THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF AEROSPACE ENGINEERING

BIOMIMETIC METHODS OF OPTIMIZING TWO-IMPULSE ORBITAL TRANSFERS

ALEXANDER BOROWSKI
Spring 2016

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Aerospace Engineering
with honors in Aerospace Engineering

ABSTRACT

Particle Swarm Optimization (PSO) and Bacteria Foraging Optimization (BFO) are both heuristic optimization methods that belong to a class of algorithms that are known as biomimetic. PSO is a population-based, stochastic method that models the behavior of swarms of insects and birds as they search for food. BFO simulates the process of bacteria searching for food, reproducing, and dying. Both techniques rely on information sharing between particles to allow the whole population to converge on an optimal configuration. These algorithms have various applications, including orbital trajectory optimizations. This thesis applies the algorithms to solve for the optimal impulsive transfer between two circular orbits, which is known to be the Hohmann transfer. PSO and BFO are then compared to determine which method is less resource-intensive and can converge on an accurate solution with less error. Future areas of study will include investigating the comparative effectiveness of these two algorithms in optimizing different types of trajectory maneuvers.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Nomenclature

Δv_1 – first impulsive change in velocity

Δv_1 – second impulsive change in velocity

δ_1 – angular displacement of first impulse

δ_2 – angular displacement of second impulse

ε_H – energy of the Hohmann transfer ellipse

μ_B – standard gravitational parameter of Earth, $3.98600 * 10^5\ km^3 s^{-2}$

a – semimajor axis of ellipse

B – position of a bacterium

BFO – bacterial foraging optimization

BLp – the lower bound of the particle's position

BLv – the lower bound of the particle's velocity

BUp – the upper bound of the particle's position

BUv – the upper bound of the particle's velocity

c_C – cognitive weight coefficient

c_I – inertial weight coefficient

c_S – social weight coefficient

C_0 – magnitude of the swim-step

DU – distance unit (initial radius)

e – eccentricity of ellipse

G_{best} – best fitness value achieved by a particular particle

J – the sum of Δv_1 and Δv_2, the fitness function

P – position of a particle

P_{best} – best position achieved by a particular particle

PSO – particle swarm optimization

r – radius of orbit

R_1 – initial radius

R_2 – final radius

t_0^- – time immediately before first impulse

t_0^+ – time immediately after first impulse

t_f^- – time immediately before second impulse

t_f^+ – time immediately after second impulse

TU – time unit (such that $\mu_B = \frac{DU^3}{TU^2}$)

u_i – unit vector along which a bacterium tumbles and swims

v_1 – velocity on first circular orbit

v_2 – velocity on second circular orbit

v_a – velocity at apoapsis

v_p – velocity at periapsis

v_r – velocity in the radial direction

v_θ – velocity in the horizontal direction

V − velocity value of a particle

Chapter 1

Introduction

Determining optimal space trajectories has been a source of study for many decades. As government programs begin to look towards missions to Mars and private companies begin launching their own space vehicles, the need to find optimum trajectories that minimize fuel consumption becomes even more important. Optimization methods are generally categorized as either deterministic or stochastic (i.e., involving some random search elements).[1,2] Historically, optimization problems with many parameters can be computationally expensive or flat out impossible to solve using deterministic, gradient-based methods. Rather than explicitly determining an equation for optimization, stochastic methods exploit a randomized population that represents many possible solutions to a problem. Stochastic methods can sometimes also be described as heuristic, meaning they are not mathematically rigorous in their nature. A common way of developing these types of algorithms is to analyze natural processes.

Imitating nature is historically a great way of solving complicated engineering problems. Velcro is perhaps the most famous example of engineering mirroring biology. Biomimetic algorithms are stochastic methods that are derived from observing natural phenomena and modeling their behavior. These algorithms have been shown to be able to solve size optimization in structural engineering, describe complex chemical processes, determine optimum airfoil shapes, and optimize orbital trajectories.[1] The two methods studied by this thesis are particle swarm optimization (PSO) and bacterial foraging optimization (BFO), which are both biomimetic. These two stochastic and heuristic algorithms take advantage of the sharing of information between population members. Using a simple MATLAB code, both algorithms are used to determine the optimal two-impulse transfer between coplanar, circular orbits. The solutions generated by each method are compared against the analytic solution. The algorithms will be compared to determine which one achieves results that are more accurate in a shorter computational time.

Particle Swarm Optimization

Particle swarm optimization was first presented in 1995 and can be described as a swarm intelligence method.[1] The algorithm mimics the flight pattern of a flock of birds or group of insects as they search for food. Each organism is represented as a particle that denotes a possible solution.

The initial population of particles is randomly initialized along the range of possible values for each parameter. Each particle is associated with two vectors: a position vector and a velocity vector. The position vector is made up of the unknown variables being optimized, and the velocity vector determines the particle's updated position after each iteration, taking into consideration the positions of the other particles. After each iteration of the code, the algorithm records the optimum particle found thus far and then updates the particle positions. PSO terminates after a set number of iterations has been reached.

A number of factors influence the effectiveness of PSO. Increasing the size of the population and the number of iterations generally leads to a more accurate solution, but this results in increasing the computational time required. Another aspect to consider is the various stochastic weights assigned to update the position of the swarm. The velocity vector of each particle is highly dependent on these weights, and changing the values can alter how the program converges on a solution.

Bacterial Foraging Optimization

Bacterial foraging optimization is similar to PSO in that the solution is found by manipulating a population of potential solutions; however, the algorithm for BFO differs in its update and iterative processes. BFO models the process of chemotaxis, the behavior where bacteria forage for food and resources, in *E. coli* colonies.[2] Rather than analyzing the patterns of a single bacterium, the algorithm takes advantage of the communication between bacteria to bring the entire population towards the optimum solution. The BFO algorithm utilized in this paper is based on a version of the one described by Chen, Zhu, and Hu.[2] The bacteria colony is modeled by a number of individual bacterium that make up the entire population. Analogous to the particle in PSO, each bacterium is composed of an n-dimensional vector, where n is the number of parameters being optimized. The solution is evaluated by the fitness of each bacterium; the smallest fitness value is representative of the optimum solution.

Once the initial population is randomly generated, the first step of the algorithm is to simulate chemotaxis. This is the tumble and swim phase where each bacterium moves in a series of randomized steps. The bacterium takes one initial step (tumble) and evaluates its fitness. If that particular bacterium has achieved a better fitness at any point in its lifetime, the bacterium then makes a series of more steps (swims) until it reaches a new lifetime best or the set maximum number of swims has been reached.

The next step in the method is to simulate reproduction. The population is sorted by fitness value, and the half with the higher fitness values are eliminated. These bacteria are replaced with copies of the better half, thereby maintaining a constant population size. The final step is important in reducing the possibility of stagnation, or converging too early. Each bacterium has a small chance of being eliminated at the end of each generation. Bacteria that are deleted are replaced by a new, randomly initialized bacterium. The process then repeats for the number of specified generations. As the generations progress, the surviving bacteria converge on the optimal solution to the problem.

Chapter 2

Statement of the Problem

In this problem, a spacecraft in an initial circular orbit of radius R_1 transfers to a second circular orbit with radius R_2 through two impulsive changes in velocity. Between the impulses is a coasting arc where to fuel is spent. The goal is to determine the optimal magnitudes of the two impulses in order to minimize the amount of propellant used. One of the key assumptions of this method is to approximate the thrust as impulsive (occurring instantaneously). This is an acceptable and accurate assumption for this problem as most rockets are capable of very high thrust that occur over short times compared to the coasting arc[3].This chapter explains the equations for solving this two-impulse problem both analytically through the Hohmann transfer and heuristically through PSO and BFO.

Hohmann Transfer

The Hohmann transfer gives the solution to the two-impulse transfer with the smallest total Δv, which corresponds to the least amount of propellant used. The optimal solution to this problem has been analytically proven to be the Hohmann transfer for cases where $1 < {}^{R_2}/_{R_1} < 11.94$.[3,4] Figure 1 illustrates the two circular orbits and the elliptical coasting arc between them.

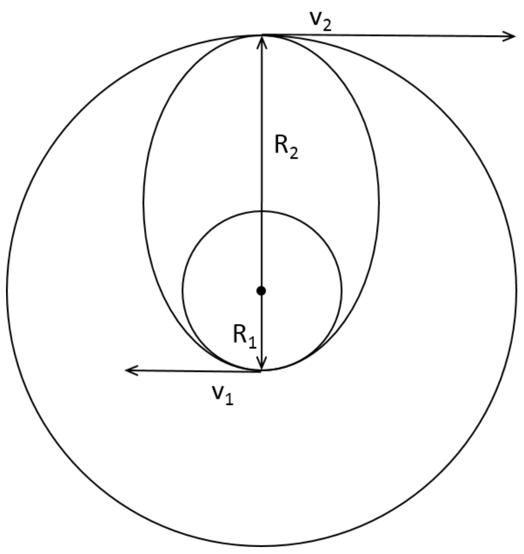

Figure 1 Transfer Ellipse between Concentric Circular Orbits

The transfer orbit is a semi-elliptical orbit that represents the optimum transfer between two circular, concentric, and coplanar orbits. A spacecraft on the initial circular orbit moves onto the periapsis of the transfer ellipse through an impulsive change in velocity, and then at apoapsis another impulsive change in velocity brings the spacecraft to the second circular orbit by decreasing its velocity.[3] The problem solved for this paper was specifically that of transfer from a low earth orbit (7,000 kilometers) to geostationary orbit (42,164.2 km), which define the initial and final radii respectively.

The impulsive changes in velocity that are trying to be minimized are defined by

$$\begin{aligned} \Delta v_1 &= v_p - v_1 \\ \Delta v_2 &= v_2 - v_a \end{aligned} \tag{1}$$

where v_p is the velocity at periapsis on the Hohmann ellipse, v_a is the velocity at apoapsis on the Hohmann ellipse, and v_1 and v_2 are the velocities on the first and second circular orbits respectively. The velocity on a circular orbit is defined by the equation

$$v = \sqrt{\frac{\mu_B}{r}} \tag{2}$$

where r is the radius of the circular orbit. The velocities at periapsis and apoapsis on the Hohmann ellipse can be found using

$$\begin{aligned} v_p &= \sqrt{2(\varepsilon_H + \frac{\mu_B}{r_p})} \\ v_a &= \sqrt{2(\varepsilon_H + \frac{\mu_B}{r_a})} \end{aligned} \tag{3}$$

where ε_H is the energy of the Hohmann transfer ellipse, defined by

$$\varepsilon_H = -\frac{\mu_B}{2a_H} \tag{4}$$

In Eq. (4), a_H is the semimajor axis of the Hohmann ellipse, which is simply the average of the initial and final radii.

Stochastic Methods

PSO and BFO both will solve the same the same two-impulse transfer problem through stochastic means. The objective or fitness function J to be minimized by PSO and BFO is given as

$$J = \Delta v_1 + \Delta v_2 \tag{5}$$

Equation (5) represents the total impulsive velocity change, the same value minimized by the Hohmann transfer. The two unknown parameters that will vary across particles and bacterium are the first velocity impulse Δv_1 and the angular displacement of the first impulse δ_1.[1] In order to calculate and minimize the objective function, Δv_2 must be calculated using the boundary conditions of the problem and the constraints on a feasible trajectory. The constraints are that the orbit must be elliptic ($a > 0$) and the apoapse radius must at least be as large as the final radius ($a[1+e] \geq R_2$).[1] The initial conditions for the problem are

$$v_r(t_0^-) = 0$$
$$v_\theta(t_0^-) = \sqrt{\frac{\mu_B}{R_1}} \tag{6}$$
$$r(t_0^-) = R_1$$

while the final conditions are given by

$$v_r\left(t_f^+\right) = 0$$
$$v_\theta\left(t_f^+\right) = \sqrt{\frac{\mu_B}{R_2}} \tag{7}$$
$$r\left(t_f^+\right) = R_2$$

Immediately after the first impulse is complete, the radial and horizontal components of the velocity become

$$\begin{aligned} v_r(t_0^+) &= v_r(t_0^-) + \Delta v_1 sin\delta_1 \\ v_\theta(t_0^+) &= v_\theta(t_0^-) + \Delta v_1 cos\delta_1 \end{aligned} \tag{8}$$

while the radius remains R_1. Between the first and second impulses, the trajectory of the spacecraft is elliptical and can therefore be described using Keplerian relationships. The following equations describe the state of the spacecraft immediately after the first impulse:

$$a = \frac{\mu_B r(t_0^+)}{2\mu_B - r(t_0^+)[v_r^2(t_0^+) + v_\theta^2(t_0^+)]} \tag{9}$$

$$e = \sqrt{1 - \frac{r^2(t_0^+)v_\theta^2(t_0^+)}{\mu_B a}} \tag{10}$$

$$\cos[f(t_0^+)] = \frac{v_\theta(t_0^+)}{e}\sqrt{\frac{a(1-e^2)}{\mu_B}} - \frac{1}{e} \tag{11}$$

$$\sin[f(t_0^+)] = \frac{v_r(t_0^+)}{e}\sqrt{\frac{a(1-e^2)}{\mu_B}} \tag{12}$$

The spacecraft coasts along this elliptical arc until the radius reaches a value of R_2. The true anomaly immediately before Δv_2 occurs is

$$f\left(t_f^-\right) = \arccos\left[\frac{a(1-e^2) - R_2}{R_2 e}\right] \tag{13}$$

Combining and rearranging Eqs. (11) – (13) gives expressions for the radial and horizontal velocities directly before the second impulse:

$$\begin{aligned} v_r\left(t_f^-\right) &= \sqrt{\frac{\mu_B}{a(1-e^2)}}\, e \sin[f\left(t_f^-\right)] \\ v_\theta\left(t_f^-\right) &= \sqrt{\frac{\mu_B}{a(1-e^2)}}\{1 + e\cos[f\left(t_f^-\right)]\} \end{aligned} \tag{14}$$

The final conditions (Eq. (7)) and Eq. (8) evaluated at t_f^- combine to form an expression for the magnitude of the second velocity impulse:

$$\Delta v_2 = \sqrt{v_r^2\left(t_f^-\right) + \left[\sqrt{\frac{\mu_B}{R_2}} - v_\theta\left(t_f^-\right)\right]^2} \tag{15}$$

Substituting Eq. (15) into Eq. (5) solves the objective function. Particle swarm optimization and bacterial foraging optimization find the value of this objective function for each particle or bacterium and iterate to find the global minimum value.

It is necessary to create a set of normalized units in order to solve the problem. Pontani and Conway[1] use a method of normalization in which the initial radius is set to a value called the distance unit (DU). The other normalized unit, called the time unit (TU), is a value such that $\mu_B = \frac{DU^3}{TU^2}$. For all optimization cases of the two-impulse circular orbit transfer, the parameters are bound by

$$0\ \frac{DU}{TU} \leq \Delta v_1 \leq 1\ \frac{DU}{TU}$$
$$-\pi \leq \delta_1 \leq \pi \tag{16}$$

Particle Swarm Optimization Equations

In each iteration of the algorithm, J is calculated for each particle. Of those J values, if the lowest value is lower than any J from any other iteration, that value is assigned to the variable G_{best}. For each particle, a record is kept of the best set of values that particular particle has achieved over the course of all the iterations. That value is stored in the variable P_{best}. The information-sharing portion of this algorithm is found in updating the particle's position. Each particle is updated based on P_{best},G_{best}, and the particle's previous position and velocity. The equation for the updated velocity is

$$V_{new} = c_I V + c_c(P_{best} - P) + c_s(G_{best} - P) \tag{17}$$

The three coefficients c_I, c_c, and c_s are the stochastic weights that affect how the velocity changes.[1] Each coefficient is associated with its own component of the new velocity. The inertial component, $c_I V$, is proportional to the previous velocity of the particle and works to prevent the particle from leaving the trajectory it is currently on. The cognitive component, $c_c(P_{best} - P)$, moves the particle in the direction of the best position encountered by that particle. The last term is the social component, $c_s(G_{best} - P)$, and this term moves the particle towards the best position found by the entire swarm. The three coefficients are defined as

$$c_I = \frac{1 + r_1(0,1)}{2}$$
$$c_c = 1.49445\, r_2(0,1) \tag{18}$$
$$c_s = 1.49445\, r_3(0,1)$$

where $r_1(0,1)$, $r_2(0,1)$, and $r_3(0,1)$ are random values along the domain [0,1]. The new position of the particle is then simply

$$P_{new} = V_{new} + P \tag{19}$$

In order to handle the case of constrained optimization, Pontani and Conway[1] give methods for both equality and inequality constraints. For an inequality constraint, if either of the inequalities $a > 0$ or $a(1 + e) \geq R_2$ are violated, an arbitrarily large value will be assigned to J for that particular particle. Additionally, the inertial component of the updated velocity must be set to zero so that the influence from other particles brings the particle back into the reasonable region of the search space.[1]

Bacterial Foraging Optimization Equations

After initializing the population of bacteria, a random unit vector, u_i, is generated. The bacterium tumbles and swims in the direction of u_i during the chemotaxis phase. The magnitude of movement in the u_i direction is the swim-step size C_0. This value is very problem dependent, and it should be chosen to be close to the same order of magnitude as the parameters being adjusted. Larger C_0 values tend to require a smaller number of steps to reach the solution, but it may lead to greater error. However, bacteria with a smaller C_0 achieve greater precision, but this comes at the cost of occasionally failing to reach the minimum before all generations have passed[2]. The bacterium parameters are updated using the equation

$$B = B + C_0 u_i \tag{20}$$

which represents a tumble. The fitness function is then evaluated for these new B values. Equation (20) is also used during each swim step.

Chapter 3

Method

This chapter is a description of the two heuristic algorithms used to optimize the two-impulse transfer between two circular orbits. These methods are compared against each other and the analytic solution that is found through the Hohmann transfer. The following sections are a detailed description of each of the five m-files used for PSO and the two m-files used for BFO. The actual MATLAB code for the PSO method is found in Appendix A, while the BFO code is found in Appendix B.

Particle Swarm Optimization Codes

PSOTest_adaptive (Main)

PSOTest_adaptive is the main function for this algorithm. The first portion of the code defines the global variables used across the five m-files. This section can be edited depending on the specific circumstances of the problem. For this case, two elements are needed per particle $(\Delta v_1, \delta_1)$, and 50 particles are used over 200 iterations (initially). The variable R_1 is set to be 7000 km, and R_2 is set to 42,164.2 km. The initial and final conditions as defined in Eq. (6) and Eq. (7) are also input at this point.

The next section sets the lower and upper bounds of the parameters for this problem. For every case, the optimal values of Δv_1 fall between 0 and 1 DU/TU while the optimal values for

δ_1 are between $-\pi$ and π. Optimization problems with more than two parameters would have an element for each parameter in the BLp and BUp arrays.

The initial swarm population is generated randomly along the bounds set up by the variables BLp and BUp. For each particle, multiplying the range of possible values by a random value between 0 and 1 and adding that to the lower bound generates a random value along the domain.

BUv and BLv are the variables that define the maximum bounds on velocity. These too are based on the bounds of the particles themselves. JBest (the best position visited by a specific particle) and GG (the best position visited by any particle) are initialized to infinity so that any values the initial particles generate in the fitness function will become the new best values.

The next portion of the main function is to call the other m-files and evaluate the fitness function, find the new JBest and GG values, update the velocity function, and then finally update the particle position so that the process can start again. At the end of each iteration, the current global best value of the objective function is recorded in an array, GGstar.

The final portion of the main calculates the analytical (Hohmann) solution and displays relevant data. The Hohmann solution is found using Eqs. (1) – (4) in order to compare to the heuristic solution found by the PSO algorithm. A few values from the PSO algorithm are then displayed in order to check accuracy. Finally, the percent error at each iterative step is displayed as a graph.

EvalJ

EvalJ is the m-file where the fitness function is evaluated. The fitness function in this problem is the total Δv of the spacecraft, found by simply adding Δv_1 and Δv_2. The first impulse is one of the parameters being adjusted by the algorithm, but the second impulse must be calculated using the equations from Chapter 2. For each particle i, vr_t0_plus and vtheta_t0_plus are calculated using Eqs. (8). The semimajor axis and the eccentricity are then calculated using Eqs. (9) and (10). It is at this point that if either of the inequality constraints are violated, the fitness function is set to an arbitrarily large number and the particle's velocity is set to zero. This helps prevent stagnation and keeps all particles within the realm of possible values.

The next part is to calculate f_tf_minus (true anomaly prior to the second impulse), vr_tf_minus (radial velocity prior to the second impulse), vtheta_tf_minus (horizontal velocity prior to the second impulse), and delta_v_2 using Eqs. (13) – (15). The sum of Δv_1 and Δv_2 is then the fitness function trying to be minimized.

EvalPGBest

EvalPGBest is the second function to be called by the main. Its purpose is to determine the best (lowest J value) position visited by each particle and record it. For each particle i, the function compares the current J value to $JBest$, and if $J < JBest$, then both the particle parameters P and the fitness value are recorded. Additionally, the J value of each particle is compared against the global best fitness value, GG. If $J < GG$, then GG takes on the that value of J and the particle parameters are recorded in the variable GBest.

UpdateV

UpdateV changes the velocity of each particle based on the randomized accelerator coefficients. The three coefficients are determined by Eqs. (18). For each particle i, the new velocity is calculated according to Eq. (17). After calculating the new velocity value, it must be tested against the bounds that were set up earlier in the code. If the velocity of a particle is below the lower bound BLv, then the velocity is changed to equal the lower bound. Similarly, if the velocity is higher than the upper bound BUv, then the velocity is changed to be equal to the upper bound. This ensures that the particle remains within the domain of possible solutions.

UpdateP

The final portion of the PSO algorithm is UpdateP, a function that updates the position of each particle between iterations. The new position of each particle is found simply through Eq. (19). Similarly to the UpdateV m-file, each particle's position is checked against the upper and lower bounds defined earlier in the code. If either bound is exceeded, the particle's position is redefined to equal the boundary condition, and the velocity is set to zero.

Bacterial Foraging Optimization Codes

BFO_test (Main)

BFO_test is the main function for the bacterial foraging optimization method. The first part of the code defines the BFO variables used in the algorithm and the problem specific variables. Like with the PSO algorithm, the number of elements being optimized is two. The initial population size was 30 bacteria, but that value is varied throughout the results. The number of swim steps is set to 100 and the number of generations is set to 50. The step size during each swim was set to 0.001 based on a series of trials. Adjusting the step size can have drastic effects on the consistency of the solutions. The variables used for solving the two-impulse problem in the PSO code are used in this code as well.

The upper boundary of the parameters is defined by the variable bU, and the lower boundary of the parameters is defined by bL. The array B contains a row for each bacterium and four columns, two more than the number of elements being optimized. These two extra columns are for storing the best fitness value that bacterium has experienced and the current fitness value, J. The first two columns of each bacterium are randomly initialized in the same way as PSO by multiplying the difference between the upper and lower bounds by a random number and adding that to the lower bound. The last two columns are initialized differently; the first is set to values of infinity, and the second is set to all zeros. Setting the initial best fitness values to infinity ensures that the first iteration will produce new values for the best fitness.

The next section of the code begins the foraging algorithm. For each bacterium i, a vector called Delta is randomly generated. This vector has a value for each element being optimized. It is normalized and becomes the unit vector u_i, denoted by the variable u in the code. The ith bacterium's elements are then increased by the swim step size C_0 times the unit vector u_i, and the fitness function for that bacteria is then calculated and stored in the last column of the array B. The swim step counter is set to zero. While both the swim step counter is less than the total allowable swim steps (100) and the current fitness is less than the previous best fitness value, the algorithm first assigns the current fitness to be the new best value. Then, the bacterium takes another step in the $C_0 * u_i$ direction (this is the swim step), and the fitness function is evaluated again. The swim step counter increases by one, and the bacterium makes another step if it still satisfies the conditions of the swim step. Once it has either taken the maximum number of swim steps or the new fitness value is greater than the previous best, the algorithm proceeds to the next bacterium.

After all bacteria have gone through chemotaxis, the code rearranges the array B in order of increasing J value. The bottom half of the array (the half with the worse fitness values) is replaced by copies of the top half of the array. B_save stores a copy of the bacteria.

The final part of this algorithm randomly destroys approximately 25% of the bacteria population and reassigns random initial values. This step is to ensure the code does not stagnate and converge on a false optimum. For every bacterium i, a random value between 0 and 1 is generated and checked against the elimination and dispersion probability of 0.25. If the value falls below 0.25, that particular bacterium is replaced by a randomly generated bacterium. After this, the code returns to the chemotaxis stage until all iterations have been completed.

Once all iterations are finished, this code outputs data in a similar way to the PSO code. The Hohmann transfer is analytically determined, and then the error is graphed over the lifetime of generations.

EvalJ_BFO

EvalJ_BFO is the only other code associated with the bacterial foraging optimization algorithm besides the main. This function is nearly identical to the EvalJ function from particle swarm optimization. Each variable of the two-impulse transfer problem is computed using the equations from Chapter 2. Just as with PSO, the fitness function J is set to an arbitrarily high value if the inequality conditions are violated. The fitness for each bacterium is computed by adding the first element of the bacterium with delta_v_2. The value of J is then passed back to the main through the variable cost.

Chapter 4

Results

This chapter details the tests that were run with the codes and the associated calculations and analysis. In order to maintain the accuracy of the results, all trials were run without restarting MATLAB. Unless otherwise noted, each data point in the following tables and figures represents one trial run of the algorithm.

The exact solution was determined in order to evaluate the effectiveness of the heuristic algorithms. Because $R_1 = 7000$ km and $R_2 = 42164.2$ km, the value of ${R_1}/{R_2}$ is 6.023, which falls within the acceptable bounds for the validity of the Hohmann transfer solution. The analytic results are shown in Table 1.

Table 1 Hohmann Transfer Analytic Solution

Parameter	Value
$\boldsymbol{J^*}$ $(\Delta \boldsymbol{v_1} + \Delta \boldsymbol{v_2})$	3.77073 km/s
$\Delta \boldsymbol{v_1}$	2.33680 km/s
$\Delta \boldsymbol{v_2}$	1.43393 km/s
$\boldsymbol{\delta_1}$	0 rad
$\boldsymbol{\delta_2}$	π rad

The first impulse is found to be larger than the second impulse. The first impulse is directed along the tangent of the initial orbit, and the second impulse is still tangent to the ellipse but in the opposite direction of motion. This means that the spacecraft slows its velocity to transfer to the second circular orbit, but the total Δv calculated is still the sum of the two impulses.

The first test run on the algorithms was to compare the accuracy and time to completion of PSO and BFO while holding the parameters of the algorithms themselves constant and equal.

Both codes were run with 50 particles or bacteria over 200 iterations/generations, generating the results shown in Table 2. The two codes were run five times each to compare inaccuracies.

Table 2 PSO and BFO at Same Run Conditions

Trial	Δv_1 **(km/s)**	Δv_2 **(km/s)**	δ_1 **(rad)**	δ_2 **(rad)**	**Time (seconds)**
PSO					
1	2.33681	1.43393	4.30567E-03	-2.2473E-05	0.1944
2	2.33680	1.43393	-6.3447E-06	-3.34394E-06	0.1914
3	2.33681	1.43393	3.5528E-03	-3.27479E-05	0.1905
4	2.33680	1.43393	1.217E-03	-1.471E-04	0.1880
5	2.33680	1.43393	-1.407E-03	-1.44613E-05	0.1894
Avg	2.33680	1.43393	1.5324E-03	-4.4025E-05	0.1907
MAPE	1.71E-04 %	7.0E-02 %	4.8779E-02 %	1.4014E-03 %	-
BFO					
1	2.35414	1.45468	0.138733	-0.167653	1.4547
2	2.34842	1.49038	0.076278	-0.279049	1.3990
3	2.37203	1.45575	0.197694	-0.167999	1.4904
4	2.36592	1.45297	-0.17795	-0.157409	1.1137
5	2.34177	1.46987	0.021756	-0.224384	1.3287
Avg	2.35646	1.46987	0.051302	-0.199299	1.3573
MAPE	0.8412 %	2.148 %	1.633 %	6.344 %	-

The PSO algorithm generated results that were very close to the analytically predicted values. The values for δ_2 appear to be nearly zero because numerical calculations of inverse trigonometric functions only give the principal value; a quadrant correction of π radians places the direction in the correctly predicted orientation. The parameter with the worst mean absolute percent error (MAPE) was δ_1 with a value of $4.88 * 10^{-2}$ %, and the most accurate parameter was Δv_1 with only $1.71 * 10^{-4}$ % MAPE. All of the parameters were optimized in less than 0.2 seconds, with an average run length of 0.1907 seconds.

The bacterial foraging algorithm quite noticeably does not converge as quickly or as accurately as PSO in this case. The MAPE was several orders of magnitude greater than in the PSO method, with the most error being calculated for δ_2 (6.344 %). BFO also takes up to 8 times as long to run as particle swarm optimization. While for this situation that only entails an increase of about 1.3 seconds, other optimization problems can take much longer to run using these same algorithms. Therefore, an eight-fold increase in time is a significant factor to consider when using BFO.

The next thing to look at when analyzing these algorithms is how quickly they converge on a solution. Figure 2 shows the percent error between the current best fitness J and the analytic best fitness J^* as the iterations progress in the particle swarm algorithm. The data show that the code gets to a solution within 0.1 % accuracy in only about 10 iterations. The solution slowly becomes more accurate over the next 120 iterations, until the particles finally converge on a value that is less than 0.002 % off.

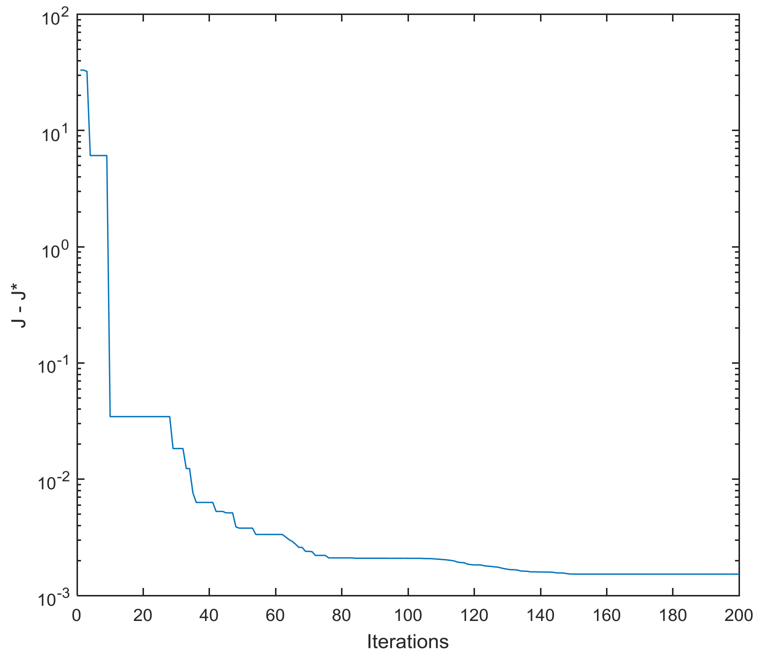

Figure 2 PSO Error vs. Iterations (50 particles, 200 iterations)

Figure 3 below shows the BFO error as the generations progress. The graph shows how there is much less dynamic change over the lifetime of the bacteria as compared to PSO. This slower rate of change could be a leading cause of why PSO converges to a more accurate solution in a quicker amount of time. There are two points on Fig. 3 where the error decreases and then increases again. This is caused by the elimination step in the foraging algorithm. Occasionally, bacteria with the best fitness will be eliminated, slowing the progress of the entire colony. However, this process is still necessary in order to prevent stagnation from occurring. More research is needed to see if there could be a way to modify the BFO method to produce results as quickly and accurately as particle swarm optimization.

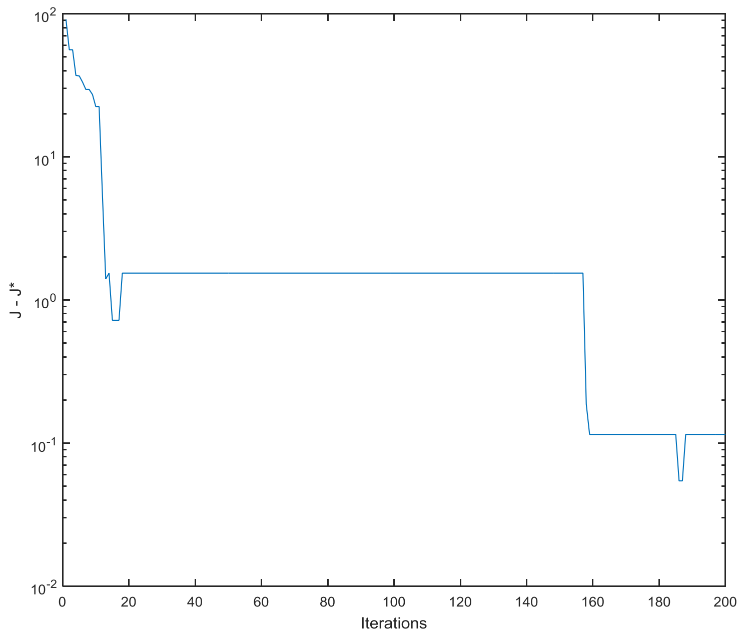

Figure 3 BFO Error vs. Iterations (50 bacterium, 200 generations)

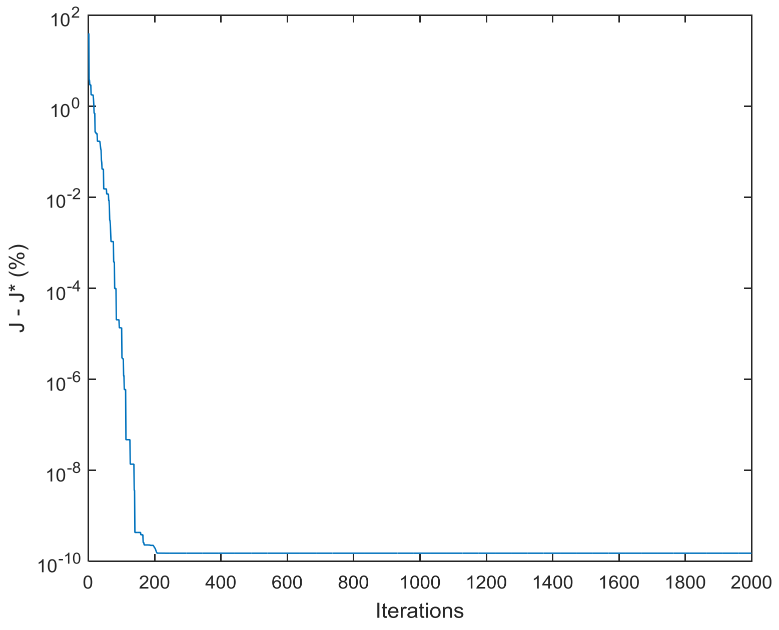

Figure 4 PSO Error vs. Iterations (50 particles, 2000 iterations)

Figure 4 once again shows the error in the PSO algorithm as the iterations progress, but this time the code was run for 2000 iterations. This trial produced a final error several orders of magnitude less than what was found with only 200 iterations. However, the minimal error was still found in approximately 200 steps, which could imply that this improved result is due to the stochastic nature of this algorithm rather than improved general performance with more iterations. More trial runs are needed in order to evaluate exactly how the algorithm parameters affect the results.

The following table shows several different run conditions for PSO. Each data point in the table is an average of five different trials. The top value in each cell is the average converged fitness function, the second value is the mean absolute percent error of J^*, the third value is the standard deviation of the five trials, and the fourth number is the average time to completion. The MAPE indicates how accurate each trial is, while the standard deviation is an indication of how precise the trials were.

Table 3 PSO Varying Particles and Iterations

		Iterations		
		10	**200**	**2000**
Particles	**10**	5.61278 km/s 48.9 % 1.990 0.03479 seconds	3.79362 km/s 0.607 % 0.02647 0.08963 seconds	3.84107 km/s 1.865 % 0.0762 0.5069 seconds
	50	3.8999 km/s 3.43 % 0.1289 0.05937 seconds	3.77073 km/s 0.000159 % 4.90E-06 0.1907 seconds	3.77094 km/s 0.005304 % 0.000296 1.1779 seconds
	500	3.78016 km/s 0.249 % 0.005635 0.09001 seconds	3.77073 km/s 2.64E-13 % 1.6E-14 1.225 seconds	3.77073 km/s 1.38E-13 % 6.6E-13 10.14 seconds

There are a few interesting trends that come out of Table 3. It appears that increasing the number of particles decreases the standard deviation of the dataset. This implies that increasing the number of particles decreases the effect of randomness on the converged solution and leads to solutions that are more precise. Not surprisingly, increasing either the number of particles or iterations increases the time to completion of the algorithm. At very low numbers of particles and iterations, the accuracy of a single trial cannot be guaranteed. The stochastic effects of the algorithm introduce variations that cannot be reduced down with such a small population. Once the parameters reach at least 500 particles and 200 iterations, the results become accurate to within very small percentages. Additionally, increasing the number of particles or iterations further than that only serves to increase computation time and does not significantly increase the accuracy. This is reinforced by Fig. 4 that shows the algorithm converging to a solution within the first 200 iterations.

Similarly to Table 3, Table 4 below shows the results of many trial runs for the BFO method, ranging over several bacteria and generation levels. Each data point in Table 4 is an average of five trials.

Table 4 BFO Varying Bacteria and Generations

		Generations		
		10	**200**	**2000**
Bacteria	**10**	3.16529 km/s 16.06 % 2.703 0.05395 seconds	3.96382 km/s 5.12 % 0.2373 0.2546 seconds	3.80284 0.8512 % 0.03568 1.3463 seconds
	50	4.10502 km/s 8.8650 % 0.3569 0.1838 seconds	3.90926 km/s 3.6737 % 0.15451 1.5322 seconds	3.77155 km/s 0.4333 % 0.02020 5.9580 seconds
	500	3.82441 km/s 1.4234 % 0.03121 1.8972 seconds	3.78116 km/s 0.27639 % 0.011926 8.2050 seconds	3.78835 km/s 0.4670 % 0.0108 74.54 seconds

Comparing Tables 3 and 4, it becomes clear that the BFO algorithm uses more computer resources and takes a longer time to complete than PSO. Like with the PSO algorithm, increasing either the number of bacteria or the number of generations both decreases error and increases computational time. In addition, the standard of deviation is also inversely proportional to bacteria population and number of generations. Analyzing the first extreme case, 10 bacteria in 10 generations, it becomes apparent that BFO requires either more generations or more bacteria in order to successfully converge on a solution. The standard deviation is at least an order of magnitude larger than most other cases. In some individual trials, the converged solution was as much as 66% off. In the other extreme case, the algorithm successfully was able to converge to a solution within less than half a percent of error with a small standard deviation of 0.0108. However, this came at the cost of an almost ten-fold increase in computational time. Using either fewer bacteria or fewer generations produces almost the same results with the added benefit of drastically speeding up the computational process.

Both particle swarm optimization and bacterial foraging optimization are capable of converging without finding the optimal solution. While increasing the population and/or iteration number (provided there is a sufficient minimum population) will decrease the likelihood of stagnation, it is always a possibility because of the random nature of these stochastic methods. Additionally, either algorithm can vary by several orders of magnitude in its accuracy for two different trials ran at the same parameters (while still being within a very small percentage of the analytical solution). In order to account for the randomness of these two algorithms, it is recommended that as many trials as possible are run in order to average all of the results.

Chapter 5

Conclusion and Recommendations for Future Work

Optimizing orbit trajectories can be very computationally difficult using rigorous mathematical models. Biomimetic algorithms that use stochastic methods have been shown to be able to find the solutions to these optimization problems. While particle swarm optimization and bacterial foraging optimization both are capable of converging to a solution within 1% error, the PSO algorithm converges in less computational time and to within a smaller error. In cases where the two algorithms complete in the same amount of time, the PSO solution is often 10 orders of magnitude more accurate than the BFO solution. Both algorithms can occasionally stagnate and converge on the non-optimum value; however, for the same number of particles and iterations, PSO is much less likely to stagnate.

Further research should apply these algorithms to other optimization problems and once again compare their effectiveness. It is possible that the algorithms could have different convergence behaviors when solving different problems. In the future, the bacterial foraging algorithm could be modified to produce results that are more accurate. The swim step size should be varied and studied to see its effect on the converged solution. For finding the optimal two-impulse transfer between circular orbits, PSO is the preferred optimization method.

Appendix A

Particle Swarm Optimization Source Code

PSOTest_adaptive (Main)

```
%% PSO With Variable Accelerator Coefficients

clc;
clear all;

tic

format longg

% Define global variables for PSO
global P J JBest PBest GG GBest N_particles N_elements V BLv BUv BLp BUp
global R_1 R_2 delta_2 delta_v_2 vr_t0_minus vtheta_t0_minus r_t0_minus
vr_tf_plus
global vtheta_tf_plus r_tf_plus mu_B vr_t0_plus vtheta_t0_plus a e f_tf_minus
vr_tf_minus
global vtheta_tf_minus delta_2_sin
global N_iterations

% Initialize the particles in the swarm
N_particles = 50;
N_elements = 2;
N_iterations = 2000;

% set problem variable values
R_1 = 7000; %initial radius, km
R_2 = 42164.2; %final radius, km
mu_B = 3.98600e5; %standard gravitational parameter, km^3*s^-2
DU = 7000; %distance unit, equal to initial radius
TU = sqrt(DU^3/mu_B); %time unit, such that mu_B = DU^3/TU^2

% initial and final conditions
vr_t0_minus = 0;
vtheta_t0_minus = sqrt(mu_B/R_1);
r_t0_minus = R_1;
vr_tf_plus = 0;
vtheta_tf_plus = sqrt(mu_B/R_2);
r_tf_plus = R_2;

%% set lower and upper bounds on unknowns (particle elements)
BLp = [0, -pi]; %lower bounds, one element for each parameter
```

```
BUp = [DU/TU, pi]; %upper bounds, one element for each parameter

%% create random initial population

for i = 1:N_particles
    P(i,1) = BLp(1) + (BUp(1)-BLp(1))*rand;
    P(i,2) = BLp(2) + (BUp(2)-BLp(2))*rand;
end

PBest = zeros(N_particles,N_elements);
J = zeros(N_particles); JBest = zeros(N_particles);
V = zeros(N_particles, N_elements);

%% determine velocity bounds
BUv = BUp - BLp;
BLv = -BUv;
for i = 1:N_particles
    JBest(i) = inf;
end
GG = inf;

%% call all functions
for j = 1:N_iterations
    EvalJ;
    EvalPGBest;
    UpdateV (j);
    UpdateP;
    GGstar(j) = GG;
end

toc

%% Analytical solution (Hohmann Transfer)
a_H = (R_1+R_2)/2;
energy = -mu_B/(2*a_H);
v_1 = sqrt(mu_B/R_1);
v_2 = sqrt(mu_B/R_2);
v_p = sqrt(2*(energy+mu_B/R_1));
v_a = sqrt(2*(energy+mu_B/R_2));

dv1 = v_p - v_1;
dv2 = v_2 - v_a;

Jstar = dv1 + dv2; %Jstar is the analytic value calculated by Hohmann

%% output necessary data

GBest

delta_v_2
delta_2_sin

%plot of error vs iterations
```

```
x = (1:N_iterations);
y = abs(GGstar - Jstar)/Jstar*100; %percent difference between calculated
optimum and actual
semilogy(x,y)
xlabel('Iterations')
ylabel('J - J* (%)')
```

UpdateP

```
function UpdateP()
%% UpdateP updates the position vector
%%
global P N_particles N_elements V BLp BUp
for i =1:N_particles
    P(i,:) = P(i,:) + V(i,:); %updates the position
    for k = 1:N_elements
        if P(i,k) < BLp(k) %prevents the particle from escaping the bounds
            P(i,k) = BLp(k);
            V(i,k) = 0;
        end
        if P(i,k) > BUp(k)
            P(i,k) = BUp(k);
            V(i,k) = 0;
        end
    end
end
```

UpdateV

```
function UpdateV(j)
%%
%% UpdateV updates the velocity vector V
%% Variable accelerator coeffs.
%%
global P J PBest GBest N_particles N_elements V BLv BUv
global N_iterations

c_I = (1 + rand)/2; %inertial coefficient
c_C = 1.49445*rand; %cognitive coefficient
c_S = 1.49445*rand; %social coefficient

for i =1:N_particles
    V(i,:) = c_I*V(i,:) + c_C*(PBest(i,:) - P(i,:)) + c_S*(GBest - P(i,:));
    for k = 1:N_elements
        if V(i,k) < BLv(k)
```

```
            V(i,k) = BLv(k);
        end
        if V(i,k) > BUv(k)
            V(i,k) = BUv(k);
        end
    end
end
```

EvalJ

```
function EvalJ()
%% EvalJ evaluates J for each particle in current iteration
global P J V N_particles
global delta_2_sin R_1 R_2 delta_2 delta_v_2 vr_t0_minus vtheta_t0_minus
r_t0_minus vr_tf_plus vtheta_tf_plus r_tf_plus
global mu_B vr_t0_plus vtheta_t0_plus a e f_tf_minus vr_tf_minus
vtheta_tf_minus

for i = 1:N_particles
    vr_t0_plus = vr_t0_minus + P(i,1)*sin(P(i,2)); %
    vtheta_t0_plus = vtheta_t0_minus + P(i,1)*cos(P(i,2));
    a = (mu_B*r_t0_minus)/(2*mu_B - r_t0_minus*(vr_t0_plus^2 +
vtheta_t0_plus^2));
    e = sqrt(1 - (r_t0_minus^2*vtheta_t0_plus^2)/(mu_B*a));

    if a < 0 %if the particle does not meet the constraint of the problem,
this sets the J value high
        J(i) = 1e12;
        V(i,:) = 0;
    elseif a*(1+e)-R_2 < 0
        J(i) = 1e12;
        V(i,:) = 0;
    else

        f_tf_minus = acos((a*(1-e^2)-R_2)/(R_2*e));
        vr_tf_minus = sqrt(mu_B/(a*(1-e^2)))*e*sin(f_tf_minus);
        vtheta_tf_minus = sqrt(mu_B/(a*(1-e^2)))*(1 + e*cos(f_tf_minus));
        delta_v_2 = sqrt(vr_tf_minus^2 + (sqrt(mu_B/R_2) -
vtheta_tf_minus)^2);

        delta_2_sin = asin(-vr_tf_minus/delta_v_2);

        J(i) = P(i,1) + delta_v_2;
    end

end
```

EvalPGBest

```
function EvalPGBest()
%% EvalP&GBest determines the best position visited by particle i up through
%%   the current iteration)
global P J JBest PBest GG GBest N_particles
for i = 1:N_particles
    if J(i) < JBest(i)
        PBest(i,:) = P(i,:);
        JBest(i) = J(i);
    end
end
for i = 1:N_particles
    if J(i) < GG
       GG = J(i)
       GBest = P(i,:);
    end
end
```

Appendix B

Bacterial Foraging Optimization Source Code

BFO_test (Main)

```
%% BFO Test  -- tests Bacterial Foraging Optimizer
%%
%% Initialize bacteria population B and set other parameters
%
clc
clear all

tic %starts a timer to determine how long it takes for the code to run

global J R_1 R_2 delta_v_2 vr_t0_minus vtheta_t0_minus r_t0_minus vr_tf_plus
global vtheta_tf_plus r_tf_plus mu_B vr_t0_plus vtheta_t0_plus a e f_tf_minus
vr_tf_minus
global delta_2_sin vtheta_tf_minus

N_B = 500;  % number of bacteria in the population (MUST be an even number)
N_elem = 2; % number of elements in each bacterium
N_s = 100;  % number of swim steps
N_generations = 200; %number of iterative generations
C = 0.001;   % step size during swim
Ped = 0.25; % probability of elimination and dispersal

% set problem variable values
R_1 = 7000; %initial radius, km
R_2 = 42164.2; %final radius, km
mu_B = 3.986e5; %standard gravitational parameter, km^3*s^-2
DU = 7000; %distance unit, equal to initial radius
TU = 927.638; %time unit, such that mu_B = DU^3/TU^2

% initial conditions
vr_t0_minus = 0;
vtheta_t0_minus = sqrt(mu_B/R_1);
r_t0_minus = R_1;

% final conditions
vr_tf_plus = 0;
vtheta_tf_plus = sqrt(mu_B/R_2);
r_tf_plus = R_2;

bL = [0 -pi];
bU = [DU/TU pi];

B = zeros(N_B, N_elem+2);
B(:,N_elem+1) = inf*ones(N_B,1);
```

```
B_store = zeros(200,4);

for i = 1:N_B
    B(i,1) = bL(1) + (bU(1)-bL(1))*rand; %generates a random distribution of
initial values between the boundaries
    B(i,2) = bL(2) + (bU(2)-bL(2))*rand;
end

for t = 1 : N_generations
    for i = 1 : N_B
        Delta = -1*ones(1,N_elem) + 2*rand(1,N_elem);
        u = Delta/norm(Delta);
        B(i,1:N_elem) = B(i,1:N_elem) + C*u;
        B(i,N_elem+2) = EvalJ_BFO(B,i);
        m = 0; %swim step counter
        while (B(i,N_elem+2) < B(i,N_elem+1)) && (m <= N_s)
            B(i,N_elem+1) = B(i,N_elem+2);
            B(i,1:N_elem) = B(i,1:N_elem) + C*u;
            B(i,N_elem+2) = EvalJ_BFO(B,i);
            m = m+1;
        end
    end
    B = sortrows(B,N_elem+2); %arranges the array B in order of increasing
fitness value
    for i = 1:N_B/2
        B(i+N_B/2,:) = B(i,:); %replaces bottom half of array with copy of
the top half
    end
    B_store(t,:) = B(1,:);
    for i = 1 : N_B
        if rand() <= Ped
            B(i,1) = bL(1) + (bU(1)-bL(1))*rand;
            B(i,2) = bL(2) + (bU(2)-bL(2))*rand;
        end
    end
end
format long

B_store(t,:)

toc %ends the timer started by tic and displays elapsed time

%% Analytical solution (Hohmann Transfer)
a_H = (R_1+R_2)/2;
energy = -mu_B/(2*a_H);
v_1 = sqrt(mu_B/R_1);
v_2 = sqrt(mu_B/R_2);
v_p = sqrt(2*(energy+mu_B/R_1));
v_a = sqrt(2*(energy+mu_B/R_2));

dv1 = v_p - v_1;
dv2 = v_2 - v_a;

Jstar = dv1 + dv2; %Jstar is the analytic value calculated by Hohmann
```

```
%% plot
x = (1:N_generations);
y = transpose(abs(B_store(:,3) - Jstar)/Jstar*100); %percent difference
between calculated optimum and actual
semilogy(x,y)
xlabel('Iterations')
ylabel('J - J*')

delta_v_2
delta_2_sin
```

EvalJ_BFO

```
function cost = EvalJ_BFO(B,i)
global J
global R_1 R_2 delta_v_2 vr_t0_minus vtheta_t0_minus r_t0_minus vr_tf_plus
vtheta_tf_plus r_tf_plus
global delta_2_sin mu_B vr_t0_plus vtheta_t0_plus a e f_tf_minus vr_tf_minus
vtheta_tf_minus

vr_t0_plus = vr_t0_minus + B(i,1)*sin(B(i,2));
vtheta_t0_plus = vtheta_t0_minus + B(i,1)*cos(B(i,2));
a = (mu_B*r_t0_minus)/(2*mu_B - r_t0_minus*(vr_t0_plus^2 +
vtheta_t0_plus^2));
e = sqrt(1 - (r_t0_minus^2*vtheta_t0_plus^2)/(mu_B*a));

if a < 0
    J(i) = 1e12;
elseif (a*(1+e)-R_2) < 0
    J(i) = 1e12;
else
    f_tf_minus = acos((a*(1-e^2)-R_2)/(R_2*e));
    vr_tf_minus = sqrt(mu_B/(a*(1-e^2)))*e*sin(f_tf_minus);
    vtheta_tf_minus = sqrt(mu_B/(a*(1-e^2)))*(1 + e*cos(f_tf_minus));
    delta_v_2 = sqrt(vr_tf_minus^2 + (sqrt(mu_B/R_2) - vtheta_tf_minus)^2);

    delta_2_sin = asin(-vr_tf_minus/delta_v_2);

    J(i) = B(i,1) + delta_v_2;
end

cost = J(i);

end
```

BIBLIOGRAPHY

[1]Pontani, M., and Conway, B., "Particle Swarm Optimization Applied to Space Trajectories," University of Illinois at Urbana-Champaign, 2011.

[2]Chen, H., Zhu, Y., and Hu, K., "Adaptive Bacterial Foraging Optimization," Key Laboratory of Industrial Informatics, Shenyang Institute of Automation, Chinese Academy of Sciences, 2010.

[3]Wiesel, William E. *Spaceflight Dynamics*, 3rd ed. N.p.: CreateSpace Independent Platform, 2010, pp. 79-81.

[4]Prussing, E., "Simple Proof of the Global Optimality of the Hohmann Transfer," University of Illinois at Urbana-Champaign, 1992.

ACADEMIC VITA

ALEXANDER D. BOROWSKI
adb5448@psu.edu

EDUCATION

THE PENNSYLVANIA STATE UNIVERSITY — UNIVERSITY PARK, PA
Schreyer Honors College — *Class of 2016*
B.S. Aerospace Engineering

Relevant Coursework: Aerospace Technical Lab, Aerospace Structures I, Aerodynamics II, Aeronautics, Dynamics and Control, Programming with C++, Introduction to Engineering Design, Effective Speech

RELEVANT EXPERIENCE

TEXTRON SYSTEMS — UNIVERSITY PARK, PA
Test Engineer Intern — *Spring 2015*
- Collaborate with astronautics professor to calculate optimal finite-thrust spacecraft trajectories
- Study methods including Collocation with Nonlinear Programming and Particle Swarm Optimization
- Program with Matlab to calculate solutions

GENERAL ELECTRIC | TRANSPORTATION — ERIE, PA
Early Identification Internship Program, Engineering Quality — *May 2014 – August 2014*
- Managed several audit processes of customer requirements flow and taught others to use the same processes
- Worked within a cross-functional team to comb through over 5,000 unmaintained database lines and bring them to current standards
- Created, updated and managed documentation of clean up efforts
- Identified over 200 Tier 4 engine requirements that were previously untraceable

ACTIVITIES

CLUB SWIMMING — UNIVERSITY PARK, PA
Competitive Member — *October 2012 – Present*
- Mentor new swimmers on technique, time management, and Penn State life
- Compete at an annual national competition in Atlanta, GA

PENN STATE IFC/PANHELLENIC DANCE MARATHON — UNIVERSITY PARK, PA
Operations Committee Member — *October 2012 – February 2014*
- Worked with others to construct, maintain, and tear down the event center that hosted 15,000+ attendees
- Taught and led new members in performing their duties during the event weekend

SKILLS AND INTERESTS

Microsoft Office	Extensive use during internship at GE Transportation and school coursework
IBM Rational DOORS	Database program used daily during internship
Matlab, C++ and SolidWorks	Learned through programming class and applied during aerospace coursework

HONORS & AWARDS

President's Freshman Award & President's Sparks Award
- Received for maintaining a 4.0 GPA Freshman and Sophomore year respectively

Leonhard Scholarship Recipient
- $5,000 scholarship awarded for outstanding academics in Aerospace Engineering (2014)