the various algorithms differ the most. Ant System is the simplest. In AS, every ant deposits pheromone. Each ant deposits pheromone only on the arcs it has visited, and the amount of pheromone deposited is inversely proportional to the cost of its tour.

One of the first extensions of AS, the Elitist Ant System, changes the pheromone update procedure slightly. It uses the same evaporation technique and the same deposit procedure for each ant except for one small change. In addition to each ant depositing an amount inversely proportional to its own tour, it adds an amount inversely proportional to the cost of the cheapest tour so far, but only if the arc it traveled on belongs to the best tour. The influence of this 'best so far tour' is assigned by the parameter *e*. Typically, this parameter is set to equal the number of ants used in the problem.

In the next modified version of AS, the Rank-Based Ant System, only *w* ants are allowed to update the pheromone levels along their trails. This parameter is typically set to 6 ants. Then, each ant updates its own trail similar to AS, except the inversely proportional cost is multiplied by *w* minus that ant's tour's rank, so better tours affect the pheromone levels more. Also, if an arc of any of the *w* ants' trails lies on the 'best so far' trail, it receives pheromone equal to *w* multiplied by the inverse value of the 'best so far' cost.

The last variation of AS that still has the ants complete their tours separately before any pheromone update is applied is the Min-Max Ant System. As the name would imply, the pheromone levels for all arcs are bounded by a pre-set minimum and maximum. The only ant that is allowed to deposit pheromone is either the ‘best so far’ ant, or the ant that has the best tour of that iteration. This may lead to premature convergence, but this possibility is minimized by setting the pheromone levels of the arcs to, initially, the maximum bound.

The last extension of AS, called Ant Colony System, is the most different. It actually adds some new parts that AS and the variations previously described do not have. In ACS, the ants move along their tours in parallel. In the previous algorithms, each ant completed a full tour before the pheromone levels were changed. Here, the pheromone level is updated after any single ant completes a tour. After an ant travels from one node to another, it removes some pheromone from that arc in order to promote exploration. Also, when the ant is traveling, it does not only look at the choice info matrix. Instead, there is a chance that an ant can ignore this decision weight and choose its next path purely based on cost (ignoring pheromone completely). After all tours have been completed, the ant which has been on the ‘best so far’ tour adds pheromone to that path, this time with the added pheromone equal to the evaporation rate multiplied by the inverse cost of that arc’s travel.

A difference unique to this work arises in the flyby penalties. In the ACO formulation, each portion of the total cost is only associated with its respective leg, and the only time the flyby penalty can be applied is when a certain pair of legs are

matched. That is to say that the costs of just the legs is not all of the information needed; the penalty could not be added to the ACO cost matrix. It was instead used as an influence on the pheromone deposit left by the ants. It is included in each ant's stored tour cost, but not the cost information for each leg that feeds into the choice info matrix.

4. GENERALIZED ISLAND MODEL

The generalized island model [7] is a method used in optimization that allows multiple algorithms to run, then allows them to share and compare their solutions. They can then use this information to explore new areas of the search space or update their population to find a solution faster. This allows different algorithms to work together and feed off of each other's strengths and overcome each other's weaknesses. Each method constitutes one island, and the layout of these islands is referred to as a topology. Different topologies, or 'archipelagos', can be applied to a problem. Having different sets of islands connected alters the topologies.

When islands share/compare their solutions, this is referred to as migration, and how often this occurs is called the migration policy. The solution(s) an island chooses to share are chosen based on that island's selection policy. Finally, whether or not an island keeps the solution(s) that other islands share with it is called an island's replacement policy.

4.1 Topology

Arguably the most important element of the Island Model is the chosen topology. Different topologies may work better with certain problems, but the exact choice of which topology to use is a difficult one to answer. A topology with more islands is typically assumed to have a better chance at finding the best solution, if islands are set to different algorithms and islands with the same algorithm have different parameters. This is a good option if not much is known about the search space or which algorithms will perform better on the problem at hand. Incorporating many different islands allows the better islands to 'control' the migration, without the user needing to know beforehand which islands will be these better islands.

The down side to this, of course, is an increase in the needed computational power and time. One strategy that can be used is to run each algorithm by itself through the problem first, to see how well the algorithm performs. It is then easier to make an informed decision on which algorithms to finally include in the topology.

4.2 Migration

Migration is the main element in the island model. The island model can almost be thought of as a cousin to evolutionary algorithms, where the migration process is just another operator for obtaining new solutions for the next generation. When choosing a migration policy, there are a few options. The first choice is between synchronous or asynchronous migration. There are pros and cons to each method.

In a synchronous migration policy, all migrations occur at the same time. This means that the migration can only occur at the pace of the slowest of the islands. In this method, all islands must obtain their solutions before any sharing can take place. When all islands have their solutions, the connected islands share/compare the solutions, all based on their selection and replacement policies.

In asynchronous migration, islands do not need to wait for other islands. With asynchronous migration, there are two possible options as well: migration driven by the sharer, or migration driven by the receiver. In the former, as soon as an island finishes, it sends its solutions to all of the islands that it is connected to. These islands then take these shared solutions, and choose whether or not to use them based on their replacement policy. This could potentially render an island useless; if it is significantly slower than another island that shares with it, then it

will never be able to finish, and will be dominated by the faster island(s). The other option, migration driven by receiver, resolves this potential issue, by giving control of the migration to the receiving island. In this case, even the slowest island is allowed to finish before any sharing occurs.

4.3 Selection

An island's selection policy dictates which solution, and how many solutions if multiple are desired, the island will share with its connected islands. The options for an island's selection policy are essentially the same as the selection policies for the GA; they can be random, or natural selection can be used, or some sort of weighted probabilities can be assigned. For a detailed explanation of these possible choices, see Section 3.3.2.

4.4 Replacement

Just because a solution is shared with an island does not mean that the island will accept it. An island's replacement policy decides whether or not the shared solution is kept, or if the island would like to only keep some of the solutions. The replacement policies follow almost the same guidelines as the selection policies: they can be random, only keep the best solutions, or keep only some solutions based on some weighted probabilities or thresholds. A possibility for a replacement policy is one where an island only keeps solutions that are better than solutions it has generated itself.

5. ALGORITHM VERIFICATION

Before any testing was done with the STOpS GUI (spacecraft trajectory problems or the Island Model), each algorithm had to be tested to examine how well it worked independently, as well as which parameter settings should be used as the defaults. The five algorithms used in this work were placed into three groups, each with its own verification process. The first verification (detailed in “Stochastic Verification A”) was applied to three of the four stochastic methods: Genetic Algorithm, Differential Evolution, and Particle Swarm. The second verification (detailed in “Stochastic Verification B”) was applied to the Ant Colony algorithm. The third verification (detailed in “Deterministic Verification”) was applied to the Local Search algorithm.

5.1 Stochastic Verification A

For the first stochastic verification process, two difficult and well-known test functions were chosen: Ackley’s function and Rosenbrock’s function [4]. Both functions were run with 10 dimensions. The search space for Ackley’s function spanned each dimension from -20 to 20, and the search space for Rosenbrock’s function spanned each dimension from -10 to 10. The two-dimensional representation of and d-dimensional equation [4] for Rosenbrock’s function can be seen below in Figure 5.1 and Eq. 5.1, respectively. The two-dimensional representation and d-dimensional equation for Ackley’s function can be seen in Figure 3.1 and Eq. 3.1, respectively. As a measure of success in this verification process, if the algorithm obtained a solution below 1.5 for Ackley’s function or 10 for Rosenbrock’s function, then that run was deemed a success. These values are more conservative than some of the tests found in the literature. They were not set

any lower because the purpose of this verification was to ensure that the algorithms could get into (or at the very least, near) the basin of the global optima. The process of moving along that basin is more the responsibility of the local search optimizer (when utilizing the entire island model), and as the algorithms communicate with each other in the island model, a close answer will still serve as valuable information for the algorithm on the next iteration.

$$f(\boldsymbol{x}) = \sum_{i=1}^{d-1}[100(x_{d+1} - {x_d}^2)^2 + (x_d - 1)^2] \quad (5.1)$$

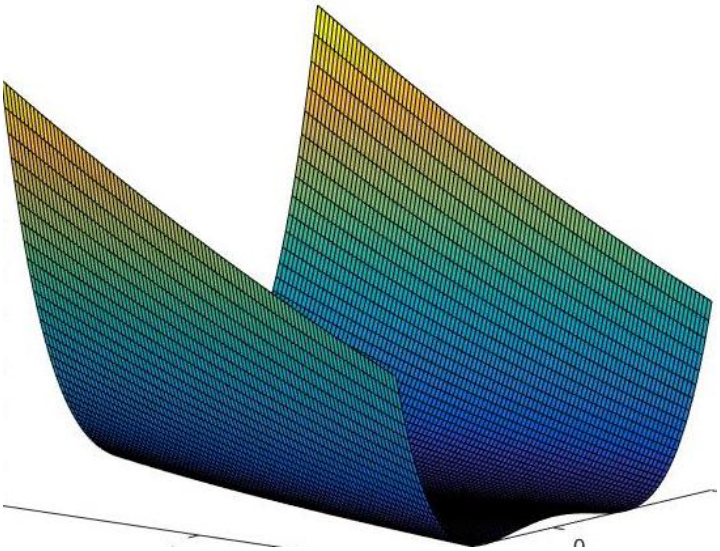


Figure 5.1. Two-Dimensional Representation of Rosenbrock's Function

5.1.1 Genetic Algorithm Verification

The first algorithm tested was the Genetic Algorithm (GA). For the GA, there are 10 total options: five generation advancement options (natural selection, rank weighted random, cost weighted random, thresholding, total random replacement) and two mating options (binary or continuous). These options were run 100 times at 10 values of crossover probability and mutation probability, each from 0-100%. The population was kept at 100 members for each test, ran for 20 generations, and kept 30 members for the next generation (when that parameter applied). On

each run, the best solution found was recorded and plotted to see how the various parameters affected the GA's performance. These surface plots shed some light on multiple aspects of the GA. First, it verified that total random replacement was not nearly as successful as the other options available to the GA. Additionally, the thresholding method was not effective, since it could only guarantee a solution as good as the threshold set by the user, and only if the algorithm found any solutions that met that threshold in the randomized phase. A variable threshold method could potentially fix this issue, but was not implemented in this work.

When examining the weighted random option sets, it can be seen that rank weighted random outperformed the cost weighted random in each case. The cost weighted method placed too much emphasis on the local minima found early on, whereas the rank weighted method allowed enough freedom to escape them.

It can also be seen that the binary GA outperformed the continuous GA in each case. In each option set, the binary GA was able to find a better solution than the continuous GA over a wider range of crossover & mutation probabilities. However, the binary and continuous GAs each had their own set of optimal parameters.

Based on these findings, the only options examined in the next step of the GA verification were natural selection and rank weighted random (each for both binary and continuous GAs). Two of the plots used to come to these conclusions are shown below in Figure 5.2 and Figure 5.3.

The full set of these surface plots can be found in Appendix A. It should also be noted that when executing this first verification step, the binary GA took significantly longer to complete than the continuous GA.

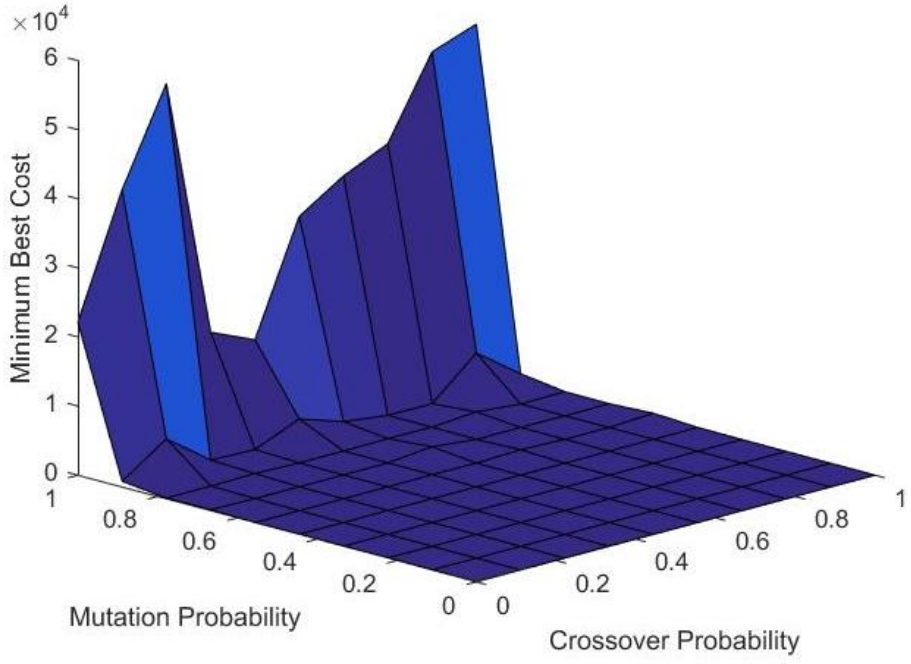


Figure 5.2. Rosenbrock: Rank Weighted Random, Binary

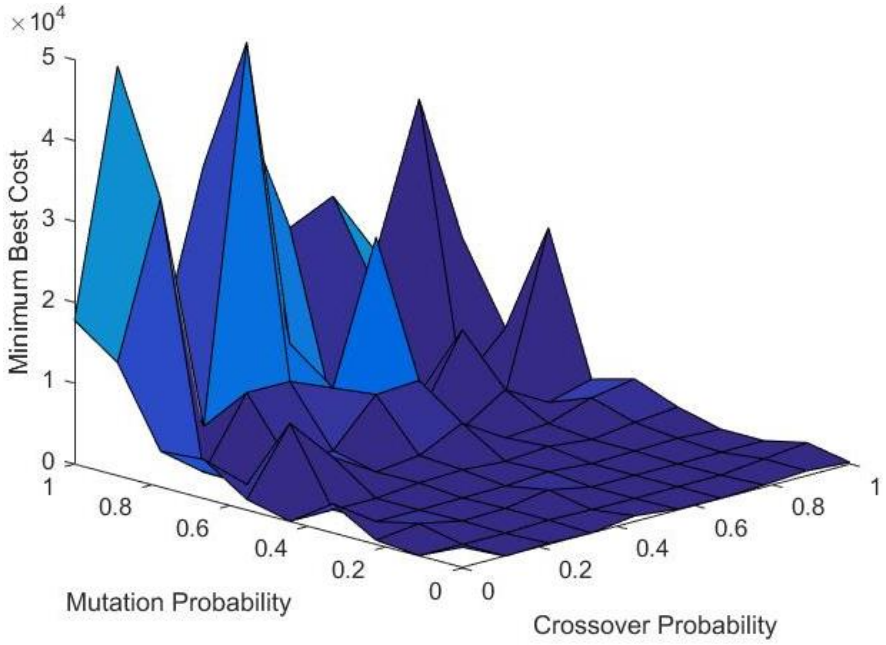


Figure 5.3. Rosenbrock: Cost Weighted Random, Binary

The plots in Figure 5.2 and Figure 5.3 both show the clear trend that a higher crossover probability and lower mutation probability lead to the best results. To get a better feeling for the best crossover and mutation probabilities for each method, the four continuing option sets were run 100 times again, but with the crossover probabilities varying from 60-100% and the mutation probability varying from 0-40%, each with 10 points again. This time, in addition to recording the best solution found, the number of successes was also recorded. Two of these plots can be seen below in Figure 5.4 and Figure 5.5, and a table showing the best success rate and the associated probabilities for each option set can be seen below in Table 5.1. In the case of continuous rank weighted random (where both functions never saw success), the best solution found for Ackley's function was 2.833, and the best solution found for Rosenbrock's function was 27.65.

These results show that using natural selection is clearly the best generation advancement choice for the GA, with binary outperforming continuous. As a result, the default settings used in this work are binary natural selection, with crossover and mutation probabilities of 70% and 10%, respectively.

Table 5.1. Success Rates of GA Option Sets (N_{pop} = 100, N_{gen} = 20, N_{keep} = 30)

		Ackley's Fxn			Rosenbrock's Fxn		
		p_{cross}	p_{mut}	Success Rate	p_{cross}	p_{mut}	Success Rate
Natural Selection	Binary	66%	3%	37%	66%	9%	43%
	Continuous	80%	0%	35%	91%	14%	7%
Rank Weighted Random	Binary	69%	3%	3%	71%	6%	5%
	Continuous	83%	11%	0%	97%	9%	0%

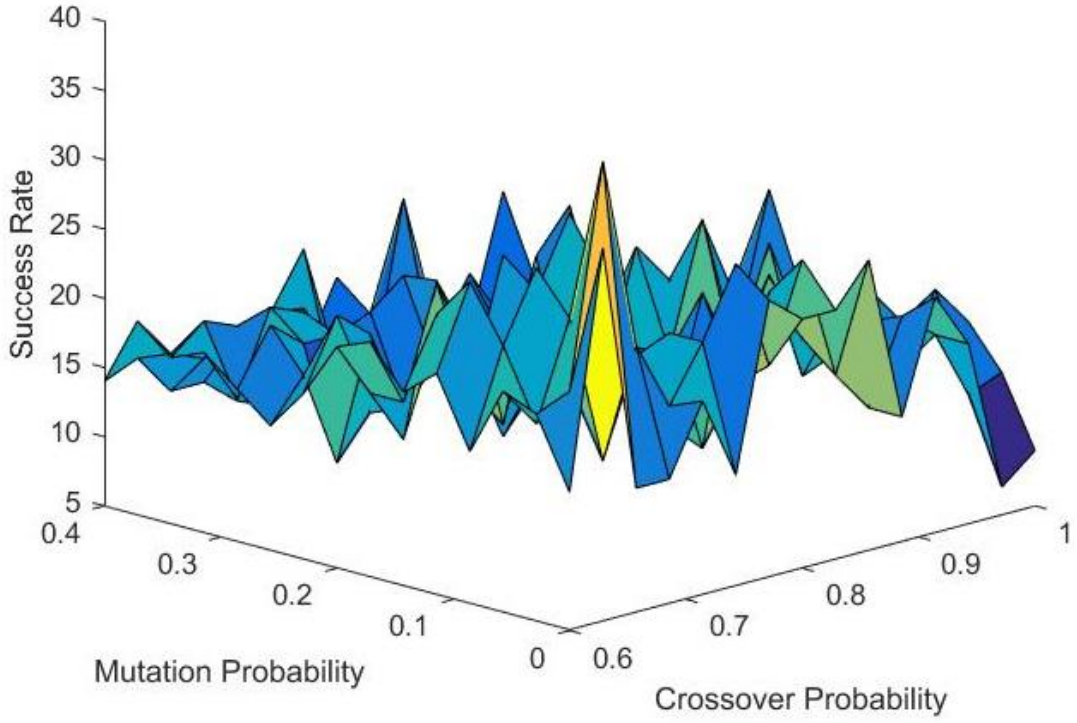


Figure 5.4. Ackley: Natural Selection, Binary

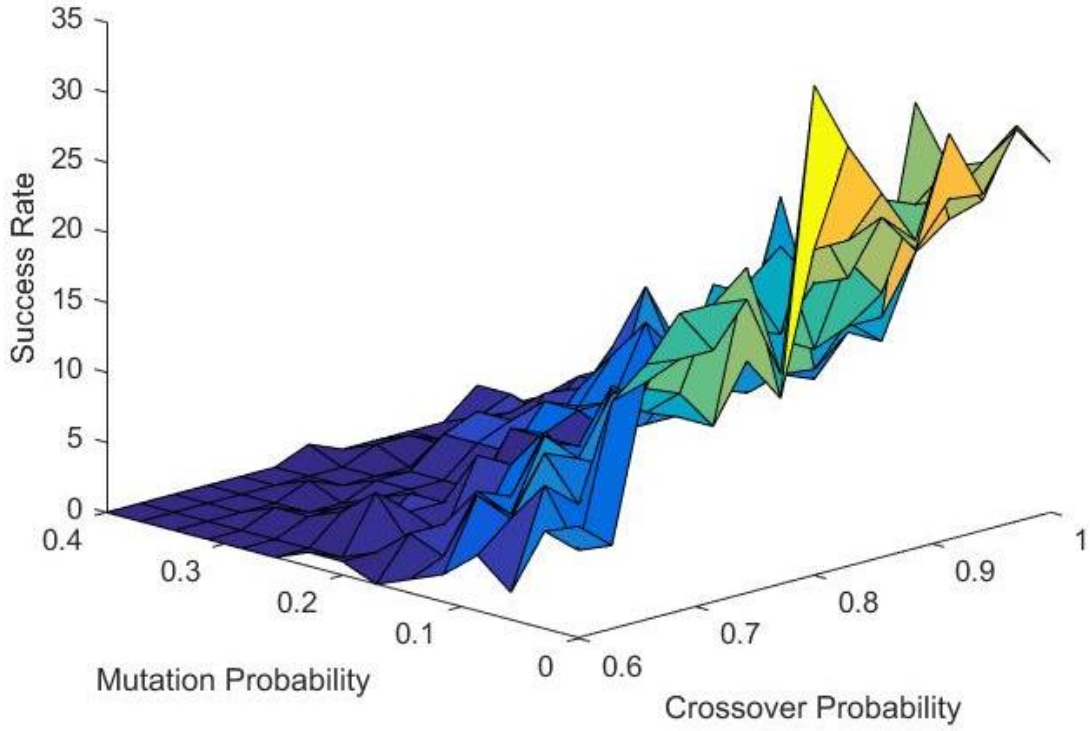


Figure 5.5. Ackley: Natural Selection, Continuous

5.1.2 Differential Evolution Verification

The second algorithm tested was the Differential Evolution algorithm (DE). For DE, there are 18 total options: three options for selection of the base vector (random, best so far, random/best so far blend), three options for the scaling factor (constant, jitter, dither), and two options for survivor selection (natural selection, tournament). As mentioned in Section 3.6, two other survivor selection methods were originally explored (rank weighted random & cost weighted random), but they were found to be incredibly ineffective (see Appendix B). First, these options were run 100 times at 10 values of crossover probability and scaling factor, changing from 0-100% and 0.2-1.2, respectively. The jitter and dither methods were not yet examined. The population was kept at 100 members for each test and ran for 20 generations. On each run, the best solution found was recorded and plotted to see how the various parameters affected the DE's performance. These surface plots shed some light on multiple aspects of the DE algorithm. It can be seen that the scaling factor should be kept low. This confirms the findings in the literature that the scaling factor should not be greater than 1, which would accelerate the solution particles [12] instead of allowing them to converge. However, the literature recommended a value of around 0.7, where these tests show that a value closer to 0.4 is more effective. The ideal crossover probability was around 40-80% for each case. Two examples can be seen in Figure 5.6 and Figure 5.7 below. The full set of these surface plots can be found in Appendix B.

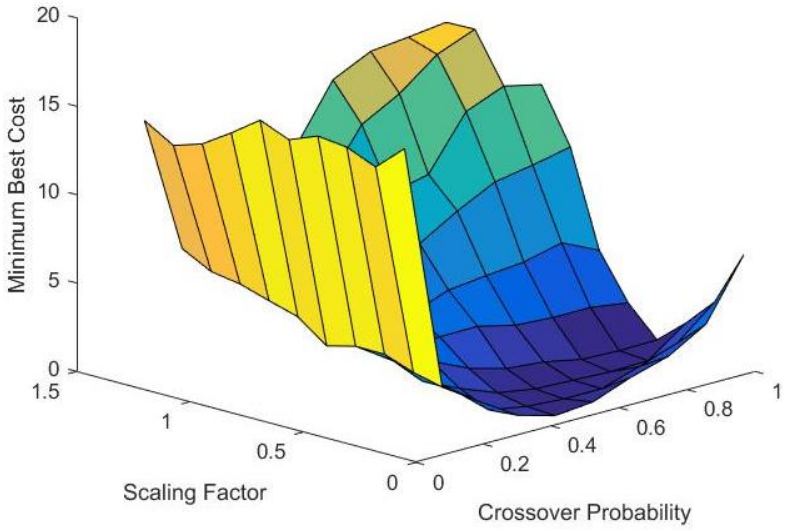


Figure 5.6. Ackley: Blended Base, Constant F, Natural Selection

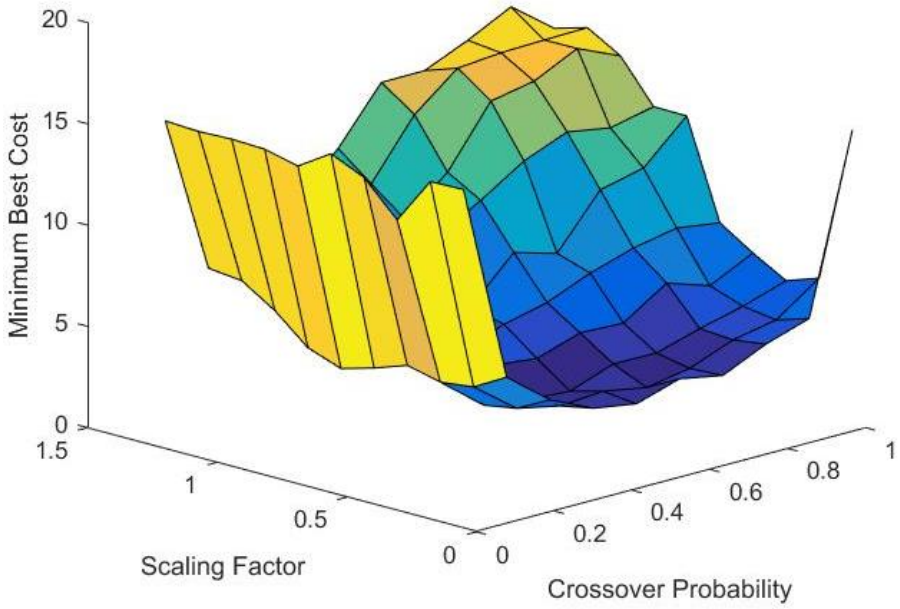


Figure 5.7. Ackley: Random Base, Constant F, Natural Selection

The plots show that when choosing the base vector, selecting random base vectors is ineffective. These cases saw no success, whereas both the best so far and blended selection processes saw success for multiple parameter values. For the next step in the verification process, the random base vector selection method was not included.

The tournament survivor selection method performed similarly to natural selection. This is because the number of competing tournament members was kept low, which allowed the method to behave similar to natural selection. As the number of tournament members increases, the algorithm approaches the behavior of the weighted random selection process, which was found to be ineffective. In particular, the tournament method was found to perform best when 10% of the total population was used for the number of competitors, but this still saw a slight decrease in performance from natural selection in all tests run. For this reason, the tournament selection method was excluded from the rest of the verification. The table from this step can be found in Appendix B.

The next verification step was to run the algorithm again with only 30 members per generation to see if the population size affected the best values to use. It was found that the population size did not change the optimal scaling factor or optimal crossover probability. It did, however, decrease the success rate, since fewer members corresponds to a smaller range of initial exploration.

One more verification step was required before the final verification. All base vector selection methods (paired with natural selection for the survival method) were run with both jitter and dither. Jitter saw no success in either function for either base vector selection method, and therefore it was removed from the rest of the verification. Dither saw moderate success, but only with the best so far base vector selection method. It is possible that since dither utilizes some randomness, when combined with the randomness involved in the blended base vector selection

method, the algorithm does not receive enough guidance. Dither was still included in the final analysis. These results are shown in Table 5.2 below.

Table 5.2. Success Rates of Jitter & Dither (N_{pop} = 100, N_{gen} = 20)

		Ackley's Fxn		Rosenbrock's Fxn	
		p_{cross}	Success Rate	p_{cross}	Success Rate
Best So Far	Jitter	~	0%	~	0%
	Dither	83%	31%	96%	27%
Blend	Jitter	~	0%	~	0%
	Dither	100%	2%	100%	1%

The last step of tests used the larger population size of 100 and the constant scaling factor technique, with surface plots generated for values from 0.1 to 0.7 with 20 points. The remaining options were run 100 times with the crossover probability ranging from 20-100%, with 20 points. The surface plots for success rates can be seen in Figure 5.8 and Figure 5.9 below, and a table showing the best success rate and the associated parameters for each option set can be seen in Table 5.3. The surface plots for best solutions found in this step can be found in Appendix B.

Table 5.3. Success Rates of Final DE Option Sets (N_{pop} = 100, N_{gen} = 20)

		Ackley's Fxn			Rosenbrock's Fxn		
		p_{cross}	F	Success Rate	p_{cross}	F	Success Rate
Best So Far	Constant F	71%	0.45	91%	83%	0.48	53%
	Dither	83%	~	31%	96%	~	27%
Blend	Constant F	83%	0.42	96%	87%	0.42	56%
	Dither	100%	~	2%	100%	~	1%

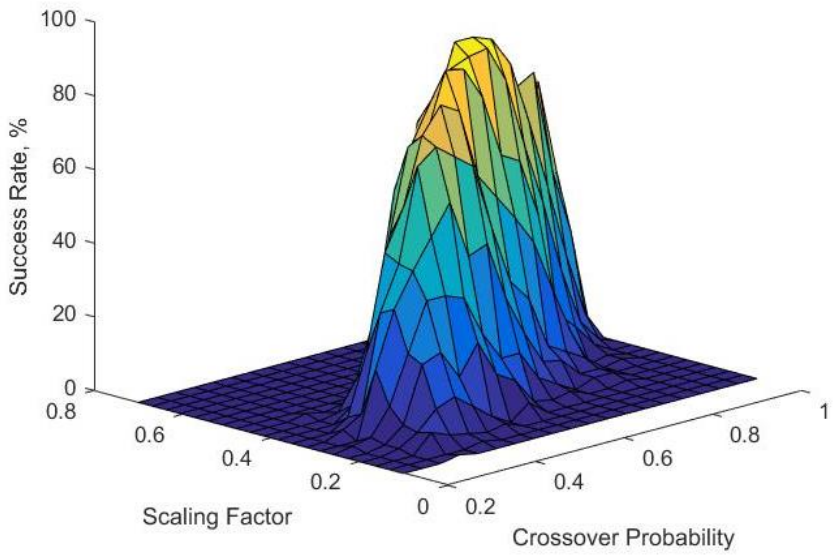


Figure 5.8. Ackley: Best So Far Base, Constant F, Natural Selection

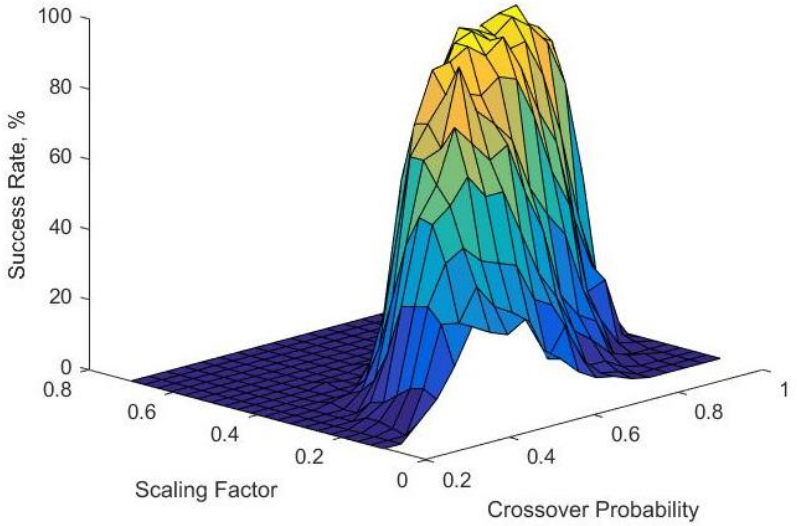


Figure 5.9. Ackley: Blended Base, Constant F, Natural Selection

These results show that dither was never more effective than the constant scaling factor. Both base vector selection methods saw the most success around the same scaling factor. Both base vector selection methods, when paired with a constant scaling factor, performed well. The blended method saw slightly higher success rates (5% more for Ackley's function and 3% more for Rosenbrock's). It can also be seen that the crossover probability fluctuates a bit, but typically performs well on the interval between 70% and 90%.

Consequently, the default settings used in this work are the blend for base vector selection, a constant scaling factor, and natural selection for survivor selection, with the scaling factor set to 0.4 and the crossover probability set to 80%.

5.1.3 Particle Swarm Verification

The third algorithm tested was the Particle Swarm Optimization algorithm (PSO). For PSO, there is only one set of options. This verification process found the best values of the two parameters that the user has control over: the number of informants for each particle (K), and the relation between the particle's confidence in its own velocity and its confidence in it informants' velocities (φ). The algorithm was run 100 times at 10 values of both K and φ, changing from 1-10 and 2.1-3, respectively. The population was kept at 50 members for each test and allowed to run for 200 time steps. On each run, the best solution found was recorded and plotted to see how the various parameters affected the PSO's performance. These surface plots shed some light on multiple aspects of the PSO algorithm. It can be seen that for both functions, the best tested number of informants was 4, and the best tested confidence relation value was 2.1 (the lower limit, which corresponded to a c_1 value of 0.6417 and a c_{max} value of 1.348). The success surface plots can be seen below in Figure 5.10 and Figure 5.11. The best solutions found can be seen in Appendix C.

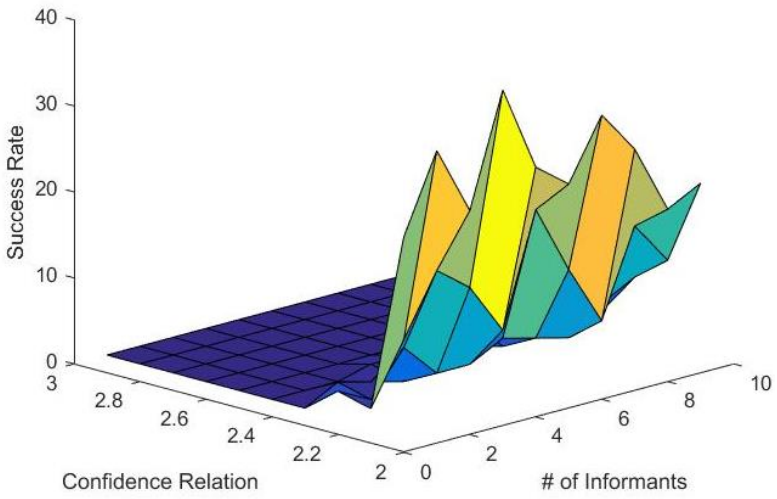


Figure 5.10. Ackley Success Rate

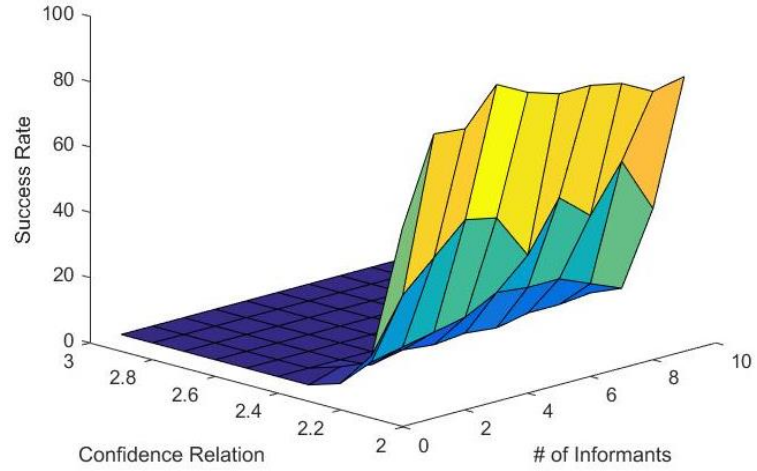


Figure 5.11. Rosenbrock Success Rate

This test left much to be desired. It showed a favorable trend in the choice for number of informants, so the confidence values were made independent again, and they were varied. The confidence in the particle's velocity was varied from .1 to 1.1, and the maximum confidence in the other pieces of information was varied from .2 to 2.2. Eleven points were used for each. This process was run for 4 informants, 8 informants, and 12 informants. The success plots can be seen below in Figure 5.12 and Figure 5.13, with the best solutions found available in Appendix C. The results are shown in Table 5.4.

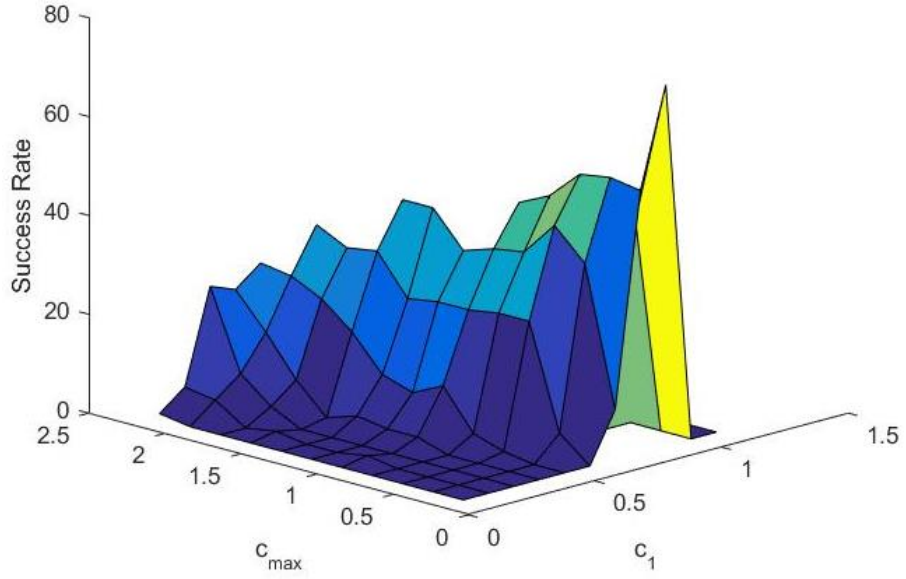


Figure 5.12. Ackley Success Rate

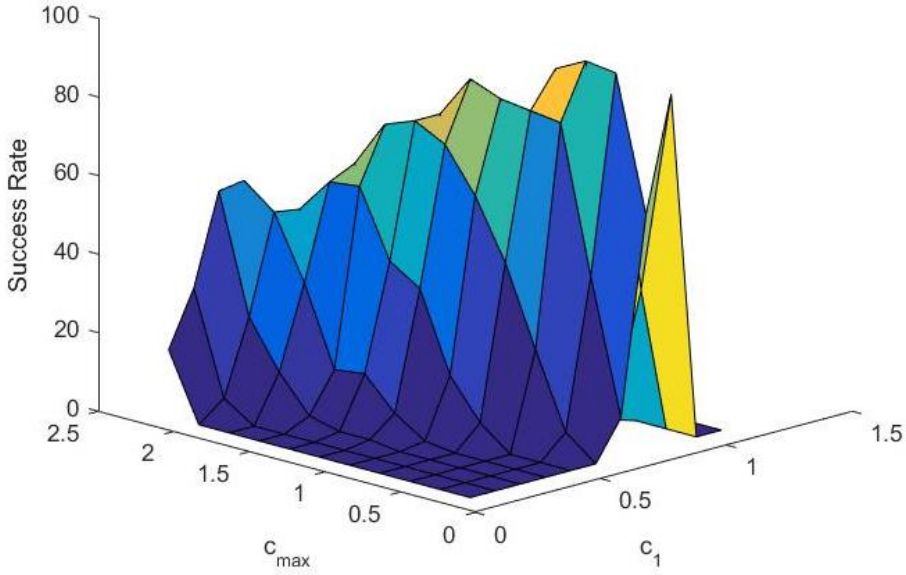


Figure 5.13. Rosenbrock Success Rate

Table 5.4. Success Rates of Final PSO Option Sets (N_{pop} = 50, t_{span} = 200)

	Ackley's Fxn			Rosenbrock's Fxn		
	c_1	c_{max}	Success Rate	c_1	c_{max}	Success Rate
K = 4	0.9	0.2	73%	0.8	0.6	95%
K = 8	0.9	0.2	69%	0.8	0.4	94%
K = 12	0.9	0.2	56%	0.8	0.6	91%

These results show that only a small number of informants are needed. It is interesting to note that the algorithm performed better when c_1 was greater than c_{max}. This contradicts the literature's claim that the two confidence values can be calculated via the relation in Eq 3.7. It seems better to have c_1 be twice the value of c_{max} (instead of vice versa). This difference may arise from a slight difference in implementation of the algorithm; although in the literature the framework for the algorithm was presented, new code was developed for this work since source code in MATLAB was not provided. Despite this difference, the findings from this verification step were used due to the success they saw.

As a result, the default values used in this work were 50 population members with 4 informants, and confidence values of 0.85 for c_1 and 0.4 for c_{max}.

5.2 Stochastic Verification B

The next verification step tested the Ant Colony Optimization algorithm. To start, the general procedure was tested on the traditional ACO test problem: the Traveling Salesman Problem (TSP). This ensured that the algorithm did in fact work before it was transferred over to the new formulation. This also served as a baseline for which extensions of the traditional Ant System were expected to have the highest success rates.

The TSP used was a randomly generated problem with 36 nodes. This problem with its optimal solution is shown below in Figure 5.14. The optimal tour length is 55.42. Keeping with the more conservative definition of success, in this verification step if the algorithm arrived at a tour length equal to or less than 57, it was deemed a success.

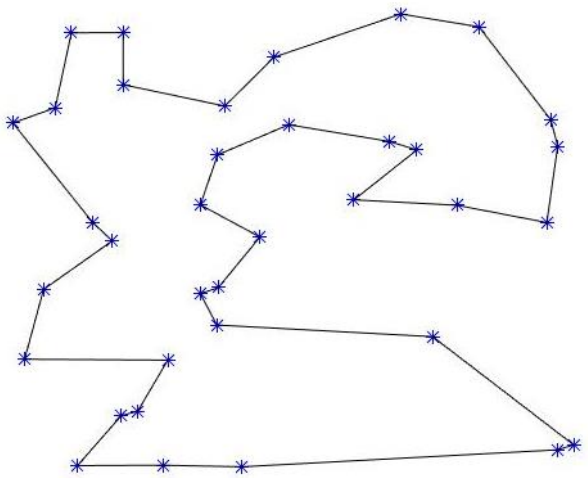


Figure 5.14. Randomly Generated TSP

The original Ant System, its three extensions, and Ant Colony System, were all run while varying the α and β parameters from 0.5 to 2 and 1 to 7, respectively. Each parameter was tested at 20 intermediate values, and each intermediate value was tested 25 times. Each run was allowed to run for 100 ant tours. Two success rate plots are shown below in Figure 5.15 and Figure 5.16, and the results are tabulated in Table 5.5. The full set of the surface plots can be found in Appendix D.

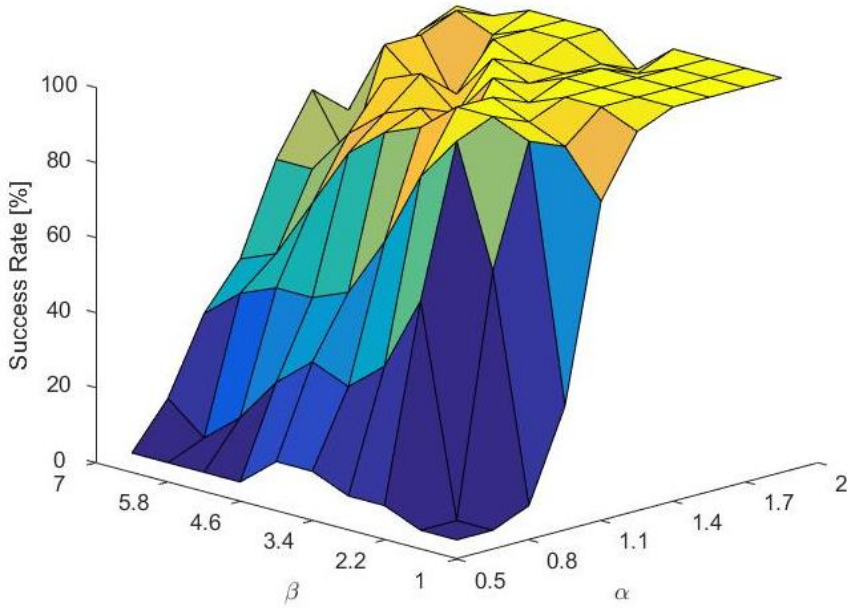


Figure 5.15. Rank-Based Ant System Success

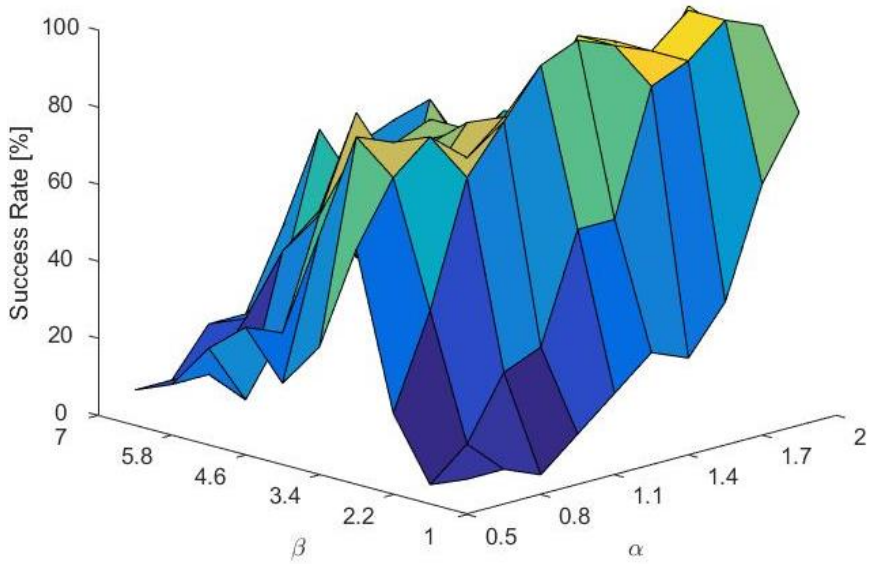


Figure 5.16. Ant Colony System Success

Table 5.5. Success Rates of ACO TSP Option Sets (N_{ants} = 15, N_{tours} = 50)

	α	β	Success Rate	Best Soln
Ant System (AS)	1.17	2.33	72%	55.58
Elitist AS	1.50	1.67	100%	55.42
Rank-Based AS	1.33	2.33	100%	55.42
Min-Max AS	2.00	6.33	64%	55.70
Ant Colony System	1.33	2.33	100%	55.42

The two extensions that performed the best were Rank-Based AS and Ant Colony System. Elitist AS saw a 100% success rate with one setting, but it was not successful once α and β were changed. Both Rank-Based AS and Ant Colony System saw low average costs, but Rank-Based slightly outperformed Ant Colony system. As a result, the default extension used in this work was the Rank-Based AS, and its associated parameters in Table 5.5 were used as the default settings for it.

There were no known test functions to verify the unique ACO formulation in this work. Instead, close attention was paid in the test cases of Section 6 to the differences between the ACO answers obtained and the answers obtained with the other algorithms. That process is explained in more detail in that section.

5.3 Deterministic Verification

Verification of the deterministic method was a simple process. Since, by definition, deterministic methods are expected to arrive at the same solution every time for a given input, the algorithm was given an input in the optimal basin, five times for each function. It was found to arrive at the bottom of that basin with the same number of function evaluations each time. Although other basins are not well understood for those functions in 10 dimensions, five random points were chosen for each function and the algorithm was allowed to run another five times (for each point). The algorithm again found the same answer every time with the same number of function evaluations. For each test, a tolerance of 0.0001 was set (the algorithm was commanded to go until the improvement was less than 0.0001). The results of this test can be seen in Table 5.6 below.

Table 5.6. Deterministic Verification Results

	Ackley's Fxn		Rosenbrock's Fxn	
	J_{final}	Fxn Evals	J_{final}	Fxn Evals
Optimal Basin	0.000037	307	0.031497	211
[8,8,….,8]	15.95934	112	47.86959	1037
[1,2,3,….,10]	14.21791	49	43.31912	1126
[10,9,8,….,1]	14.21192	140	8.21095	1272
[-1,-1,….,-1]	3.57449	211	8.56036	416
[-5,-5,….,-5]	12.63227	189	8.56583	466

6. TEST CASES

With the algorithms verified independently, two test cases were run using the STOpS trajectory optimization interface. Both test cases are based on actual missions that have utilized multiple gravity assists. The first test case (Mariner 10) looked at the differences between the different individual algorithms as well as a few different topologies. The second test case (Voyager 2) examined the effects of the weights applied to the parameters being optimized. Each test case is discussed in the sections below. The default parameter values discussed in Section 5 were used throughout this analysis.

6.1 Mariner 10 Mission

The Mariner 10 mission was the first mission to ever successfully utilize a flyby maneuver to alter its trajectory [15]. The mission launched in November of 1973 and performed a flyby of Venus in February of 1974 to reach Mercury in March of 1974 [16]. This mission is interesting not only because it is the first ever gravity assist maneuver, but because it had unique objectives from a trajectory design standpoint. The Venus flyby was necessary because the launch capabilities at that time could not place a craft on a direct path to Mercury. However, even having the craft arrive in an orbit around Mercury after utilizing a Venus flyby would require a large ΔV. So, the mission designers decided to place Mariner into an obit that was achievable at the time: a heliocentric orbit that had a period equal to twice that of Mercury's, so that it could observe Mercury every 176 days.

When this test case was run, there were three components to the objective function, all of which were being minimized: the required V_∞ when launching the craft from Earth, a penalty ΔV induced if the Venus flyby did not match exactly to

a natural flyby, and the ΔV required to alter the craft's velocity at Mercury to place it in an orbit that hits the same heliocentric position 176 days later. The dates from the actual mission, when run through this cost function, returned a value of 4.5372 km/s for the total cost.

This first test case was considered to be relatively easy since it had only three variables and therefore its primary purpose was to establish a good default number for how many solutions each algorithm should accept via its replacement policy. Each algorithm was run as one island and allowed one migration (meaning it could communicate with itself one time). Three quantities of accepted solutions were examined, each running 25 times for each algorithm. The solution quality for each scenario was used to establish the defaults. The results from the first step of this test are shown below in Table 6.1.

Table 6.1. Mariner 10 Test Case Results: Number of Accepted Replacements

	N_{rep}	GA	DE	ACO	PSO
Average Final Cost	**2**	6.7952	4.6135	9.6617	6.2906
	5	8.0964	4.6069	9.8738	5.1060
	10	8.7376	5.8848	9.8038	5.4575

This first step tested three values for the number of replacements that each algorithm would accept after migration: 2, 5, and 10. The GA, DE, and PSO algorithms saw the best performance when only 5 solutions were accepted after the migration, whereas the ACO algorithm saw better performance with 10 solutions shared. This shows that if too many solutions are shared (except in ACO's case), then the algorithms are influenced too heavily by the previously discovered optima. This leads to premature convergence. Sharing only a few

solutions allows the algorithms to still adequately explore the search space with only slight guidance towards the better solution(s).

To get a better idea of how effective the different algorithms (global search optimizers) were when connected to a local search island (local search optimizer), the individual islands were tested with that topology. For GA, DE, and PSO, five solutions were accepted, and for ACO ten solutions were accepted. The results are shown in Table 6.2. It is important to note the in this table and the tables following it, when function evaluations are reported, they are only reported to three significant figures of the average, since the purpose of reporting this value was to show the general computational expense; the exact number will change based on the random initialization phase, but was always near the number shown.

Table 6.2. Mariner 10 Test Case Results: Effectiveness of LS Island

	GA	DE	ACO	PSO	LS
Total Cost (solo)	8.0964	4.6069	9.8038	5.1060	34.1679
Fxn Evals (solo)	12000	12400	45000	12000	34000
Total Cost (w/ GB)	7.3585	4.6138	7.8558	5.5976	n/a
Fxn Evals (w/ GB)	13000	13500	47000	13000	n/a

As expected, the GA and ACO algorithms found better solutions when paired with the local search algorithm. When the local search performed by itself, it only saw solutions as good as the local basin it started in. After the first migration, there was no solution improvement, since it had already met its tolerance. However, it appears that the DE and PSO algorithms actually did better on their own than they did with the local search island. This is not exactly the case. These

algorithms consistently found final solutions in the optimal basin (values around 4.2). The reason their averages are higher is from one, two, or three of the 25 evaluations ending in the non-optimal basins (around 9.2 or 12.4) a few times. The differences in the averages stem from one case where the non-optimal basin was located as the final answer. The addition of the LS island should not be misconstrued as detrimental. Instead, it shows that the DE and PSO algorithms, after finding the global basin, can actually navigate to that local optima comparably as well as the LS algorithm.

The next step was to see how much the solution improved when utilizing more than one algorithm (multiple global search optimizers with a local search optimizer). The solution quality here is expected to improve for two reasons. First, since all algorithms find a solution before they compare with each other, whichever algorithm is best suited for this problem will be able to guide the other algorithms towards the best solution found so far, even if the other algorithms may have struggled. Second, since all the algorithms are allowed to run to completion, more function evaluations occur, which means more area of the search space is explored in the randomized phase.

The Mariner test case was run 10 more times with all included algorithms and a local search island. The first topology for this step was all five islands fully connected. Then this setup was run 10 more times with a second topology where each island was only connected to two others: a ring. After these results had been collected, the problem was run another 10 times with a third topology: only the two other algorithms that performed the best in the first step (DE and PSO) connected

to a local search island. All topologies allowed two migrations. To further examine how DE and PSO worked with LS, the results were also recorded when only one migration was allowed. These three topologies are shown in Figure 6.1 and Figure 6.2. The results are shown in Table 6.3.

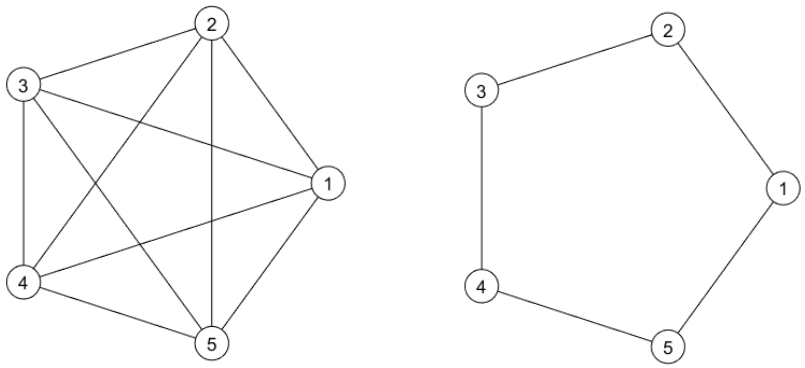


Figure 6.1. Topologies 1 (left) & 2 (right): Islands (in order) are GA, DE, ACO, PSO, & LS

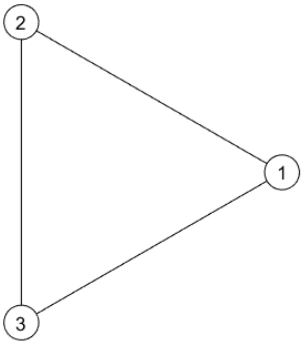


Figure 6.2. Topology 3: Islands (in order) are DE, PSO, & LS

Table 6.3. Mariner 10 Test Case Results: Island Model Results

	FULLY CONNECTED	RING	DE, PSO, LS 2 MIGRATIONS	DE, PSO, LS 1 MIGRATION
Best Cost Found	4.2058	4.2030	4.1979	4.2046
Fxn Evals	120000	120000	38000	26000

When the island model was utilized, the solution quality increased, as expected. Interestingly, when only the two most effective algorithms were used (in

conjunction with a local search island), the suite was actually able to find a better answer than when all the algorithms were used together. This may show a flaw in the fully connected topology. If every island is connected, they all receive the same information as a start for the next generation, if they all only accept the five best solutions found by all islands. This limits exploration, so they are likely to converge to the same local optima. That is why the test was repeated with the ring connection topology. This means that each island is only connected to two of the other four islands. This promoted more exploration, and therefore more diversity amongst the shared solutions. However, the solution quality was still not as good as when only including DE and PSO. This is likely because the optimal connections were not established; it is possible that the solutions would have better if different algorithms communicated. Given enough migrations, the best solutions would make their way around the ring, but two migrations was not enough to see this effect. Every connection combination for the ring topology was not evaluated. Instead, focus was placed on the DE and PSO algorithms.

Although the DE and PSO islands saw slightly worse performance (than when operating by themselves) when paired with only a local search island, when paired with each other and a local search island they performed better. This was the case even when only one migration was allowed. When the DE and PSO algorithms can run together and then send all of their solutions to the local search island, there is a much higher chance of actually being in the global optima's basin for two reasons. First, there is a larger initial randomized search population. Second, the two algorithms give twice as many final answers for the local search

island to choose from as when that island was only connected to one of them. This avoids the potential passing on of detrimental information that was seen when either island was connected only to a local search island. This success is party attributed to the island model, and partly attributed to the higher number of function evaluations. This test showed that the extra migration did add a small improvement; the improvement when compared with the additional function evaluations could be argued to be worth it in some cases, but unnecessary in others.

It is interesting to note that many of the solutions found in this test have a lower cost than the cost obtained when using the dates from the actual mission. In all of these tests, the best solution found had a cost value of 4.1979 km/s, compared to the cost of 4.5372 km/s that is obtained with the true mission inputs. This makes sense because in reality, the mission designers had to plan around feasible launch windows, orbital perturbations, manufacturing inaccuracies, and other factors that were not reflected in this work. Additionally, the actual mission did not exactly match the objectives laid out in this scheme. The objectives placed on the trajectory here are similar to the actual mission, but are more tailored to this work than the actual Mariner trajectory.

6.2 Voyager 2 Mission

After Mariner 10 blazed the trail and opened the doors for mission designers to utilize flybys, a few more missions were conceived. Among these were the two Voyager probes, which took advantage of the unique alignment of the outer planets, which occurs only once every 176 years [15]. Voyage 1 flew past Jupiter and Saturn before heading out of the solar system, whereas Voyager 2 continued on to Uranus and Neptune [17]. Since Voyager 2 had a trajectory with more variables up for optimization, this was the next test case examined. This test case served as an observation of the importance of the weights applied to the parameters being optimized. In order to focus on this aspect, only one topology was used: a DE island, a PSO island, and a LS island all fully connected and allowing for two migrations (the topology from Figure 6.2).

When this test case was run, there were many components to the objective function. Most of the parameters were minimized: required V_∞ when launching the craft from Earth, the flyby periapsis at each planet (Jupiter, Saturn, Uranus), and the flyby penalty at each planet. One parameter was maximized: the heliocentric specific energy when arriving at Neptune.

The flyby periapsis was minimized to allot importance to the scientific discoveries available from high resolution photos as well as a more detailed atmospheric analysis (which are more valuable at lower altitudes). When maximizing the heliocentric velocity, the value was multiplied by -1 before adding it to the total cost.

The flyby periapsis and flyby penalty for Neptune were not included because there was no specific destination after Neptune in this test. Due to the nature of the patched conics used to solve this problem, this makes any periapsis possible then. Instead of looking at the heliocentric specific energy after the flyby, the energy before the flyby was examined, since at this point the ideal flyby to maximize the increase in heliocentric velocity is always possible.

As intended, this case required some fine tuning of the weights applied to each parameter. In the Mariner test case, all objectives had the same units and were on the same order of magnitude. However, in this case, there are three different units (km/s, km, and km^2/s^2) and three different orders of magnitude (10^1, 10^6, and 10^{10}). The algorithms in this work treat the final cost value as a single unitless parameter; it is purely a quality value for the particular inputs the algorithm currently has. The weights applied to these objectives were intended to bring all values to the same order of magnitude. As a baseline, 25 runs were executed where none of the objectives were normalized; all of their weights were left at 1. This case was heavily biased towards the solution with the highest possible heliocentric velocity at the end, since that parameter has the largest absolute value by four orders of magnitude. This meant the shortest possible time of flight between the planets was optimal. This, obviously, is unrealistic. The minimum bound for the time of flight of each leg was set to 100 in this case (arbitrarily), but as a result this was what the optimal solution was found to be. This trajectory is shown in Figure 6.4. This figure, and all trajectory figures contained in this section, follow the legend displayed in Figure 6.3.

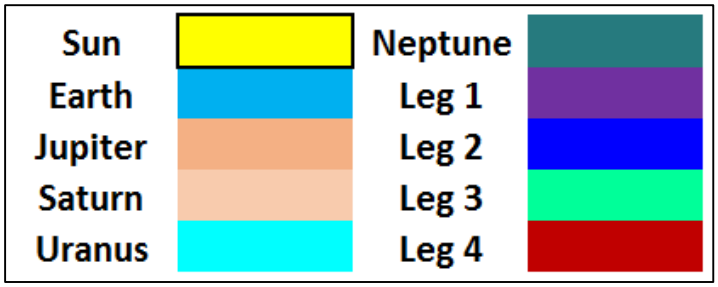


Figure 6.3. Voyager 2 Test Case Trajectory Legend

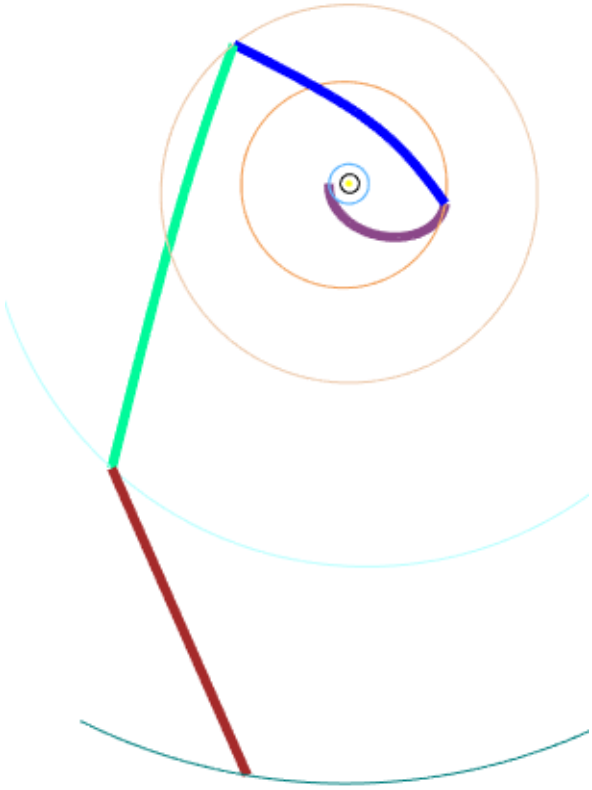


Figure 6.4. Voyager 2 Test Case: No Normalization

The next test normalized the parameters to the same order of magnitude and similar absolute values. The dates for the actual mission were run through the cost function to see what values should be used to normalize. The cost function returned a value of 18.438 km/s for the associated ΔV's, a value of 3144399 km for the sum of all flyby periapses, and a value of 33977129766 km^2/s^2 for the final heliocentric specific energy.

In running this next test, the ΔV parameters had their weight slightly adjusted up to 2 (putting their adjusted value around 37). The flyby periapsis values were assigned a weight of 10^{-5} (putting their adjusted value around 31), and the heliocentric velocity was assigned a weight of 10^{-9} (putting its adjusted value around 34). This brought all the values down to the same order of magnitude and similar absolute values.

Twenty-five runs were executed with these weights. The results are shown in Table 6.4 (in the "Normalized to ΔV" column). This resulted in a mission that was better than the actual mission for each parameter. The ΔV was brought down, the total periapsis altitudes were brought down, and the heliocentric energy at Neptune was increased. Interestingly enough, the time from launch to Neptune rendezvous was also decreased. The resulting trajectory for this case can be seen in Figure 6.5. Again, like the Mariner test case, the objectives laid out in this example do not match the exact objectives of the Voyager mission.

Table 6.4. Voyager 2 Test Case Results

	Actual Mission	No Normalization	Normalized to ΔV	Adding in TOF	Only ΔV
ΔV [km/s]	18.438	301828	14.928	15.426	9.479
Periapsis Heights [km]	3144399	48	374595	323556	2346912
Heliocentric Energy [km^2/s^2]	3.40E+10	1.04E+12	5.42E+10	5.46E+10	2.00E+10

The table also has two more columns for other objective weight cases. The "Adding in TOF" column used the same weights as the "Normalized to ΔV" column, but also included the time of flight for each leg of the trajectory in the final cost. Since the actual mission took 4338 days, the weight applied to this parameter was

10^{-3}. This places the value one order of magnitude smaller than the other values, because it was intended to be added only as a secondary objective. It found a solution with a slightly higher energy orbit, and a lower total periapses value, but it requires more ΔV. It still requires less than the actual mission. Although these two answers are similar, they would still require a trade study from the mission designers; the TOF objective had a noticeable effect on the overall mission.

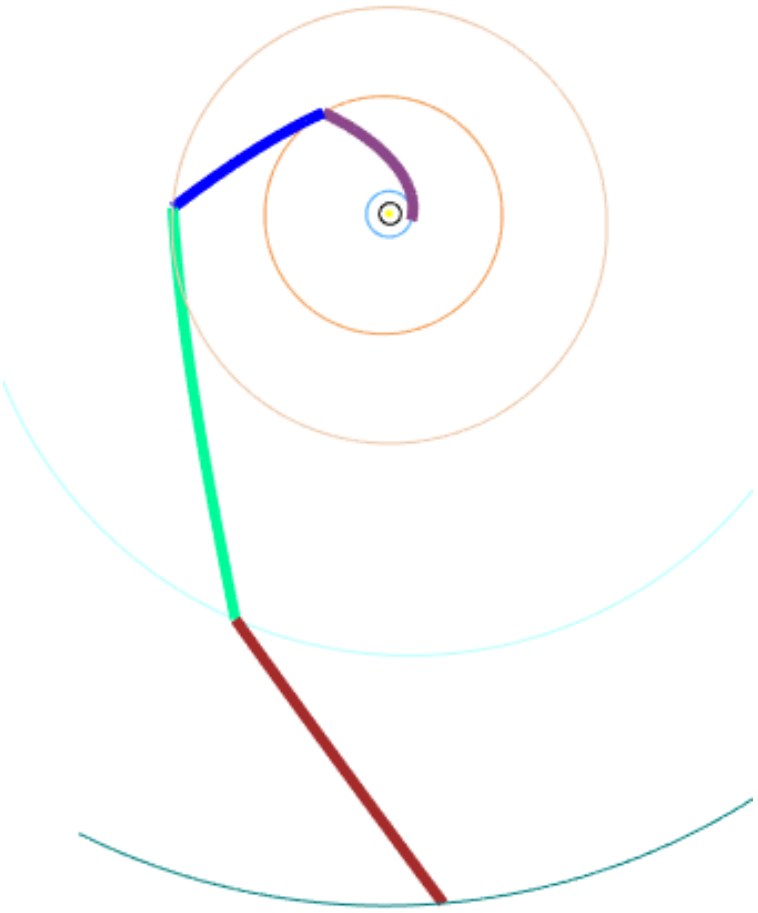


Figure 6.5. Voyager 2 Test Case: Normalized to ΔV

For a last comparison, 25 runs were executed with only the ΔV parameters included (flyby periapsis, specific energy, and time of flight were given weights of 0). This trajectory saw a drastic decrease in the overall ΔV, but at the cost of raising the periapsis altitudes significantly, as well as reducing the specific energy at

Neptune arrival by more than half. Additionally, the total time for this scenario was five years longer.

To show the effects of the optimization weights in a different light, the time of flights for each leg with each scenario are shown in Table 6.5, next to the actual time of flights for Voyager 2's true trajectory.

Table 6.5. Voyager 2 Test Case Time of Flights [days]

	Actual Mission	No Normalization	Normalized to ΔV	Adding in TOF	Only ΔV
Earth to Jupiter	688	100.0	1150.1	1124.9	836.1
Jupiter to Saturn	779	100.1	773.2	750.2	1187.6
Saturn to Uranus	1612	100.0	1218.7	1230.5	2141.7
Uranus to Neptune	1309	100.0	989.9	988.4	1817.3
TOTAL TIME:	4388.0	400.1	4131.9	4094.0	5982.7

This test case showed the importance of the weights applied to the objectives of the optimization scheme. Only the weights were changed across each scenario, yet each time different results were obtained.

7. GUI ENVIRONMENT

The user interface for this work is explained briefly in this section. The initial set-up is shown in Figure 7.1 below; this is the interface that appears when the suite is first run. The left-most and right-most panels change, but the rest remains constant. The top menu has buttons that allow the user to load preset trajectories or optimization settings (“Load”), as well as an option to save custom trajectories and optimization settings (“Save”). The “OPTIMIZE” button in the bottom left will optimize the trajectory based on the user inputs, showing current progress in the middle panel. The “Ephemerides Generation” button group next to that button allows the user to choose between using JPL Horizons ephemeris data [20] or using MATLAB’s built in ode45 solver to generate the information using simple 2-body orbital equations of motion and planetary ephemeris equations from Vallado [19]. The middle panel shows the current progress for each algorithm/island and shows how many migrations have occurred. Later, it allows the user to choose which analysis plots are shown.

The left hand side of the GUI is where the user deals with all available inputs. There are six available panels that are chosen via a dropdown menu. The right hand side is where results are displayed. There are four available panels there, again available via a dropdown menu. The dropdown menu panel choices for each side are shown below in Figure 7.2.

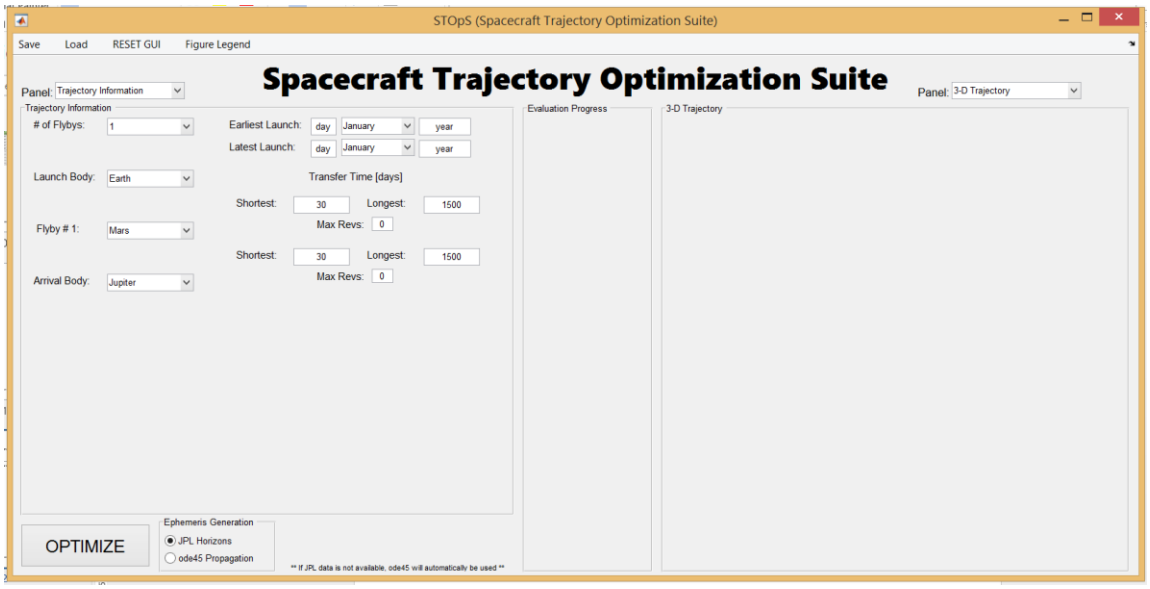


Figure 7.1. Initial STOpS GUI Window

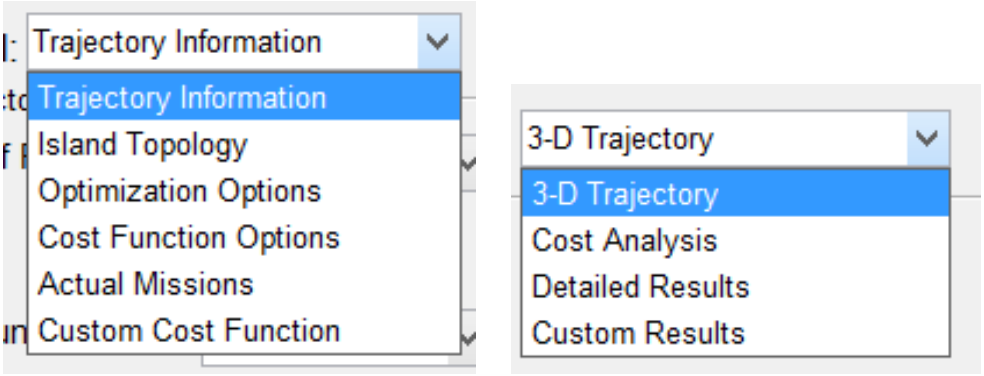


Figure 7.2. Options for Left Panels (left) and Right Panels (right)

The Trajectory Information panel is the left-most panel shown in Figure 7.1. In this panel, the user chooses how many flybys occur in the sequence. They choose which bodies are involved in the trajectory, and in which order. They also set the bounds for all variables by choosing an earliest and latest launch date, then they set the minimum and maximum time of flights for each leg of the trajectory. They also choose how many revolutions are allowed per leg. This information can be saved and loaded using the “Save” >> “Save Trajectory Information” and “Load” >> “Load Trajectory Information” menu buttons.

The Island Topology panel is where the user sets how many islands will be used, how many migrations will occur, and what topology to use. There is a dropdown menu of some preset topologies, or next to “Panel Choice” the user can alter the connection matrix to decide exactly which algorithms are connected. The user also sets which algorithm each island will use here. For each island, the replacement policy and selection policy can be specified, as well as the associated additional parameters that are needed for each policy. This panel is shown in Figure 7.3.

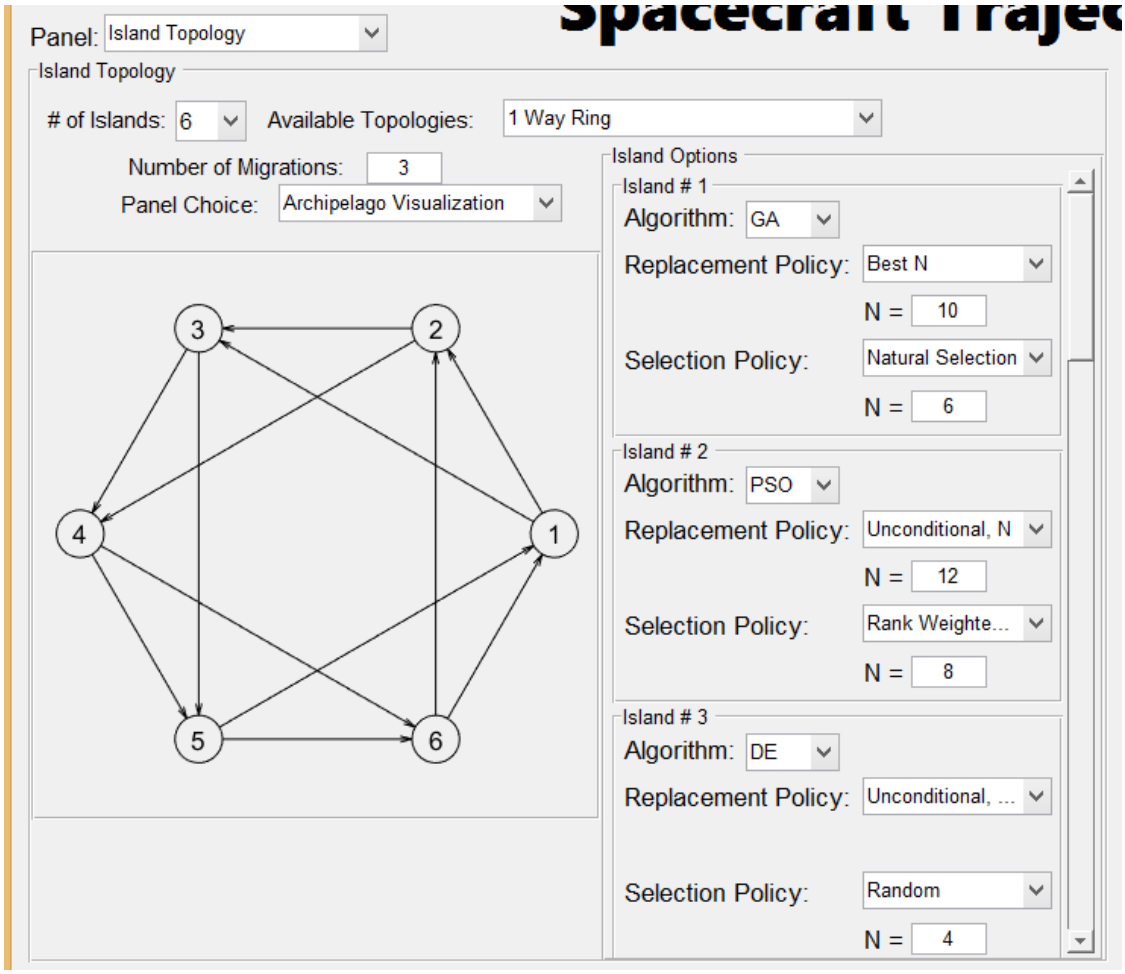


Figure 7.3. Island Topology Panel

The Optimization Options panel is where the user has the option to fine tune all the parameters available based on which algorithms are chosen. The GUI uses the default values mentioned in Section 5 of this work, but if the user desires, they can be changed here. Each of the five algorithms has its own set of choices, and certain parameters are only visible when they apply to particular extensions of that algorithms that have been selected. This panel is shown in Figure 7.4.

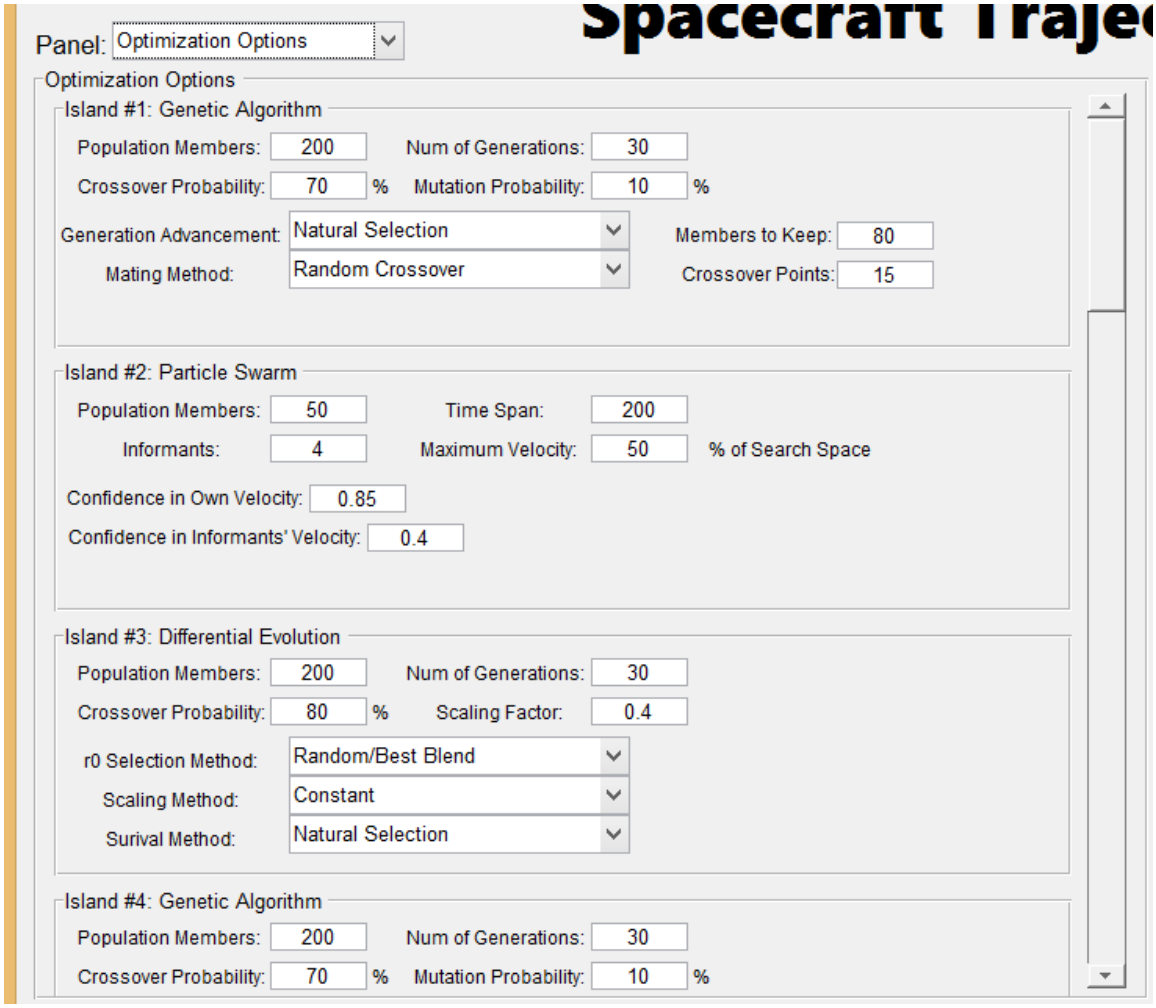


Figure 7.4. Optimization Options Panel

The Cost Function Options panel is where the user chooses the values to be optimized over the trajectory, as well as the weight to assign to each value. Certain parameters can be applied to the entire trajectory or to only particular legs or flybys. Some of the options are shown in the panel below in Figure 7.5.

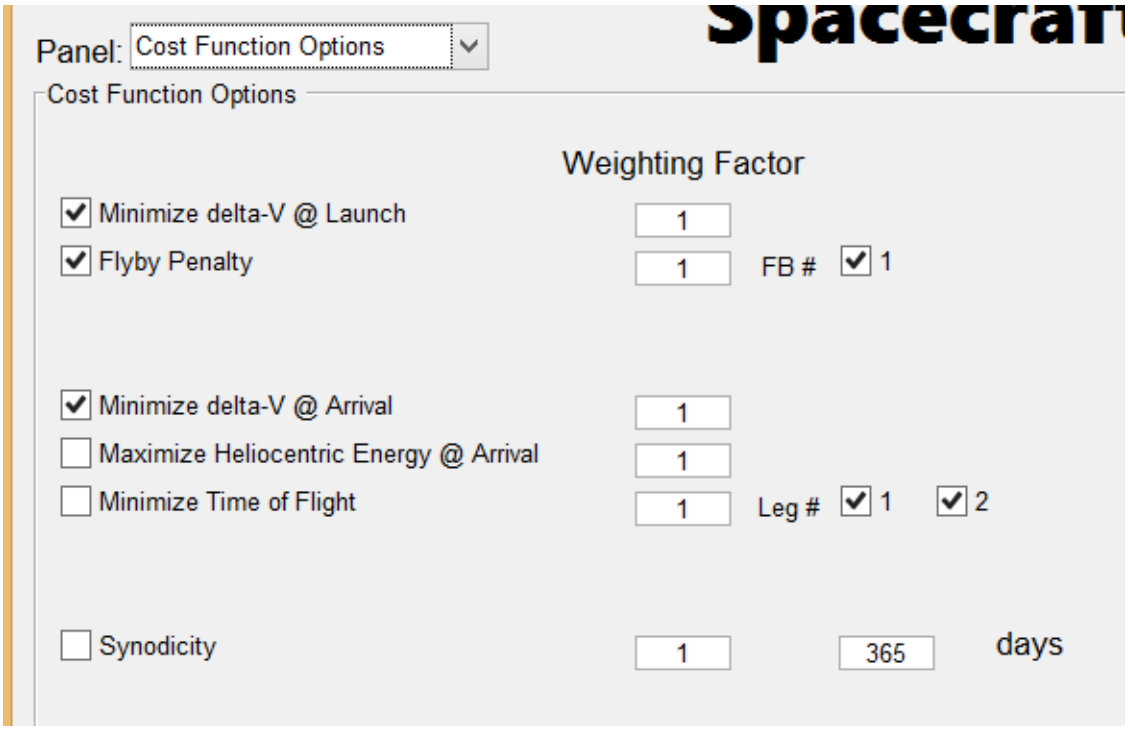


Figure 7.5. Cost Function Options Panel

The Actual Missions panel allows the user to show a few pre-defined real life missions that were used in the verification process for this work. They serve as good examples when the user is starting to learn how to use the interface.

The Custom Cost Function panel allows the user to implement the Island Model Paradigm (the central part of this work) on a custom cost function that does not fit within the bounds of this specific user interface. Some examples of custom cost functions would be GTOC problems [8].

The 3-D Trajectory panel is where the final optimized trajectory is shown. This is useful to show that the trajectory is feasible and realistic, and also adds to the aesthetic appeal of the interface. This panel is shown below in Figure 7.6. The top menu of the GUI also has a “Legend Window” button that pulls up a color coded legend for this panel. There is an option to animate the trajectory, or the user can change between zooming on the trajectory or rotating it in 3D.

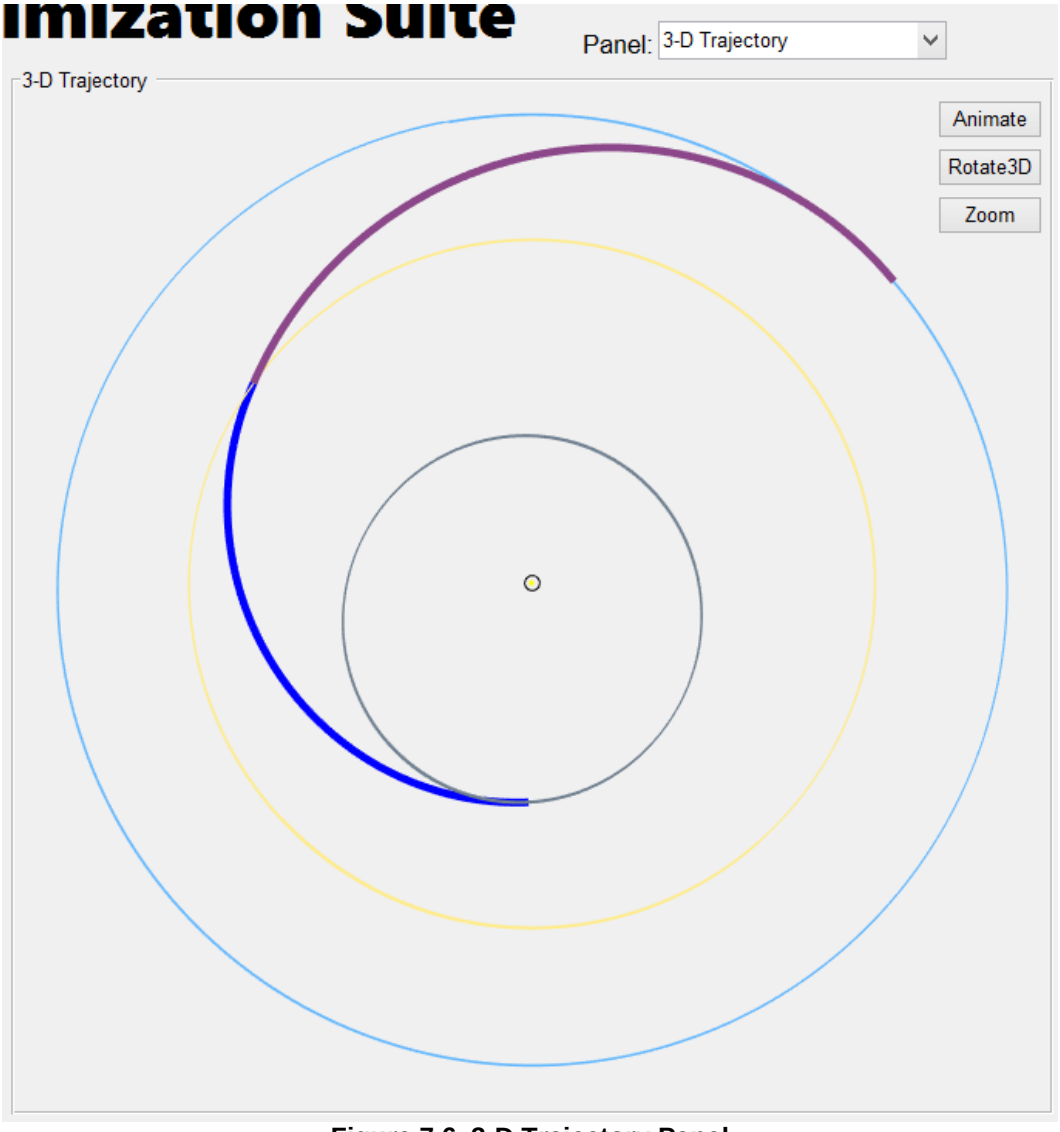


Figure 7.6. 3-D Trajectory Panel

The Cost Analysis panel shows the maximum, minimum, and average cost for each algorithm as they progress through their iterations. This panel is useful for determining which algorithms are performing the best or performing the worst if the user is looking to eliminate algorithms in the interest of increasing computational efficiency. This panel is filled after all migrations have occurred. At this point, the central progress panel pulls up checkboxes that allow the user to choose which islands' trends to show. There are also checkboxes that allow the user to toggle the visibility of the minimum, maximum, and average cost per iteration of all islands. This panel is shown below in Figure 7.7.

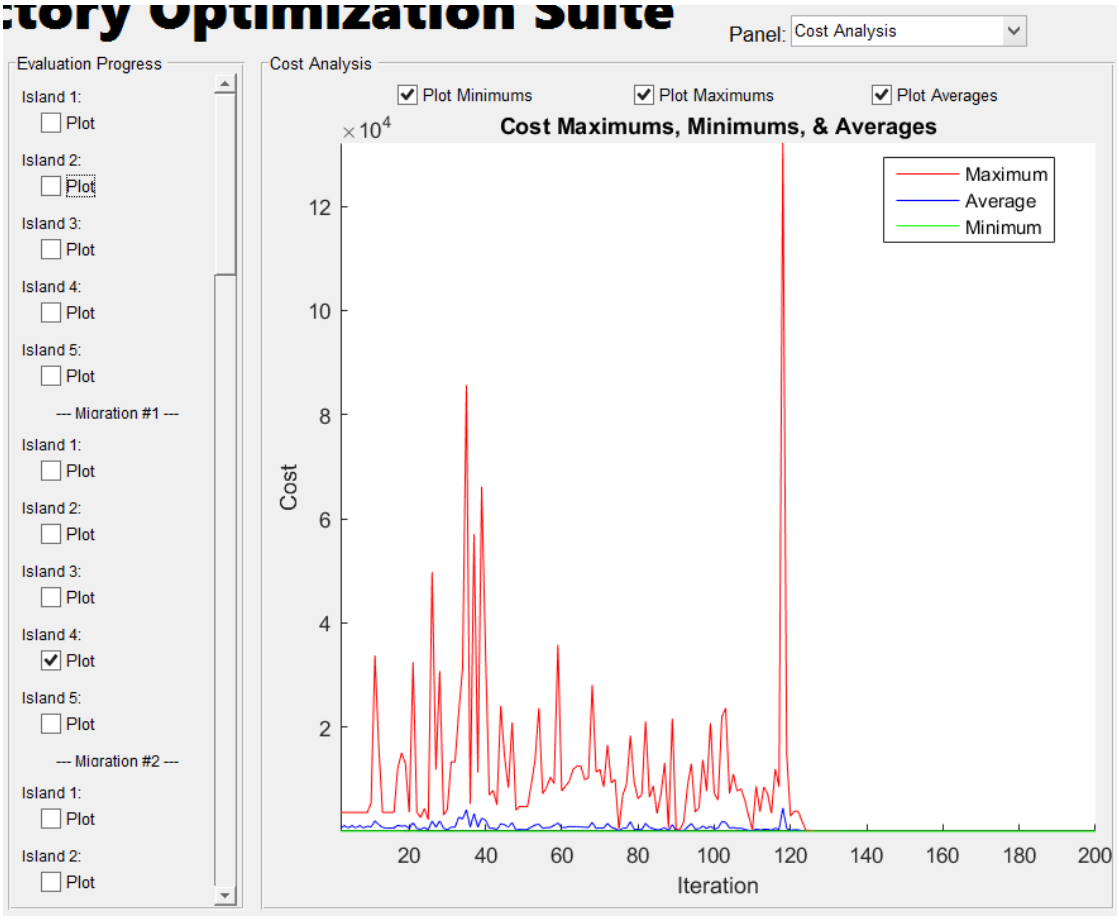


Figure 7.7. Cost Analysis Panel

The Detailed Results panel shows all information about the trajectory. It shows all possible information that could be used as values to be optimized. It also shows all relevant dates. This panel is shown below in Figure 7.8.

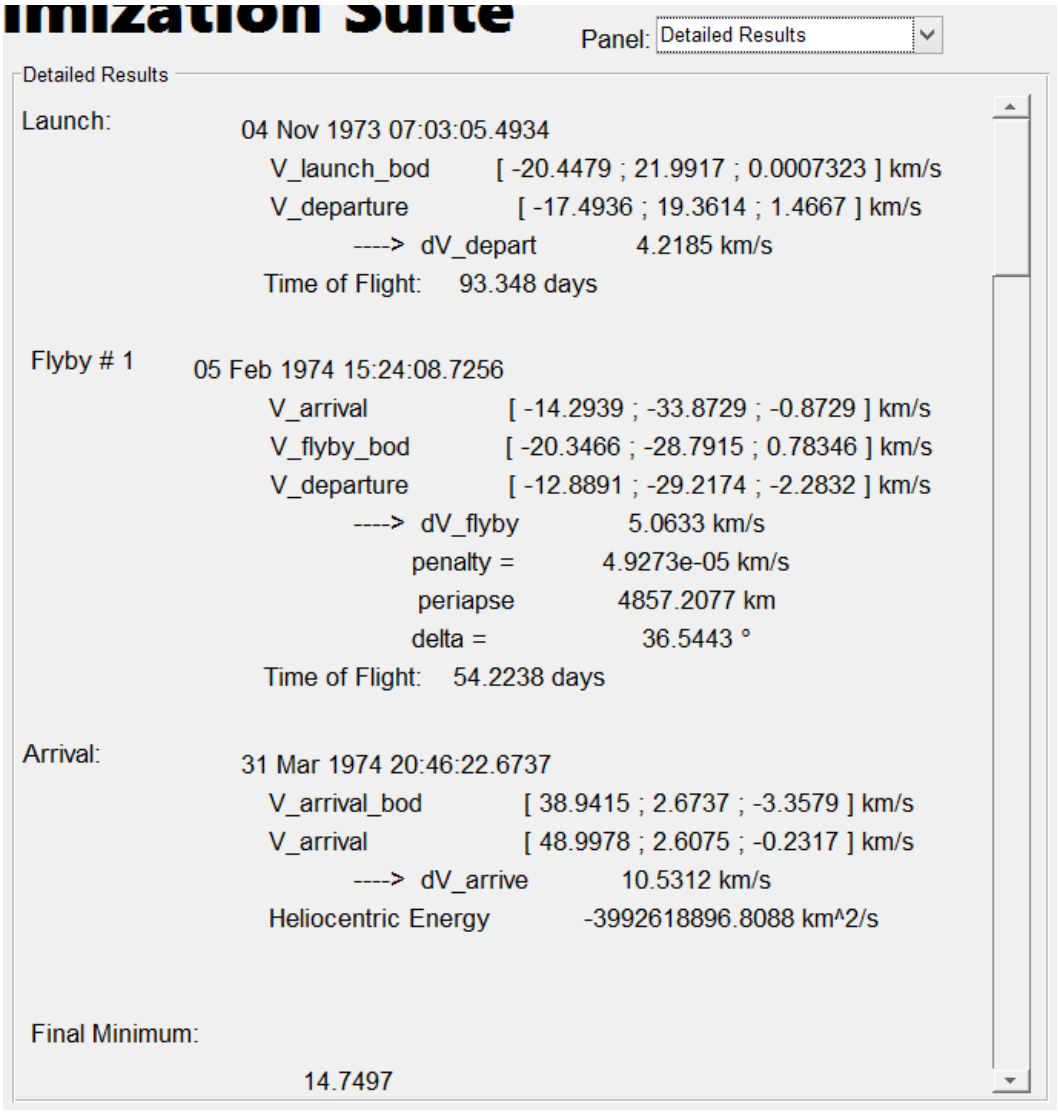


Figure 7.8. Detailed Results Panel

The Custom Results panel fills when a custom cost function is optimized. The final cost value is displayed, as well as all the variables that give that associated cost.

8. CONCLUSIONS

The Spacecraft Trajectory Optimization Suite created for this work successfully implemented five separate optimization algorithms (four stochastic methods for global search and one deterministic method for local search), both unique to this work and already existent in literature. These algorithms were verified with known multi-dimensional test functions, then run on actual spacecraft trajectory missions.

STOpS successfully found optimal trajectories for the Mariner 10 mission and the Voyager 2 mission that were similar to the actual missions flown. The costs observed here were lower costs than those found when using the dates for the actual missions, but the takeaway is not that STOpS found better trajectories than those actually flown for these missions. Instead, what is important is that STOpS demonstrated the capability to quickly and successfully analyze/plan these trajectories in the preliminary design phase of missions. The analysis for each of these missions took only 2-3 days each. When used for non-test case applications this time will likely be shorter since not every element of the tests performed here will be necessary. The development for STOpS took much longer, but the result is a robust tool that has taken existing techniques and applied them to the specific problem of trajectory optimization, so it can repeatedly and reliably solve these types of problems.

8.1 Future Work

There are certain generalizations that could be added to the suite, but were left for future work because although they would increase the fidelity of the solutions found, they would not affect the underlying optimization processes. Some other improvements were considered to be more coding based and less of a contribution to the engineering community, but they could easily be added later. These include physical effects of a particular spacecraft (drag, SRP, other orbital perturbations), the capability to optimize trajectories with different systems/bodies, including moons when examining flybys, and allowing for departure from specific parking orbits or arrival in particular parking orbits.

There were also some modifications/additions to particular optimization algorithms that interested the author, but time did not permit their full exploration. These include a variable threshold method for the Genetic Algorithm and the GA's behavior when both parents are identical: randomly searching nearby the parents. Since the static threshold method showed little utility, the variable threshold method may hold some promise. The identical parent mating method was intended to mimic a randomized local search, but it has been theorized by the author that a more valuable solution would be to have one child actually perform a univariate local search to find that solution's basin and have the other child undergo mutation. This combination explores the local area and promotes exploration later in the optimization stages.

When working on the Differential Evolution algorithm, the only method explored when using the jitter and dither techniques was a uniform random

distribution between the upper and lower limits for the scaling factor. The literature discussed some other options for the scaling factor distributions, including a log-normal or power law distribution (equations 2.36 and 2.38, respectively, in [12]).

There are also some methods that could be added to the Particle Swarm Optimization method. An alternative form of PSO uses the parameter *M* in addition to *K*. This new parameter determines the number of memory particles to use. The memory particles take the memory responsibility away from the movers. This division of responsibilities can potentially reduce the computational load of the algorithm and lead to improved convergence rates. In addition to determining the number of each type of particles, the user must also decide how many particles speak with each other (particle topology).

The Ant Colony Optimization algorithm poses some interesting questions in this work. Since it traditionally dealt with round trip problems, the application to spacecraft trajectories was essentially an open door. The method employed in this work saw success, but the author and committee members have theorized some additional methods. First, it may be possible to utilize a one-way method similar to that used here, but with the first city being a “ghost” city, so each initial city has a path leading to it that could have pheromone. It also may be possible to treat the trajectory as a round trip, with a ghost city connecting the last body to the first. The ants could start at any leg of the trajectory, then based on whatever trip is taken, the cost function knows to ‘ignore’ the ghost city and calculate the trajectory that works spatially and according to time.

A last potential improvement to ACO is a formulation that functions solely on pheromone levels. The fact that the current ACO formulation also relies on knowing all possible costs between nodes makes it very computationally expensive. While the author did begin to explore this option with an undesirable amount of success, it is possible that with more time and effort the formulation could be perfected.

One addition to this work that was not pursued due to time constraints was the analysis of orbital synodic periods. It has been theorized that given an initial range of inputs, the minima that occur may have a periodic re-appearance. That is to say, that since optimal trajectories are based heavily on planetary locations relative to each other, it could be useful for the program to determine the periodicity of the legs of the trajectories, as well as possibly the entire trajectory.

The last major area of improvement for this work would be to implement options for deep space maneuvers and low-thrust trajectories. These options can technically still be optimized here by utilizing the custom cost function interface, but having them built into the suite would be more practical. Due to time constraints, these aspects were left for future work.

BIBLIOGRAPHY

1. Brondel, Brian. Newton Cannon. Digital image. Wikipedia: The Free Encyclopedia. N.p., 13 Nov. 2010. Web. 4 Aug. 2015. <https://upload.wikimedia.org/wikipedia/commons/7/73/Newton_Cannon.svg>. Wikipedia contributors.

2. “How Orbits Work." Space Place. Ed. Kristen Erickson and Nancy Leon. NASA, 30 July 2015. Web. 04 Aug. 2015. http://spaceplace.nasa.gov/how-orbits-work/en/.

3. Lasunncty. Orbit1. Digital image. Wikipedia: The Free Encyclopedia. N.p., 10 Oct. 2007. Web. 4 Aug. 2015. https://commons.wikimedia.org/wiki/File:Orbit1.svg.

4. Wikipedia contributors. "Test functions for optimization." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 5 Jul. 2015. Web. 4 Aug. 2015.

5. Curtis, Howard D. Orbital Mechanics for Engineering Students. 2nd ed. Amsterdam: Elsevier, Butterworth-Heinemann, 2010. Print.

6. Adby, Paul R., and M. A. H. Dempster. Introduction to Optimization Methods. London: Chapman and Hall, 1974. Print.

7. Vega, Francisco Fernández De, Hidalgo Perez Jose Ignacio, and Juan Lanchares. "The Generalized Island Model." *Parallel Architectures and Bioinspired Algorithms.* Heidelberg: Springer-Verlag, 2012. N. pag. Print.

8. Izzo, Dario. Global Trajectory Optimization Competition (GTOC). GTOC Portal. European Space Agency's Advanced Concepts Team, n.d. Web. <http://sophia.estec.esa.int/gtoc_portal/>.

9. Goldberg, David E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley, 1989. Print.

10. Haupt, Randy L., and Sue Ellen Haupt. Practical Genetic Algorithms. 2nd ed. Hoboken, N.J: Wiley-Interscience, 2004. Print.

11. Dorigo, Marco, and Thomas Stützle. Ant Colony Optimization. Cambridge, MA: MIT, 2004. Print.

12. Price, Kenneth V., Rainer M. Storn, and Jouni A. Lampinen. Differential Evolution: A Practical Approach to Global Optimization. Berlin: Springer, 2005. Print.

13. Clerc, Maurice. Particle Swarm Optimization. London: ISTE, 2006. Print.

14. Surjanovic, Sonja, and Derek Bingham. Virtual Library of Simulation Experiments. Test Functions and Databases. Simon Frazer University, Jan. 2015. Web. 21 Sept. 2015. <http://www.sfu.ca/~ssurjano/ackley.html>.

15. Sparrow, Giles. Spaceflight: The Complete Story From Sputnik To Shuttle - And Beyond. London: Dorling Kindersley, 2007. Print.

16. "Mariner 10." National Space Science Data Center. Ed. E. Bell. NASA, 26 Aug. 2014. Web. 23 July 2015. <http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=1973-085A>.

17. "Voyager 2." National Space Science Data Center. Ed. E. Bell. NASA, 26 Aug. 2014. Web. 23 July 2015. <http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=1973-085A>.

18. Oldenhuis, Rody P.S. "Trajectory Optimization for a Mission to the Solar Bow Shock and Minor Planets." Thesis. Delft University of Technology, 2010. Web. <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CB0QFjAAahUKEwjMo966_oHJAhWBeyYKHSyJAYE&url=http%3A%2F%2Frepository.tudelft.nl%2Fassets%2Fuuid%3A6672aa53-6ecc-4571-85b2-2d4aafbba5bd%2FMSC_thesis_FINAL.pdf&usg=AFQjCNFtJcixINIkiaB8dQ-f39RsJ2NL6Q&sig2=ub38mio1aocLFhPamqH1Hg&bvm=bv.106923889,d.eWE&cad=rja>.

19. Vallado, David A., and Wayne D. McClain. Fundamentals of Astrodynamics and Applications. Dordrecht: Kluwer Academic, 2001. Print.

20. "HORIZONS Web-Interface." Jet Propulsion Laboratory. Ed. Ryan S. Park. NASA, 30 July 2015. Web. 30 July 2015. <http://ssd.jpl.nasa.gov/horizons.cgi>.

21. "Bullseye - Interplanetary Trajectory Simulation Software." N.p., 6 June 2015. Web. <http://www.sei.aero/sw/bullseye.html>.

22. "Bullseye - Interplanetary Trajectory Simulation Software." N.p., 6 June 2015. Web. <http://www.sei.aero/sw/bullseye.html>.

23. "Mission Analysis Environment (MAnE)." Space Flight Solutions, n.d. Web. 6 June 2015. <http://spaceflightsolutions.com/products/mane.asp>.

24. "MIDACO-SOLVER Global Optimization Software for Mixed Integer Nonlinear Programming." N.p., n.d. Web. 6 June 2015. <http://www.midaco-solver.com/index.php/about>.

25. "Java Astrodynamics Toolkit." NASA, n.d. Web. 6 June 2015. <http://opensource.gsfc.nasa.gov/projects/JAT/>.

26. Oldenhuis, Rody P.S. "Trajectory Optimization for a Mission to the Solar Bow Shock and Minor Planets." Thesis. Delft University of Technology, 2010. Print.

27. "Parallel Global Multidimensional Optimization." European Space Agency, n.d. Web. 6 June 2015. <https://github.com/esa/pagmo>.

28. Bryan, Jason M. "Global Optimization of MGA-DSM Problems Using the Interplanetary Gravity Assist Trajectory Optimizer (IGATO)." Thesis. California Polytechnic State University, 2011. Print.

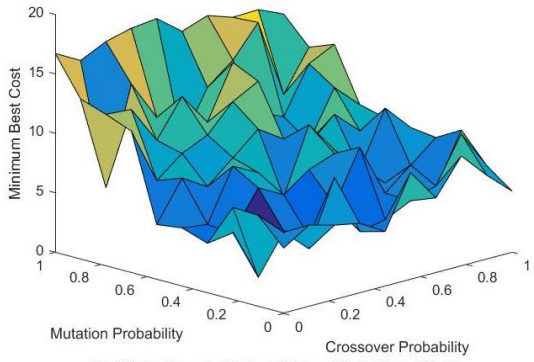
APPENDICES

APPENDIX A: Genetic Algorithm Verification

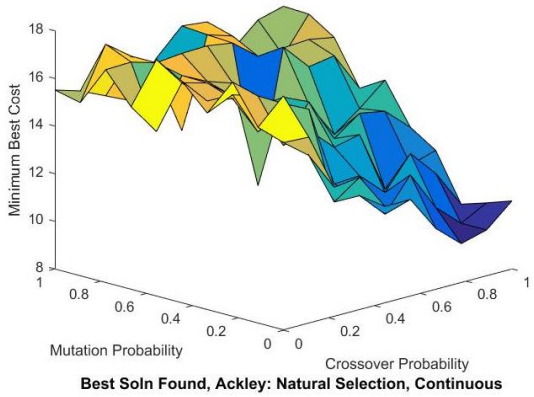
Comparison of Parameters: Step 1

N_{pop} = 100 N_{keep} = 30 N_{gen} = 20 p_c = [0,1] p_m = [0,1]

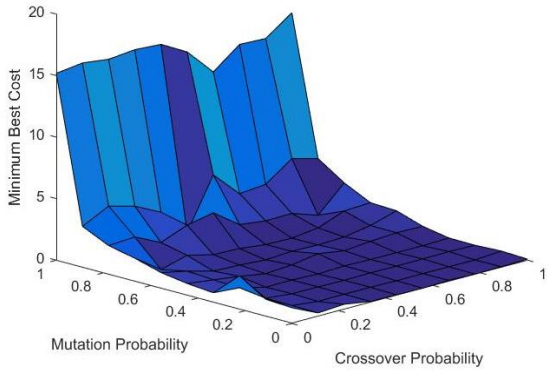
Best Soln Found, Ackley: Cost Weighted Random, Binary



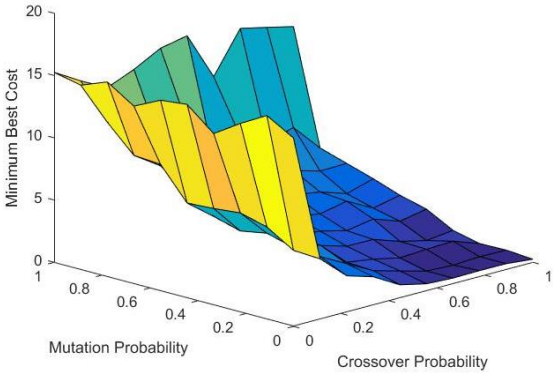
Best Soln Found, Ackley: Cost Weighted Random, Continuous



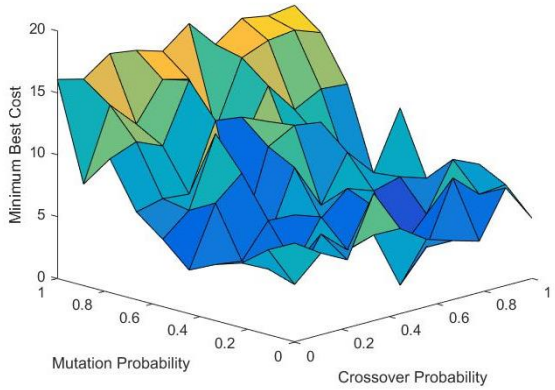
Best Soln Found, Ackley: Natural Selection, Binary



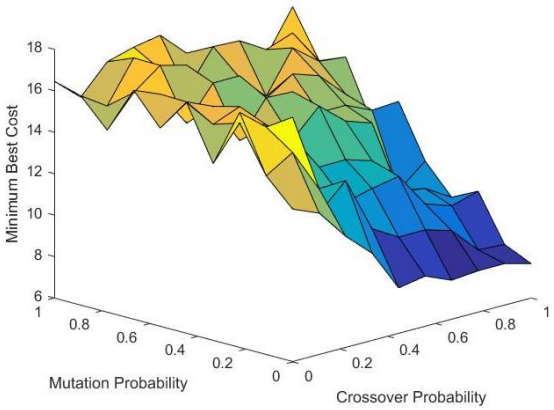
Best Soln Found, Ackley: Natural Selection, Continuous



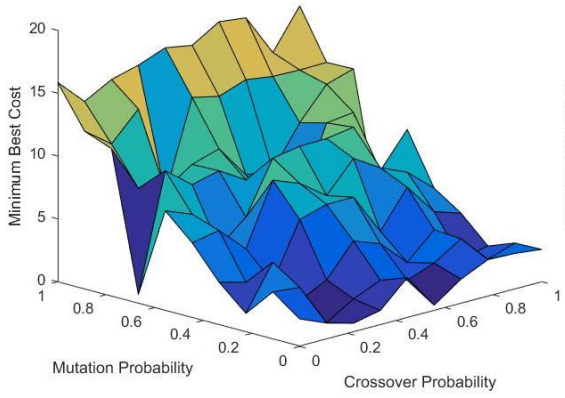
Best Soln Found, Ackley: Total Random Replacement, Binary



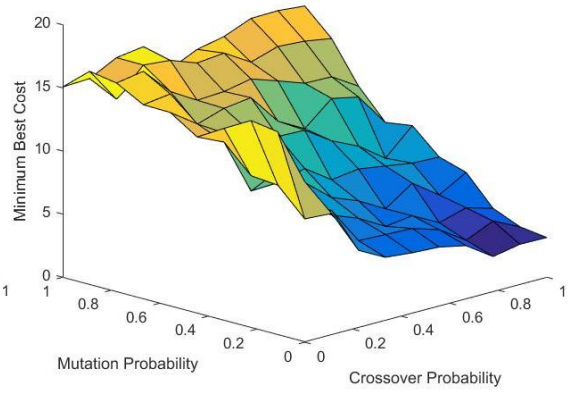
Best Soln Found, Ackley: Total Random Replacement, Continuous



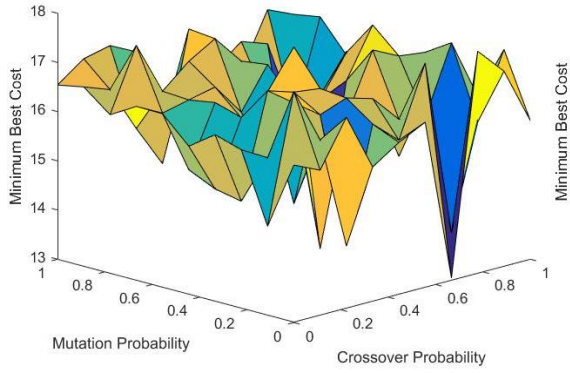
Best Soln Found, Ackley: Rank Weighted Random, Binary



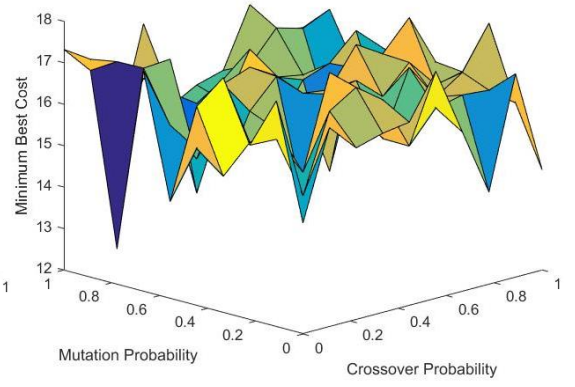
Best Soln Found, Ackley: Rank Weighted Random, Continuous



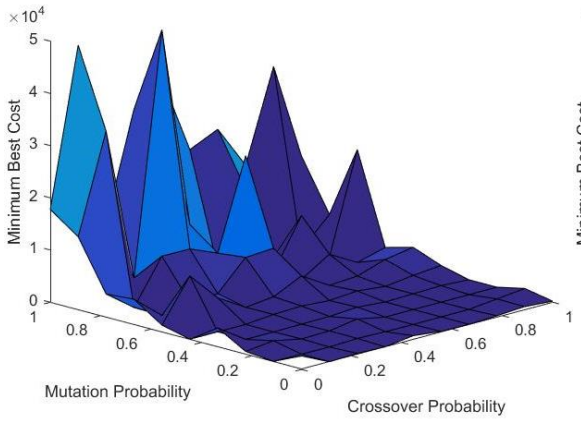
Best Soln Found, Ackley: Thresholding, Binary



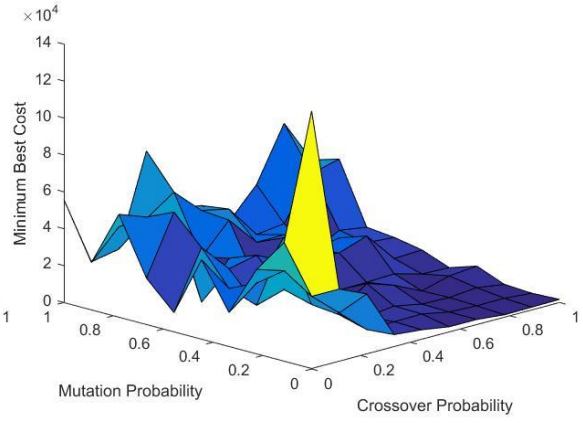
Best Soln Found, Ackley: Thresholding, Continuous



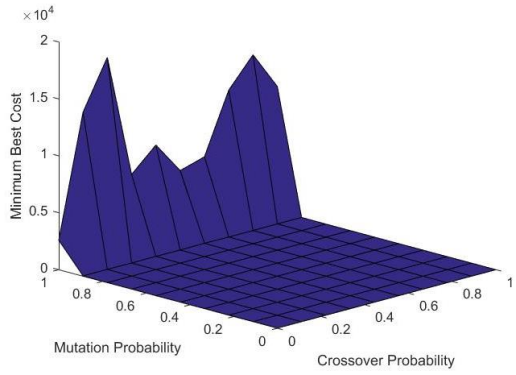
Best Soln Found, Rosenbrock: Cost Weighted Random, Binary



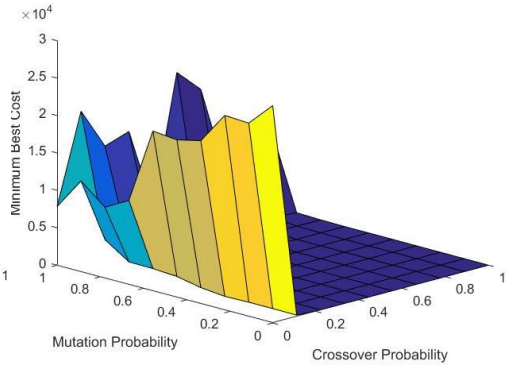
Best Soln Found, Rosenbrock: Cost Weighted Random, Continuous



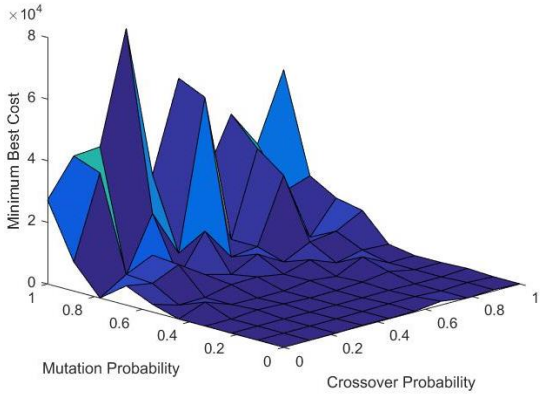
Best Soln Found, Rosenbrock: Natural Selection, Binary



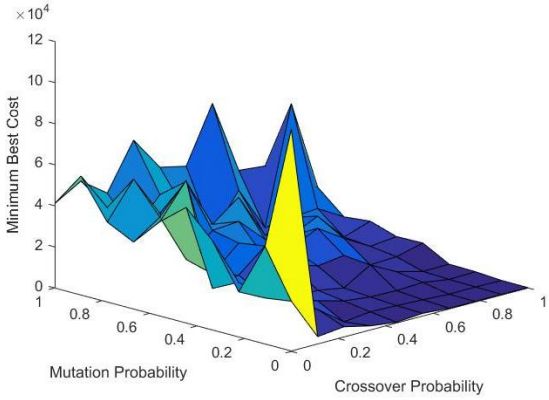
Best Soln Found, Rosenbrock: Natural Selection, Continuous



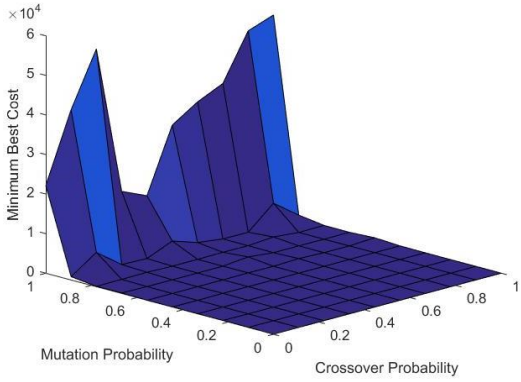
Best Soln Found, Rosenbrock: Total Random Replacement, Binary



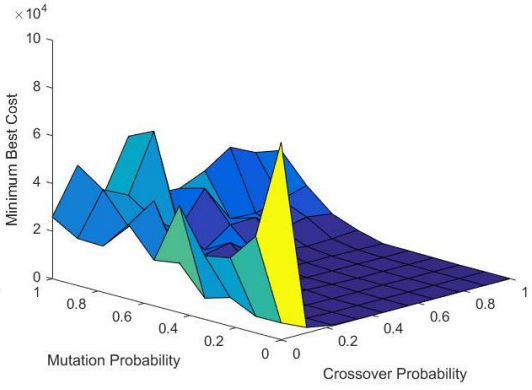
Best Soln Found, Rosenbrock: Total Random Replacement, Continuous

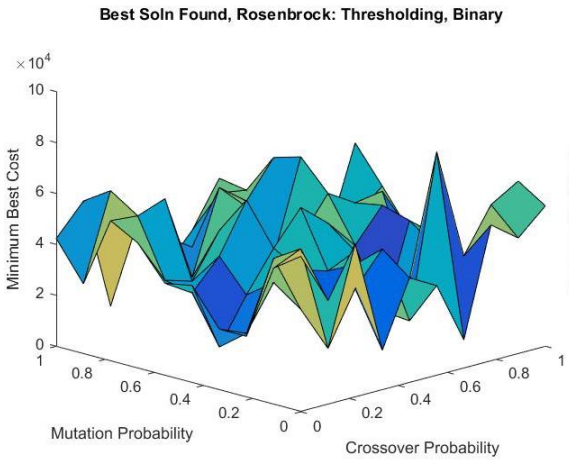


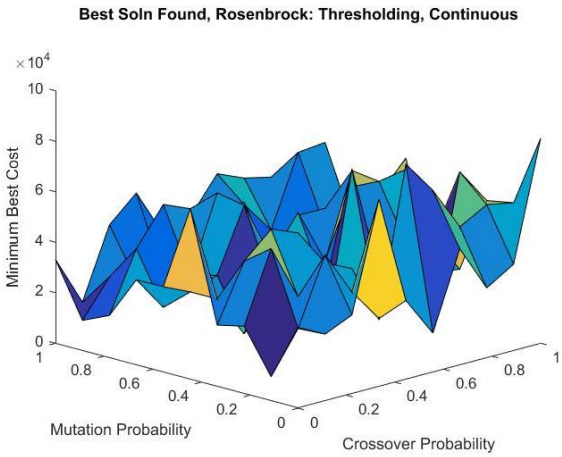
Best Soln Found, Rosenbrock: Rank Weighted Random, Binary



Best Soln Found, Rosenbrock: Rank Weighted Random, Continuous

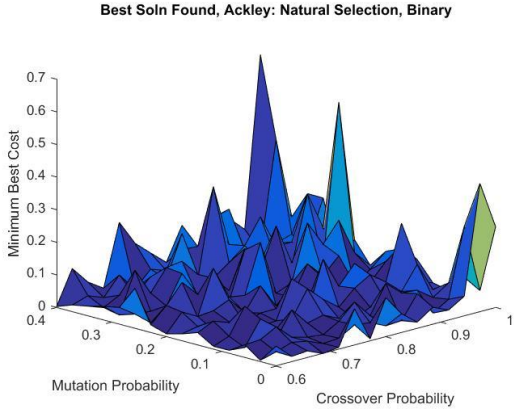


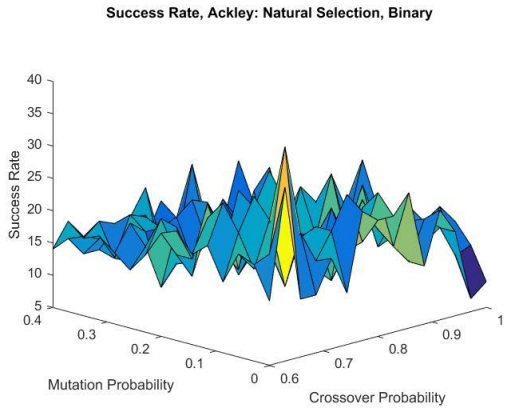


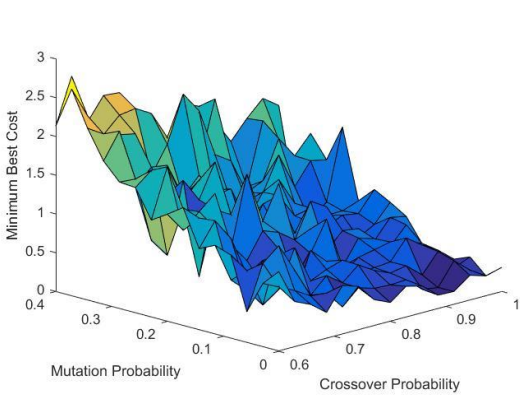


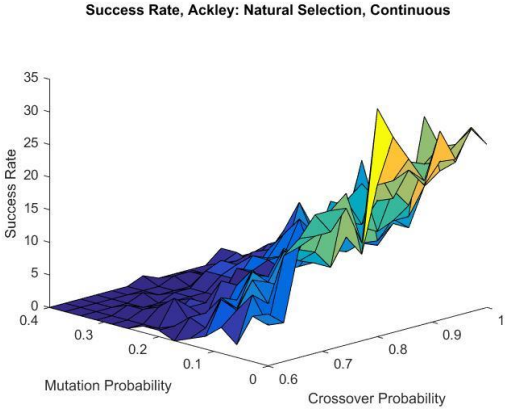
Comparison of Parameters: Step 2

N_{pop} = 100 N_{keep} = 30 N_{gen} = 20 p_c = [0.6,1] p_m = [0,0.4]
Ackley Success: <1.5 Rosenbrock Success: <10

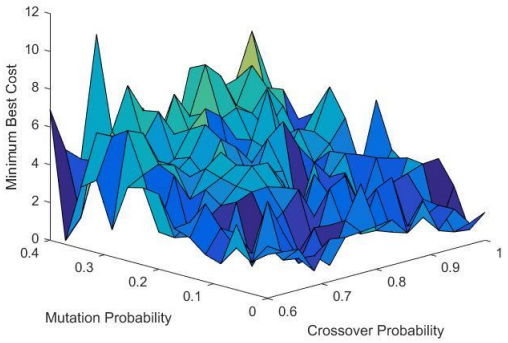




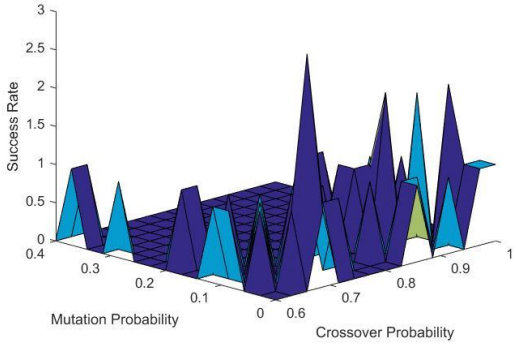




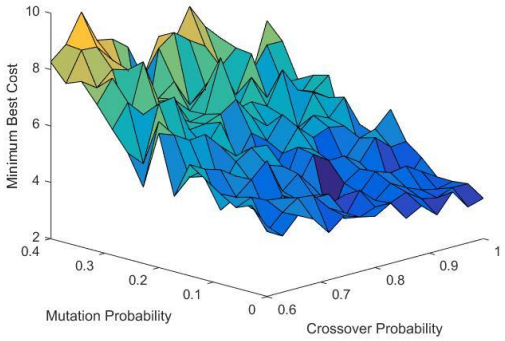
Best Soln Found, Ackley: Rank Weighted Random, Binary



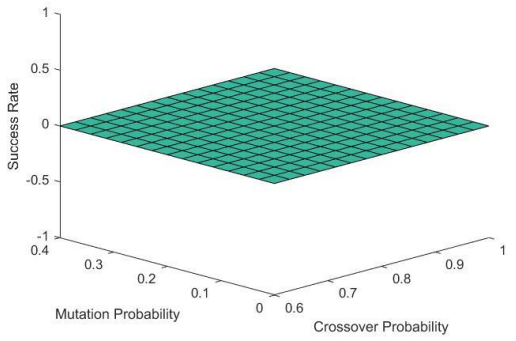
Success Rate, Ackley: Rank Weighted Random, Binary



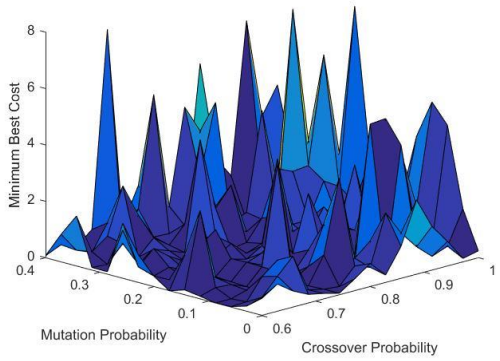
Best Soln Found, Ackley: Rank Weighted Random, Continuous



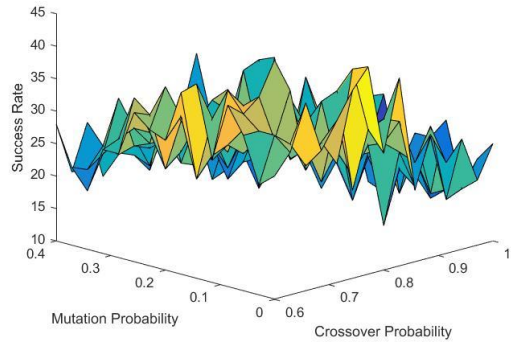
Success Rate, Ackley: Rank Weighted Random, Continuous



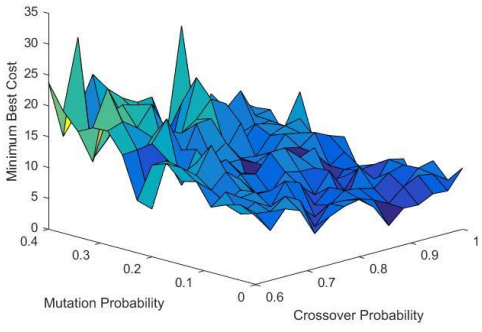
Best Soln Found, Rosenbrock: Natural Selection, Binary



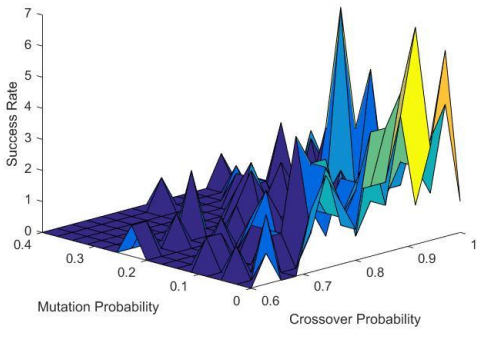
Success Rate, Rosenbrock: Natural Selection, Binary

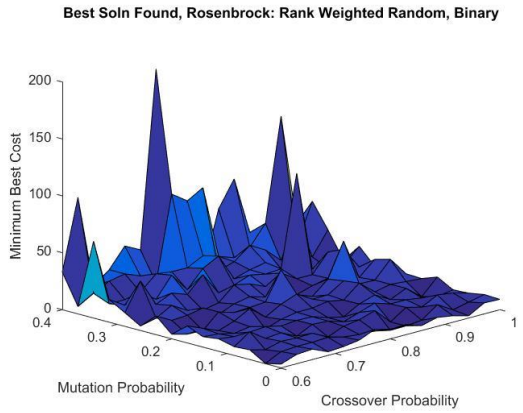


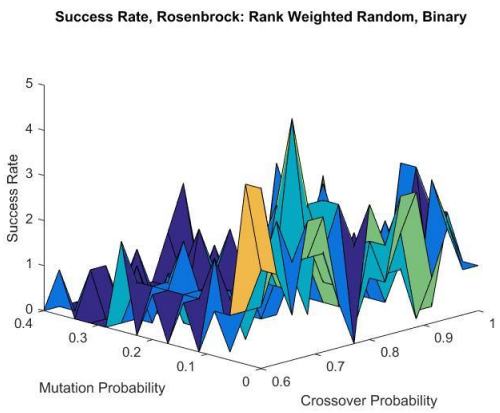
Best Soln Found, Rosenbrock: Natural Selection, Continuous

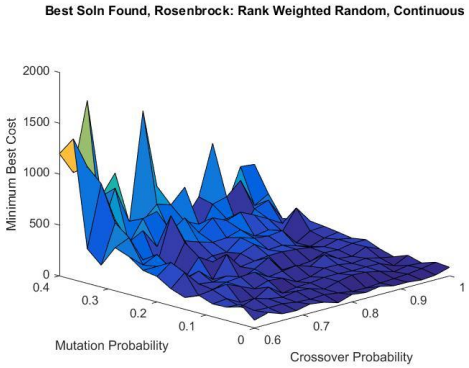


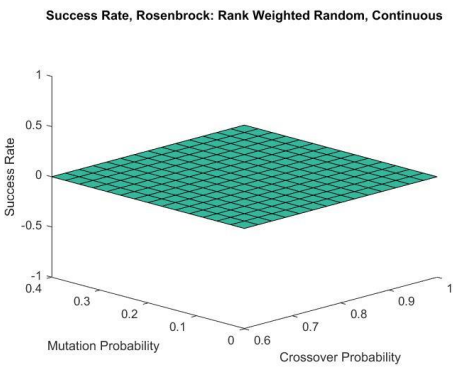
Success Rate, Rosenbrock: Natural Selection, Continuous




Best Soln Found, Rosenbrock: Rank Weighted Random, Binary
Minimum Best Cost
200
150
100
50
0
0.4
0.3
0.2
0.1
0
Mutation Probability
0.6
0.7
0.8
0.9
1
Crossover Probability


Success Rate, Rosenbrock: Rank Weighted Random, Binary
Success Rate
5
4
3
2
1
0
0.4
0.3
0.2
0.1
0
Mutation Probability
0.6
0.7
0.8
0.9
1
Crossover Probability


Best Soln Found, Rosenbrock: Rank Weighted Random, Continuous
Minimum Best Cost
2000
1500
1000
500
0
0.4
0.3
0.2
0.1
0
Mutation Probability
0.6
0.7
0.8
0.9
1
Crossover Probability

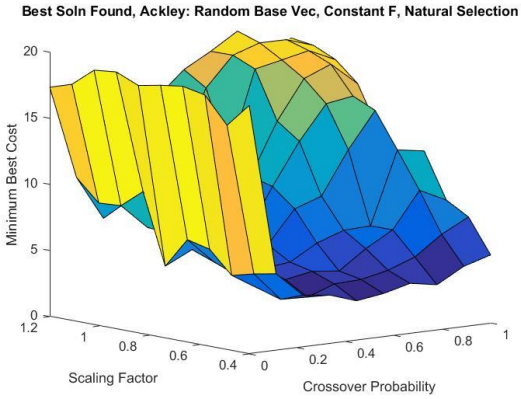

Success Rate, Rosenbrock: Rank Weighted Random, Continuous
Success Rate
1
0.5
0
-0.5
-1
0.4
0.3
0.2
0.1
0
Mutation Probability
0.6
0.7
0.8
0.9
1
Crossover Probability

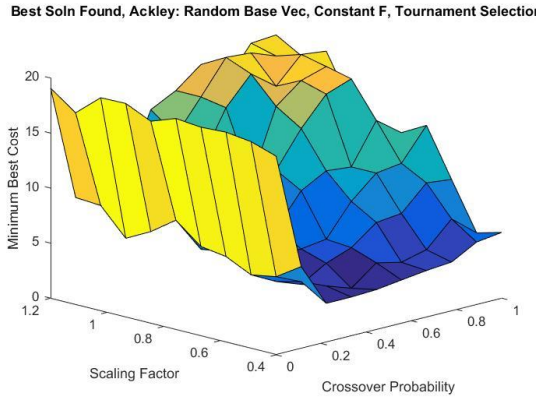
APPENDIX B: Differential Evolution Verification

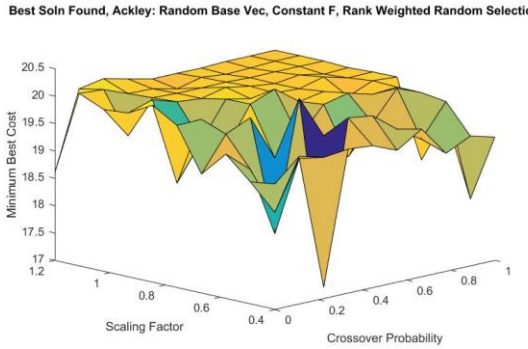
Step 0: Comparison of Survivor Selection Methods

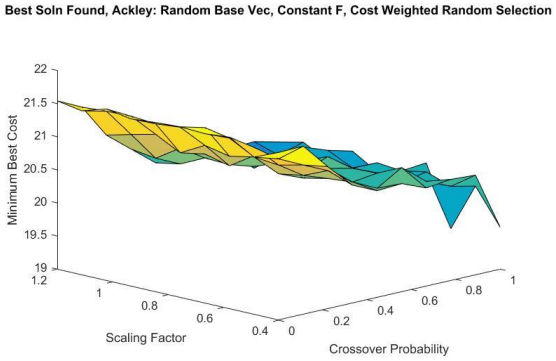
N_{pop} = 50 N_{gen} = 30 p_c = [0,1] F = [0.4,1.2] T = 3

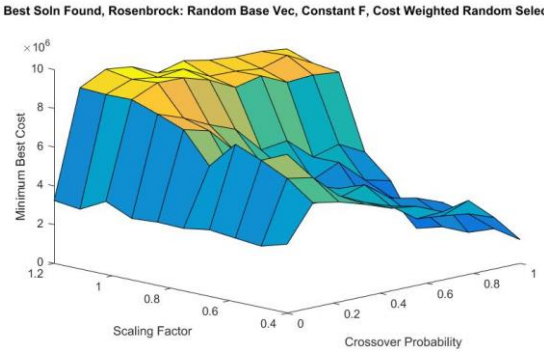
Takeaway: Both rank weighted random and cost weighted random are extremely ineffective compared to random and tournament selection. Consequently, they will not be included in the STOpS GUI.

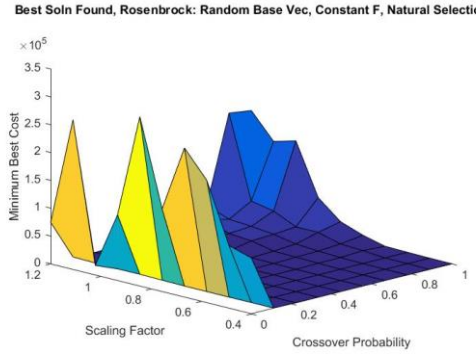




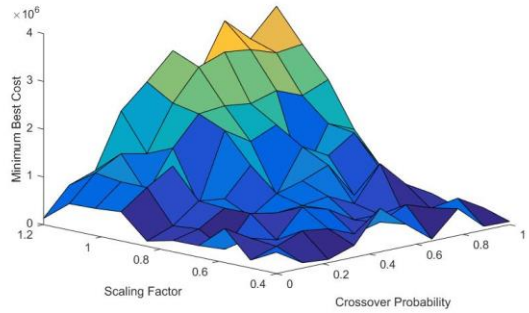




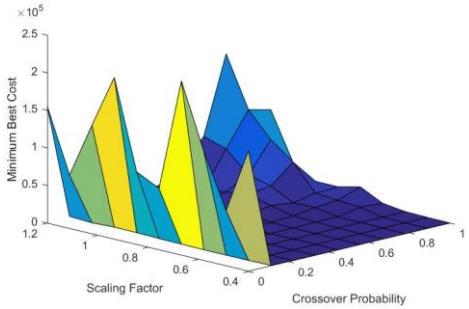




Best Soln Found, Rosenbrock: Random Base Vec, Constant F, Rank Weighted Random Selection



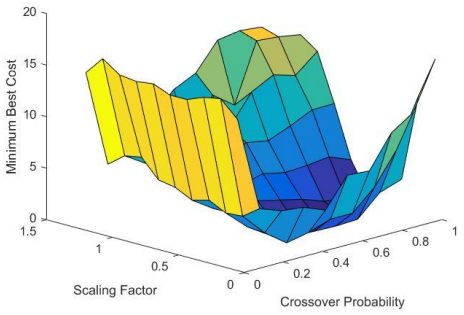
Best Soln Found, Rosenbrock: Random Base Vec, Constant F, Tournament Selection



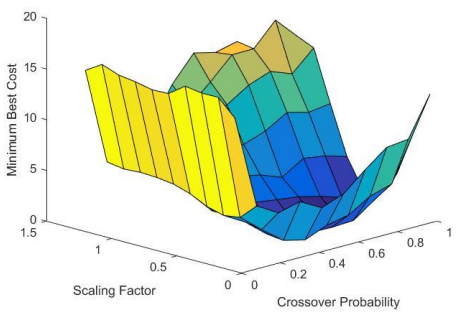
Comparison of Parameters: Step 1

N_{pop} = 100 N_{gen} = 20 p_c = [0,1] F = [0.05,1.2]

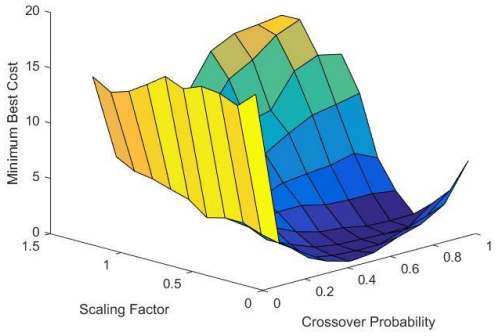
Best Soln Found, Ackley: Best So Far Base Vec, Constant F, Natural Selection



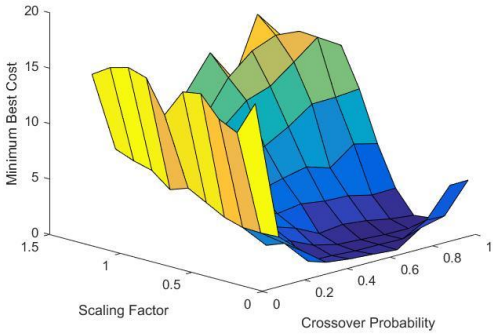
Best Soln Found, Ackley: Best So Far Base Vec, Constant F, Tournament



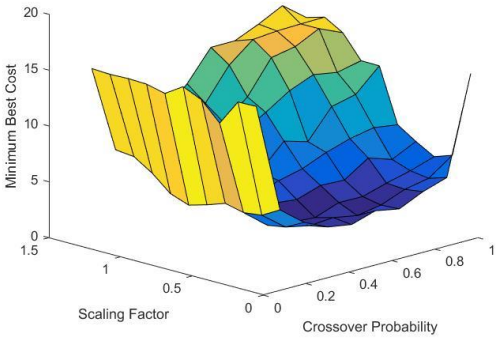
Best Soln Found, Ackley: Blend Base Vec, Constant F, Natural Selection



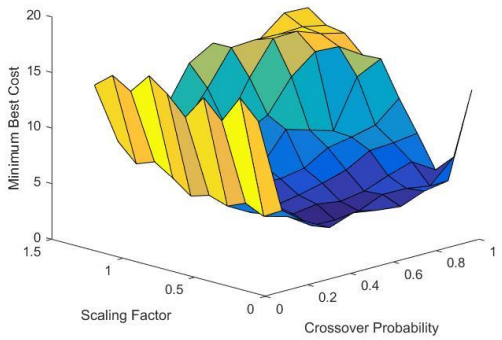
Best Soln Found, Ackley: Blend Base Vec, Constant F, Tournament



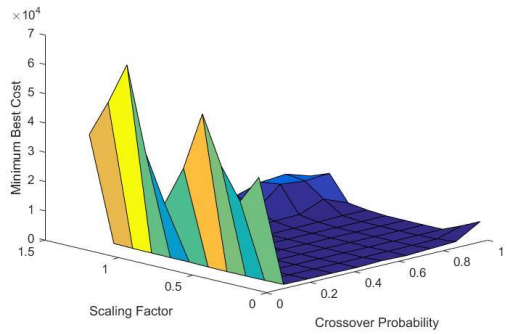
Best Soln Found, Ackley: Random Base Vec, Constant F, Natural Selection



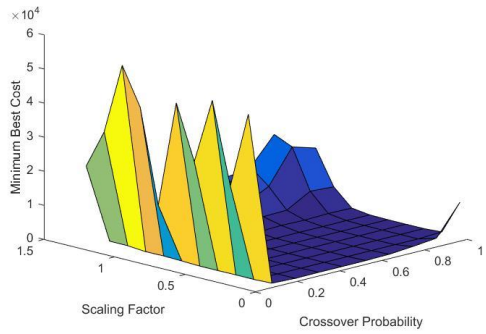
Best Soln Found, Ackley: Random Base Vec, Constant F, Tournament



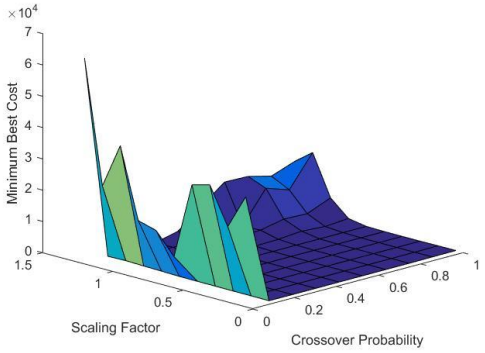
Best Soln Found, Rosenbrock: Best So Far Base Vec, Constant F, Natural Selection



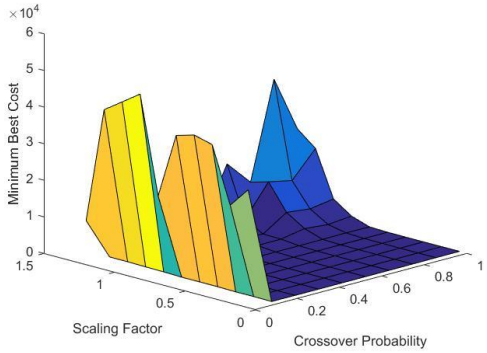
Best Soln Found, Rosenbrock: Best So Far Base Vec, Constant F, Tournament



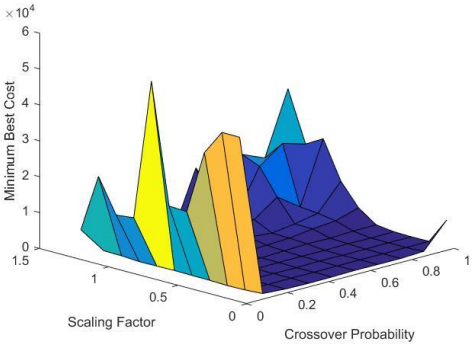
Best Soln Found, Rosenbrock: Blend Base Vec, Constant F, Natural Selection



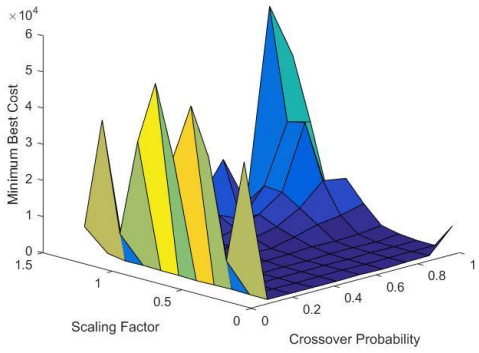
Best Soln Found, Rosenbrock: Blend Base Vec, Constant F, Tournament



Best Soln Found, Rosenbrock: Random Base Vec, Constant F, Natural Selectio



Best Soln Found, Rosenbrock: Random Base Vec, Constant F, Tournament



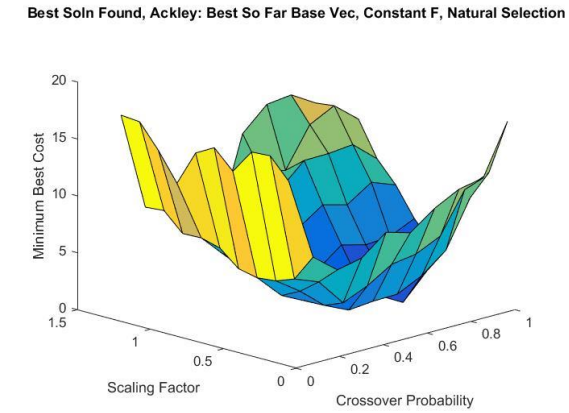
Comparison of Parameters: Step 1B, Tournament Exploration

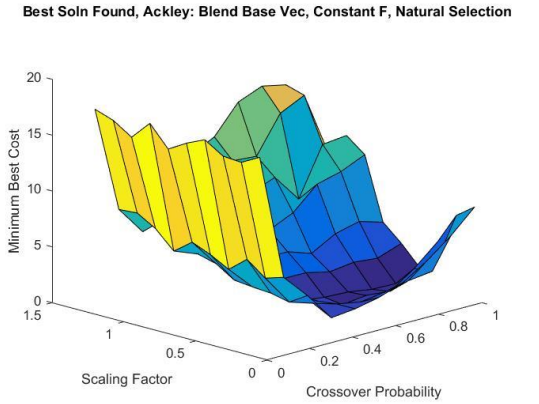
N_{pop} = 30 N_{gen} = 20 p_c = [0,1] F = [0.05,1.2] T = 5,10,15

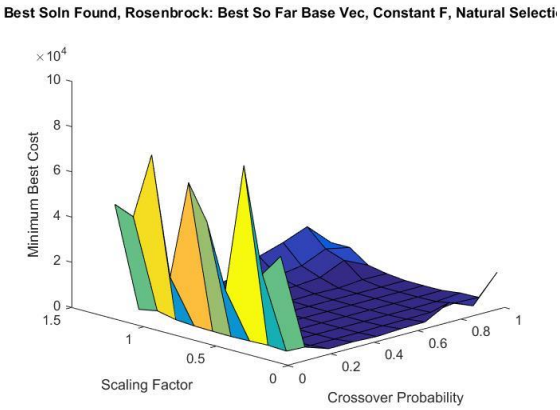
	Ackley's Fxn		Rosenbrock's Fxn	
	T	Success Rate	T	Success Rate
Best So Far	5	84%	5	49%
	10	88%	10	42%
	15	86%	15	43%
Blend	5	89%	5	43%
	10	93%	10	49%
	15	89%	15	43%

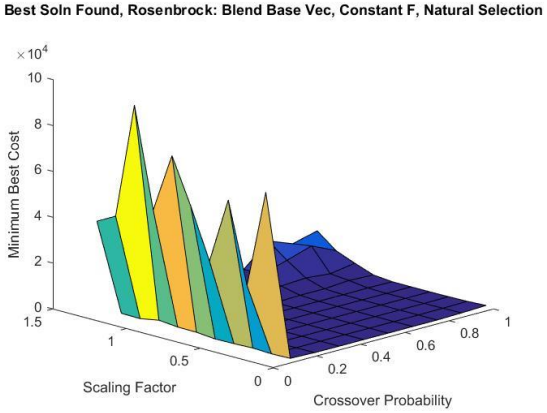
Comparison of Parameters: Step 2

N_{pop} = 30 N_{gen} = 20 p_c = [0,1] F = [0.05,1.2]





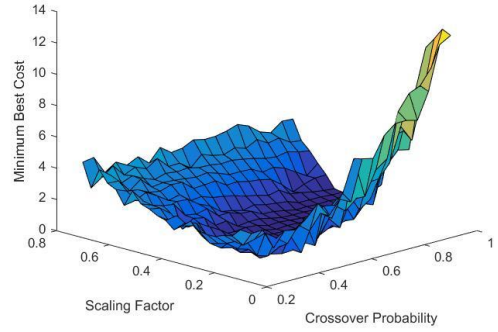




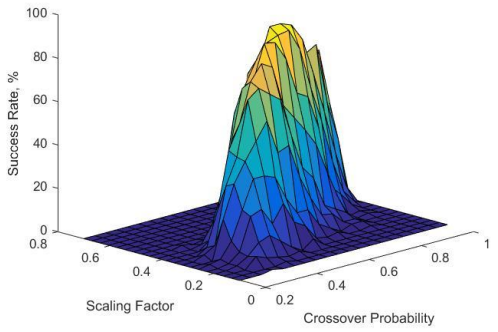
Comparison of Parameters: Step 4

N_{pop} = 100 N_{gen} = 20 p_c = [0.2,0.8] F = [0.2,0.7]

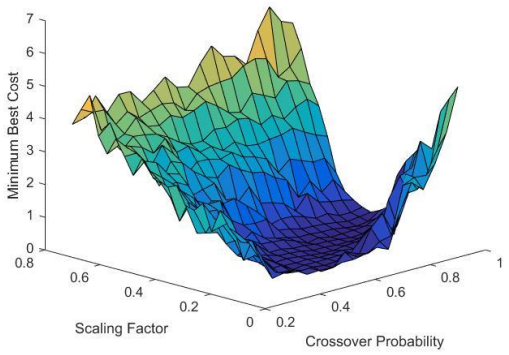
Best Soln Found, Ackley: Best So Far Base Vec, Constant F, Natural Selection



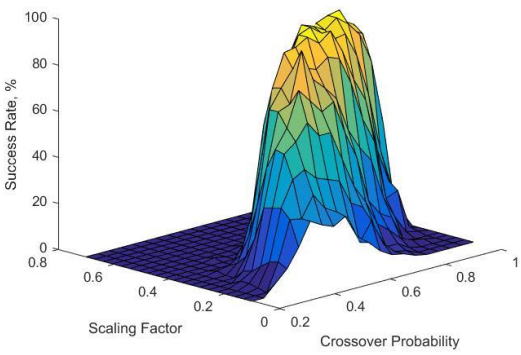
Success Rate, Ackley: Best So Far Base Vec, Constant F, Natural Selection



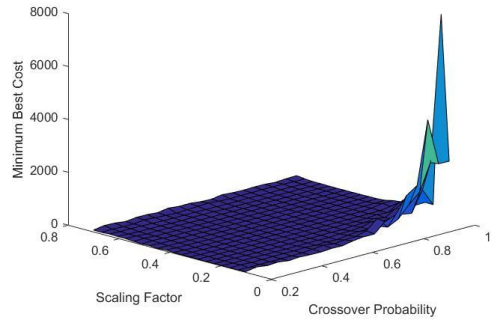
Best Soln Found, Ackley: Blended Mix Base Vec, Constant F, Natural Selection



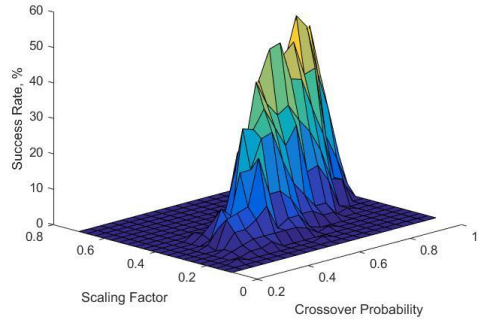
Success Rate, Ackley: Blended Mix Base Vec, Constant F, Natural Selection



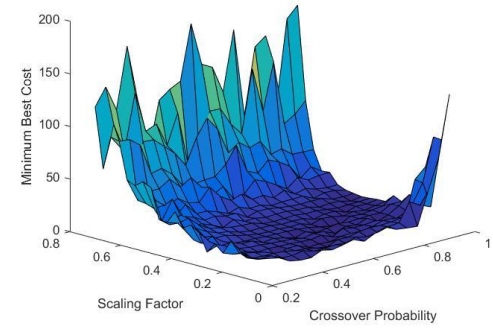
Best Soln Found, Rosenbrock: Best So Far Base Vec, Constant F, Natural Selecti



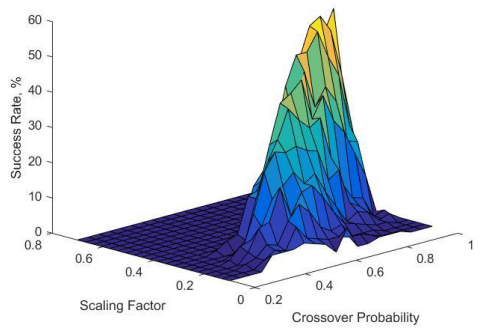
Success Rate, Rosenbrock: Best So Far Base Vec, Constant F, Natural Selectio



Best Soln Found, Rosenbrock: Blended Mix Base Vec, Constant F, Natural Selecti



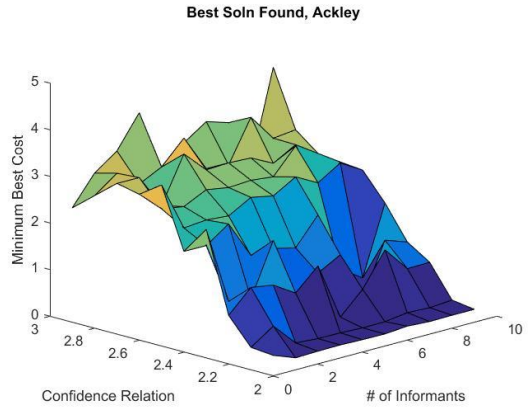
Success Rate, Rosenbrock: Blended Mix Base Vec, Constant F, Natural Selectio

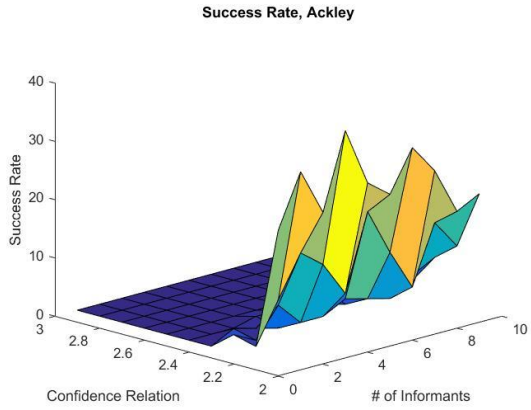


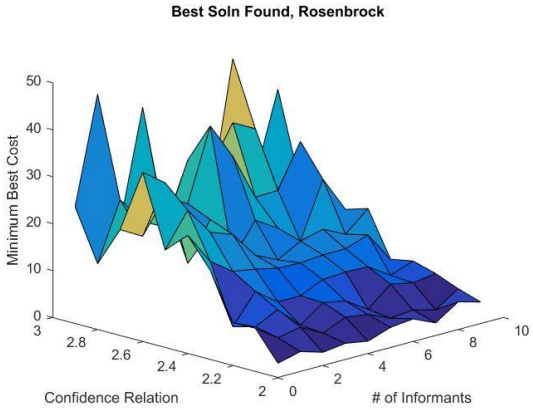
APPENDIX C: Particle Swarm Optimization Verification

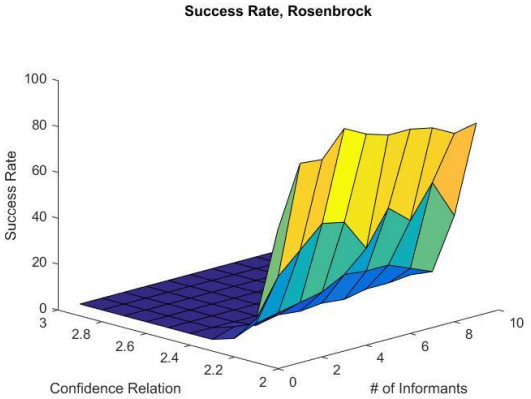
Comparison of Parameters: Step 1

N_{pop} = 50 t_{span} = 200 K = [1,10] φ = [2.1,3]

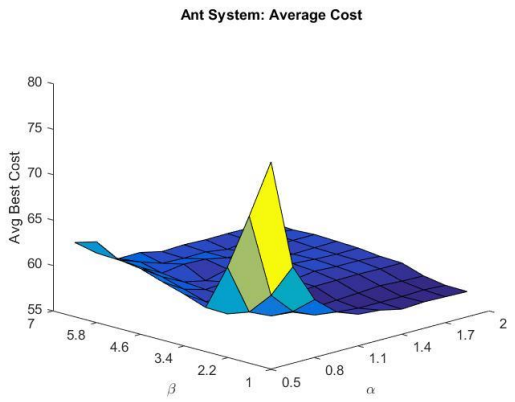


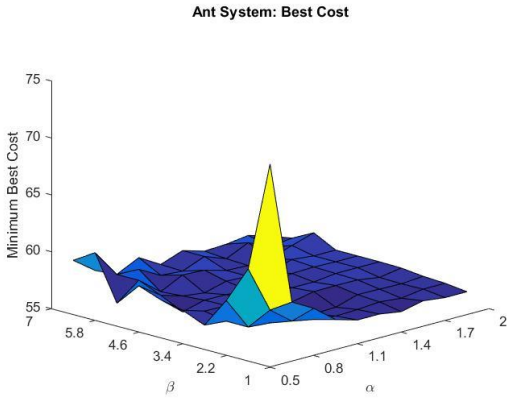


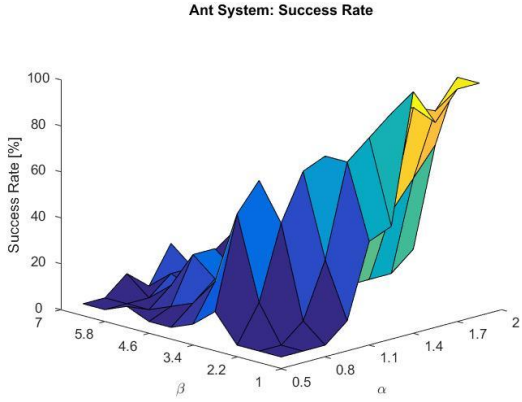


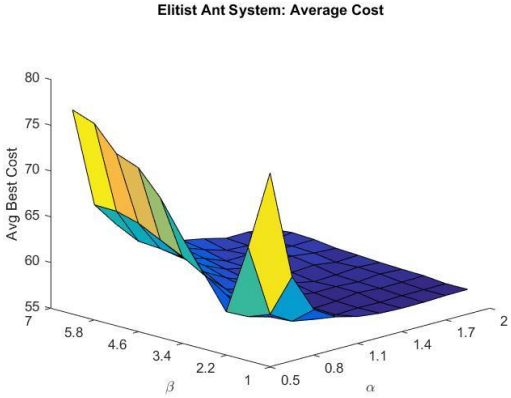


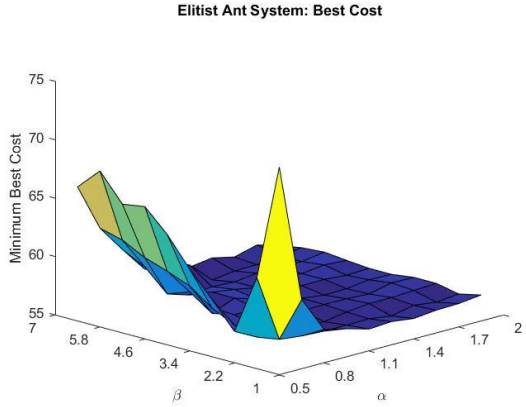
APPENDIX D: Ant Colony Verification

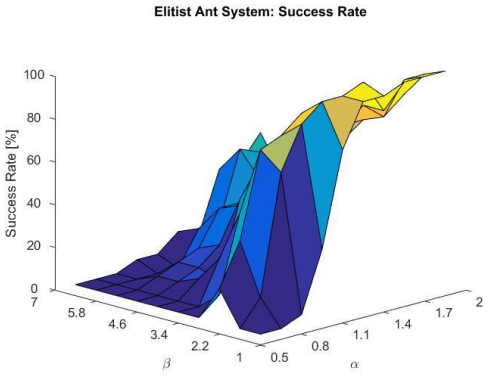


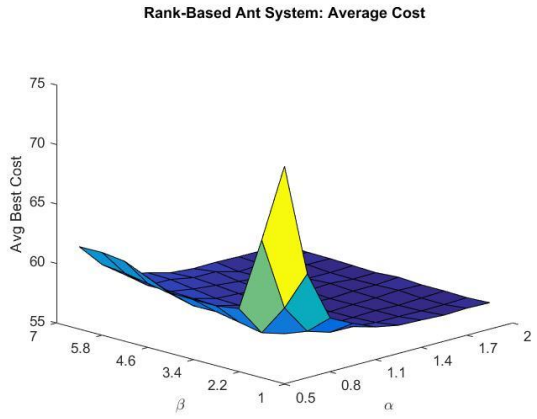


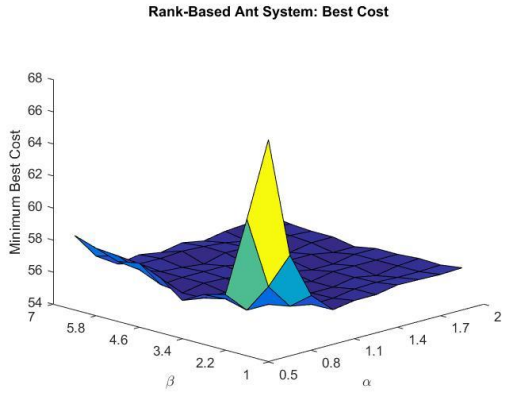


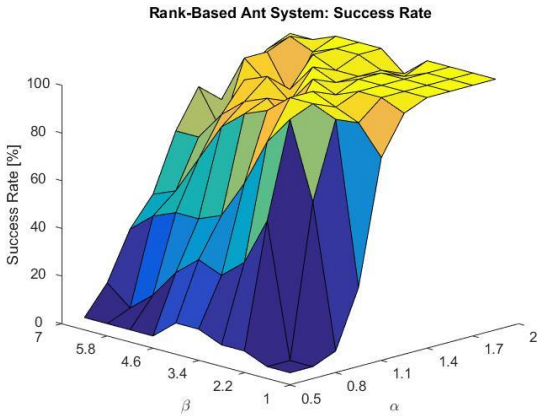


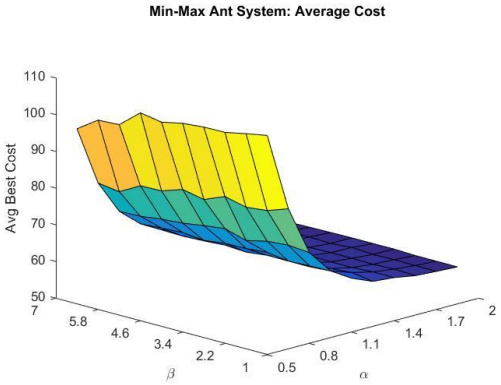


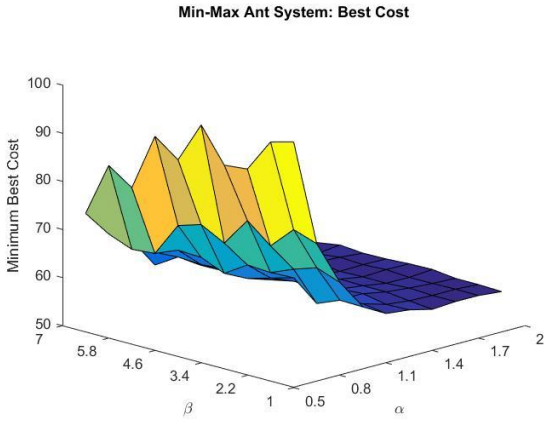


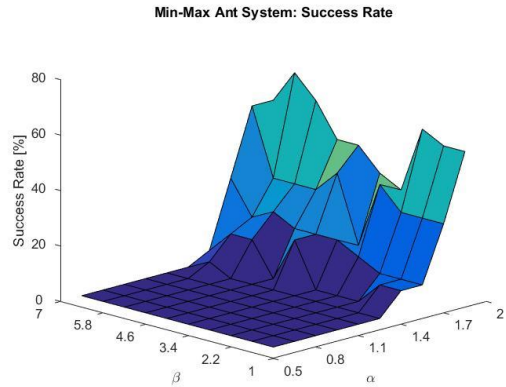

Rank-Based Ant System: Average Cost
Avg Best Cost
75
70
65
60
55
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α

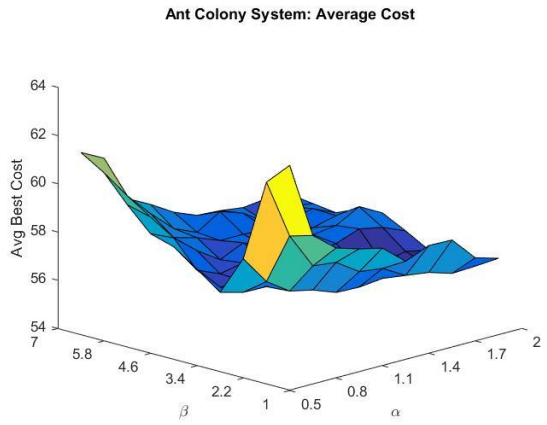

Rank-Based Ant System: Best Cost
Minimum Best Cost
68
66
64
62
60
58
56
54
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α

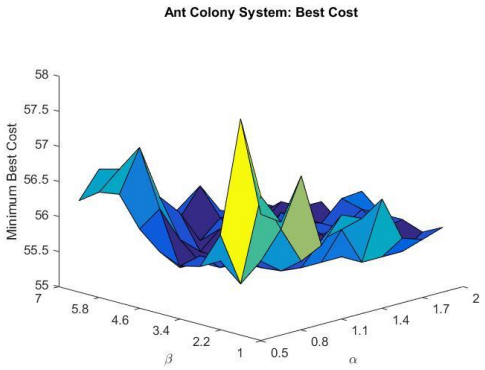

Rank-Based Ant System: Success Rate
Success Rate [%]
100
80
60
40
20
0
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α

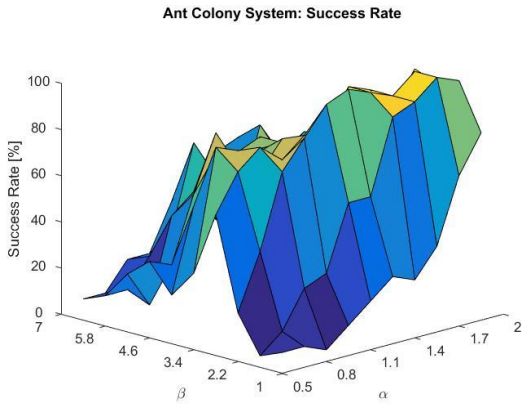

Min-Max Ant System: Average Cost
Avg Best Cost
110
100
90
80
70
60
50
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α


Min-Max Ant System: Best Cost
Minimum Best Cost
100
90
80
70
60
50
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α


Min-Max Ant System: Success Rate
Success Rate [%]
80
60
40
20
0
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α


Ant Colony System: Average Cost
Avg Best Cost
64
62
60
58
56
54
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α


Ant Colony System: Best Cost
Minimum Best Cost
58
57.5
57
56.5
56
55.5
55
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α


Ant Colony System: Success Rate
Success Rate [%]
100
80
60
40
20
0
7
5.8
4.6
3.4
2.2
1
β
0.5
0.8
1.1
1.4
1.7
2
α