THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF AEROSPACE ENGINEERING

INVESTIGATION OF INITIALIZATION METHODS FOR PARTICLE SWARM OPTIMIZATION OF FINITE-THRUST ORBITAL TRANSFERS

MATTHEW HONEYCHUCK
FALL 2014

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

Reviewed and approved* by the following:

Robert G. Melton
Professor of Aerospace Engineering
Director of Undergraduate Studies
Thesis Supervisor and Honors Adviser

David B. Spencer
Professor of Aerospace Engineering
Faculty Reader

George A. Lesieutre
Professor and Head of the Department of Aerospace Engineering
Faculty Reader

* Signatures are on file in the Schreyer Honors College.

ABSTRACT

Particle Swarm Optimization is an evolutionary numerical optimization method that has proven capable of finding global optimum solutions when applied to a wide variety of problems. The algorithm takes advantage of information sharing between particles in a population, or swarm, to converge to a solution. In this thesis, the algorithm is applied to the problem of a finite thrust orbital transfer between two co-planar circular orbits, and is solved for three different ratios of orbital radii. Different methods of initializing the swarm are investigated to determine any performance benefits. Three different techniques for creating the initial swarm are analyzed: uniform random number generation, and two variations of quasi-random number generation using Sobol sequences. Additionally, initial swarm sizes of 50, 100, 200, 500, 1000, 2000 and 3000 particles are used. After running all cases, results generally suggest that increasing the initial swarm size improves the performance of the PSO algorithm. The best method of swarm creation appears to be problem dependent. Future work should consider a larger number of cases to better define trends in the data and, if possible, reveal more general conclusions.

TABLE OF CONTENTS

List of Figures iii

List of Tables iv

Chapter 1 Introduction 1

- Literature Review 2
- Sobol Sequences 3

Chapter 2 Problem Statement 5

Chapter 3 Method 10

- Execute Multiple PSO Runs 10
- PSO Main Function 11
- Evaluate Objective Function 12
- Evaluate Best 50 Particles 13
- Evaluate Best Particle and Global Position 13
- Update Velocity 14
- Update Position 15
- ODE for First Thrust Arc 15
- ODE for Second Thrust Arc 15
- Create Sobol Sequence Arrays 16
- Map Sobol Sequence Arrays 16

Chapter 4 Results 17

Chapter 5 Conclusion 26

Appendix A PSO MATLAB Code 28

- run_multiple.m 28
- PSO_Vary_Initial_Swarm.m 29
- EvalJ.m 32
- Eval_Best_50.m 34
- EvalPGBest.m 34
- UpdateV.m 35
- UpdateP.m 35
- ThrustArc1_ODE.m 36
- ThrustArc2_ODE.m 36
- create_sobol_set_arrays.m 37
- map_array.m 37

BIBLIOGRAPHY 39

ACADEMIC VITA 40

LIST OF FIGURES

Figure 1: Comparison of Sampling Techniques 4

Figure 2: First Iteration Time vs. Initial Swarm Size, Uniform, β = 4.....18

Figure 3: GG vs. Initial Swarm Size- β = 2 19

Figure 4: GG vs. Initial Swarm Size- β = 4 20

Figure 5: GG vs. Initial Swarm Size- β = 8 21

Figure 6: Added Benefit vs. Initial Swarm Size- β = 2.....22

Figure 7: Added Benefit vs. Initial Swarm Size- β = 4.....23

Figure 8: Added Benefit vs. Initial Swarm Size- β = 8.....23

Figure 9: GGstar vs. Iteration Number- Sobol, β = 2, 50 initial particles.....24

Figure 10: GGstar vs. Iteration Number- Sobol, β = 2, 3000 initial particles.....25

Figure 11: Optimal Transfer Trajectories Found Using PSO- Sobol, β = 2 25

LIST OF TABLES

Table 1: Summary of Cases Run..17

Chapter 1

Introduction

Particle Swarm Optimization (PSO) is a stochastic numerical optimization method designed to find a global optimum solution to a given problem. It was first introduced by Eberhart and Kennedy in 1995 [1]. Intended to mimic the behavior of a flock of birds searching for food, the algorithm takes advantage of information sharing between particles in a population to converge to a solution. For each iteration during its execution, the PSO algorithm also uses information from previous iterations, or generations, to adapt its search for the global optimum. For this reason, PSO is referred to as an evolutionary algorithm.

When using PSO to solve a problem, the first step is to create a random population of particles. Each particle has a position, which represents a possible solution to the problem and corresponds to a value of the objective (or cost) function. The objective function is the expression that will be minimized or maximized. Each particle also has an associated velocity, which determines how the position changes. For each iteration in the PSO process, the value of the objective function is evaluated for every particle in the swarm and the best solution is noted. Next, the velocity of each particle is updated based on the best solutions found by each particle and the best solution found by the entire swarm. The velocity is then used to update the position of each particle, and the process repeats for the next iteration. At the end of a given number of iterations, the PSO algorithm produces the best solution found across all particles and generations.

Overall, PSO is intuitive and easy to program, yet has often proved capable of finding a global optimum solution to a problem in a reasonable amount of time. Researchers have shown that increasing the number of particles in the swarm or increasing the number of iterations can improve the accuracy and

effectiveness of the PSO algorithm. Unfortunately, both of these adjustments can be computationally expensive. Thus far, however, little research has been done into how the particle swarm is initialized. In this thesis, two different swarm initialization techniques are analyzed while the PSO algorithm is applied to optimize the finite thrust orbital transfer between two co-planar circular orbits.

First, the number of particles in the swarm is increased for the initial generation only. Initial swarms of 100, 200, 500, 1000, 2000 and 3000 particles are evaluated. The best 50 particles are then chosen from the initial swarm and execution of the PSO algorithm proceeds normally. Additionally, three different methods of generating the random initial swarm are examined: uniform random number generation and two different methods of quasi-random number generation using Sobol sequences. For all combinations of these swarm initialization techniques, the PSO code is run twenty times. The speed and performance of the algorithm is then averaged across all runs and analyzed to determine any benefits.

Literature Review

Pontani and Conway apply Particle Swarm Optimization to solve four different problems: the determination of periodic orbits in the context of the circular restricted three-body problem, the optimization of the Earth-to-Mars low-thrust orbital transfer, the optimization of the impulsive transfer between two circular orbits, and the optimization of the finite thrust transfer between two circular orbits [2]. The authors' work on the latter of these problems was used as a basis for this research. They demonstrate PSO's ability to find optimal trajectories in all four situations, and offer methods for dealing with equality and inequality constraints inherent in each problem [2]. In another paper, Pontani and Conway use PSO in four more space trajectory applications, including the determination of the globally optimal two- and three-impulse transfer trajectories between two co-planar circular orbits, the determination of the optimal transfer between two coplanar, elliptic orbits with arbitrary orientation, the determination of the optimal two-impulse transfer between two circular, non-coplaner elliptic orbits, and

the determination of the globally optimal two-impulse transfer between two co-planar elliptic orbits [3]. Again, they find PSO capable of finding globally optimal solutions in all cases. They also note that the number of particles and iterations required for the success of the PSO algorithm changes by problem. The authors suggest that a greater number of particles and iterations should be used for problems with a large number of constraints and/or unknowns [3].

Angeline applies a hybrid version of Particle Swarm Optimization to four test problems: the sphere function, the Rosenbrock function, the generalized Rastrigrin function, and the generalized Griewank function [4]. For each iteration in his algorithm, the swarm is sorted based on the value of the objective function. The positions and velocities of the best half of the population are then used to replace those of the bottom half of the population. Angeline finds that this selection technique significantly improves the performance of the Particle Swarm algorithm in three of the four experimental problems [4].

Similarly, Richards and Ventura use PSO to solve eight different test functions. However, instead of random initial particles, they use generators from centroidal Voronoi tessellations (CVT) to create the starting positions of the swarm. This method produces an initial swarm that is more evenly distributed throughout the solution search space [5]. While noting that CVT initialization can be computationally expensive, Richards and Ventura find that using it can have significant impact on the performance of the PSO algorithm [5].

Sobol Sequences

A Sobol sequence is a set of quasi-random numbers that sample an n-dimensional space more evenly than a simple random distribution. They are named for Russian mathematician I. M. Sobol, who first introduced them in 1967 [6]. Sobol sequences are useful in many applications because they avoid sample clustering; this can be visualized for a 2-dimensional case in Figure 1 below. The plot on the left was produced using uniform random number generation, while the plot on the right was created using

Sobol sequencing. The gaps and clustering in the sample data is evident in the plot showing random number generation. Another advantage of the Sobol sequence is that if further refinement is desired, additional points can be added while still preserving the even distribution across all of the data. This advantage can also be observed in Figure 1. Each color marks an additional 30 points that were added to the existing data.

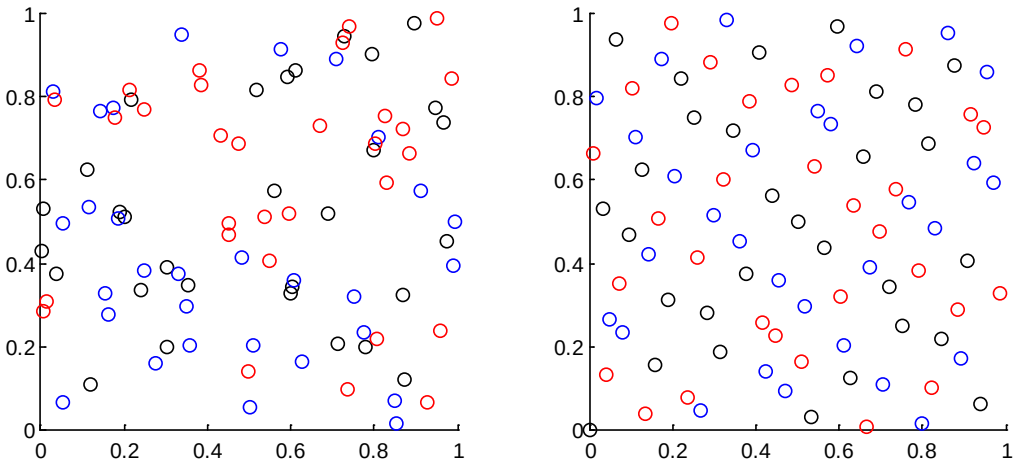

Figure 1: Comparison of Sampling Techniques

For this study, two different methods of creating the initial swarm using Sobol sequences are used. In the first method, which for the purposes of this paper will be referred to simply as "Sobol," the values that are used to initialize the swarm are always taken from the beginning of the Sobol sequence. In the second method, the values used to initialize the swarm are taken from different portions of the Sobol sequence for each of the twenty runs. For the n-th run, values will be drawn starting with $3000n + 1$ term in the Sobol sequence. In other words, for each consecutive run, an additional 3000 terms in the Sobol sequence are skipped. For this reason, this method of using Sobol sequences will be referred to as "Sobol skip." The number 3000 was chosen because the largest swarm that needs to be initialized contains 3000 particles.

Chapter 2

Problem Statement

The problem analyzed with PSO is the finite thrust transfer of a spacecraft from a circular orbit with radius R_1, to a co-planar circular orbit with radius R_2, where $R_2 > R_1$. The ratio of orbital radii β is defined as

$$\beta = \frac{R_2}{R_1} \tag{1}$$

In this analysis, the problem is solved for three different values of β: $\beta = 2$, $\beta = 4$, and $\beta = 8$. The transfer trajectory is assumed to have three parts: an initial thrust portion, a coast portion, and a second thrust portion. The desired solution is the thrust pointing angle time history and duration for each of the three portions, such that the spacecraft achieves the desired orbit while using the minimum amount of propellant. The initial thrust portion is assumed to begin at time $t_0 = 0$. The initial conditions for the spacecraft are

$$v_r(t_0) = v_{r_0} = 0 \tag{1}$$

$$v_\theta(t_0) = v_{\theta_0} = \sqrt{\frac{\mu}{R_1}} \tag{2}$$

$$r(t_0) = r_0 = R_1 \tag{3}$$

$$\xi(t_0) = \xi_0 = 0 \tag{4}$$

where v_r is the spacecraft's radial velocity, v_θ is the spacecraft's tangential velocity, μ is the gravitational parameter of the attracting body, r is the radius of the orbit, and ξ is the spacecraft's total angular displacement from its starting position. Throughout this document, as seen in Equations 1-4, a numeric

subscript n will be used to denote the value of that variable at time t_n. Additionally, note that with the exception of angles, which are expressed in radians, all values are expressed in canonical units.

For the duration of both thrust arcs, it is assumed that the spacecraft uses the maximum thrust. The thrust-to-mass ratio $\frac{T}{m}$ is defined as

$$\frac{T}{m} = \begin{cases} \dfrac{cn_0}{c - n_0 t} & for\ t_0 \le t \le t_1 \\ 0 & for\ t_1 \le t \le t_2 \\ \dfrac{cn_0}{c - n_0(t_1 + t - t_2)} & for\ t_2 \le t \le t_3 \end{cases} \tag{5}$$

where c is the effective exhaust velocity of the spacecraft thrusters, n_0 is the spacecraft's initial thrust-to-mass ratio, t_1 is the time at which the first thrust arc ends and the coast arc begins, t_2 is the time at which the coast arc ends and second thrust arc begins, and t_3 is the time at which the second thrust arc ends. The time interval for each of the three trajectory portions is determined by the PSO algorithm. The equations of motion for the spacecraft are

$$\dot{v}_r = -\frac{\mu - r v_\theta^2}{r^2} + \frac{T}{m}\sin\delta \tag{6}$$

$$\dot{v}_\theta = -\frac{v_r v_\theta}{r} + \frac{T}{m}\cos\delta \tag{7}$$

$$\dot{r} = v_r \tag{8}$$

$$\dot{\xi} = \frac{v_\theta}{r} \tag{9}$$

where δ is the thrust pointing angle, relative to the local horizontal. This angle is assumed to be a third-degree polynomial function of time. For the first thrust arc, δ is defined as

$$\delta = \zeta_0 + \zeta_1 t + \zeta_2 t^2 + \zeta_3 t^3 \tag{10}$$

Like the time intervals, the values of the coefficients ζ_0, ζ_1, ζ_2, and ζ_3 are optimally determined by the PSO algorithm. Equations 6-9 can be integrated over time to determine the first thrust portion of the trajectory, and the state of the spacecraft when the thrust arc ends.

The coast portion of the transfer trajectory is assumed to be an elliptic Keplerian arc. The semi major axis a and eccentricity e of the coast orbit can therefore be calculated using the following

$$a = \frac{\mu r_1}{2\mu - r_1\left(v_{r_1}^2 + v_{\theta_1}^2\right)} \tag{11}$$

$$e = \sqrt{1 - \frac{r_1^2 v_{\theta_1}^2}{\mu a}} \tag{12}$$

where v_{r_1}, v_{θ_1}, r_1, and $\xi_1$are determined after integration. Note that in order to be an elliptic orbit, a must be a positive value. The spacecraft's true anomaly ϑ and eccentric anomaly E at time t_1 are defined as

$$\vartheta_1 = \begin{cases} \cos^{-1}\left(\frac{v_{\theta_1}}{e}\sqrt{\frac{a(1-e^2)}{\mu}} - \frac{1}{e}\right) & for\ v_{r_1} \geq 0 \\ 2\pi - \cos^{-1}\left(\frac{v_{\theta_1}}{e}\sqrt{\frac{a(1-e^2)}{\mu}} - \frac{1}{e}\right) & for\ v_{r_1} < 0 \end{cases} \tag{13}$$

$$E_1 = 2\tan^{-1}\left(\sqrt{\frac{1-e}{1+e}}\tan\left(\frac{\vartheta_1}{2}\right)\right) \tag{14}$$

The change in eccentric anomaly during the coast segment ΔE is determined experimentally by the PSO algorithm. From that value, the eccentric anomaly at the end of the coast arc can be calculated using

$$E_2 = E_1 + \Delta E \tag{15}$$

The time interval for the coast segment, the time at the end of the coast segment, and the true anomaly at the end of the coast segment can then be calculated using the following:

$$\Delta t_{coast} = \sqrt{\frac{a^3}{\mu}}\left[E_2 - E_1 - e\left(\sin E_2 - \sin E_1\right)\right] \tag{16}$$

$$t_2 = \Delta t_{coast} + t_1 \tag{17}$$

$$\vartheta_2 = 2\tan^{-1}\left(\sqrt{\frac{1+e}{1-e}}\tan\frac{E_2}{2}\right) \tag{18}$$

Using this true anomaly and the orbital elements calculated using Equations 11 and 12, the spacecraft's velocity components, orbital radius, and total angular displacement at the beginning of the final thrust arc can be calculated using

$$v_{r_2} = \sqrt{\frac{\mu}{a(1-e^2)}}e\sin\vartheta_2 \tag{19}$$

$$v_{\theta_2} = \sqrt{\frac{\mu}{a(1-e^2)}}(1+e\cos\vartheta_2)e \tag{20}$$

$$r_2 = \frac{a(1-e^2)}{1+e\cos\vartheta_2} \tag{21}$$

$$\xi_2 = \xi_1 + (\vartheta_2 - \vartheta_1) \tag{22}$$

The equations of motion that govern the spacecraft's behavior during the second thrust arc are the same as those for the first thrust arc (Equations 6-9). However, in this case, the thrust pointing angle is defined as

$$\delta = \zeta_4 + \zeta_5(t-t_2) + \zeta_6(t-t_2)^2 + \zeta_7(t-t_2)^3 \tag{23}$$

As before, the coefficients ζ_4, ζ_5, ζ_6, and ζ_7 are optimally determined by the PSO algorithm. The equations of motion can again be integrated over time to determine the second thrust portion of the trajectory.

When the spacecraft terminates the second thrust arc, its velocity and orbital radius must match those of the target orbit. In other words, the following equations must hold true:

$$v_{r_3} = 0 \tag{24}$$

$$v_{\theta_3} - \sqrt{\frac{\mu}{R_2}} = 0 \tag{25}$$

$$r_3 - R_2 = 0 \tag{26}$$

where v_{r_3}, v_{θ_3}, r_3, and $\xi_3$are determined after integrating.

Overall, there are eleven unknowns in this problem: the eight coefficients for calculating the thrust pointing angles, and the time intervals for each of the three trajectory segments. The optimal values for each of these unknowns is determined such that the total thrust time is minimized. This corresponds to the desired condition of minimum propellant consumption. However, in addition to the thrust time, the objective function for this problem also includes a penalty term. This penalty term inflates the value of the objective function if the spacecraft's final velocity and orbital radius do not match those of the target orbit (i.e., Equations 24, 25, and 26 are not satisfied) within an allowable error margin. Thus, the objective function J is calculated as

$$J = \Delta t_{01} + \Delta t_{23} + \sum_{k=1}^{3} \alpha_k |d_k| \tag{24}$$

$$\Delta t_{01} = t_1 - t_0 \;\; and \;\; \Delta t_{23} = t_3 - t_2$$

where d_k is the error calculated in each of Equations 24, 25, and 26, and α_k is a multiplier. This multiplier term will be either a finite number or zero, depending on whether or not the corresponding error value violates a specified error margin.

Chapter 3

Method

In this section, the specific PSO algorithm that was used to solve the above problem is described in detail. Each step was programmed and run using MATLAB, and the names of corresponding .m files are provided. The full text for each file can be found in Appendix A.

Execute Multiple PSO Runs

In this thesis, there were several unique versions of the PSO code that were investigated, with each case being run 20 times. As a result, there were over a thousand PSO runs that needed to be completed. The script run_multiple.m was used to automate the process of executing all of these runs. Using nested for loops, the script iterates through each unique case. The inner most loop has twenty iterations, with each one calling the main function that executes the PSO algorithm, PSO_Vary_Initial_Swarm.m. The control structure is set up as a parfor loop (parallel for loop), which is a built-in MATLAB function that allows several iterations of the loop to be executed simultaneously across multiple computer processors, either locally or on a remote cluster. Since each individual run takes around ten minutes to complete, utilizing the parfor loop drastically reduces the time it takes to complete the full set of runs.

PSO Main Function

PSO_Vary_Initial_Swarm.m is the main function for the PSO code. It initializes all the necessary variables and constants, defines the bounds for the particles' position and velocity, calls the functions that execute the other steps in the PSO algorithm, and saves a log file at the end containing the results.

PSO_Vary_Initial_Swarm requires inputs for the value of β (variable "beta"), the initial swarm size (N_particles_initial), and the swarm creation technique (distribution). If the creation technique is not explicitly specified, the code defaults to the uniform random number generation. Next, the basic swarm characteristics are specified. The number of particles in the swarm is set to 50, while the number of iterations for the PSO code to run is set to 1000. The number of elements being optimized is 11. This number comes from the four thrust angles and burn duration for each of the two thrust arcs, and the change in eccentric anomaly for the coasting arc. The spacecraft and orbit parameters are also specified. The effective exhaust velocity and initial thrust to mass ratio are 0.5 DU/TU and 0.2 DU/TU^2, respectively, while the gravitational parameter of the attracting body is set to 1 DU^3/TU^2 and the initial orbit radius is set to 1 DU. Again note that these are all canonical units.

Next, the position vector for each particle is initialized. The position vector has the form

$$P = [\zeta_0 \quad \zeta_1 \quad \zeta_2 \quad \zeta_3 \quad \zeta_4 \quad \zeta_5 \quad \zeta_6 \quad \zeta_7 \quad \Delta t_{01} \quad \Delta E \quad \Delta t_{23}] \tag{25}$$

where the values of ζ are unit-less coefficients, Δt_{01} and Δt_{23} have units of TU, and ΔE has units of radians. The upper and lower bounds, BUp and BLp, for each element in the position vector are given by

$$BLp = [-1 \quad -1 \quad -1 \quad -1 \quad -1 \quad -1 \quad -1 \quad -1 \quad 0.06 \quad 0 \quad 0.04] \tag{26}$$

$$BUp = [1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1.4 \quad 2\pi \quad 1.0] \tag{27}$$

The initial value of each element for all initial particles is then assigned between these bounds. For the uniform case, this assignment is performed using MATLAB's random number generator. For the Sobol and Sobol skip cases, the initial element values for all initial particles are mapped from a Sobol sequence using the function map_array.m. That function is described in a later subsection.

Using the position bounds, the velocity bounds BLv and BUv are calculated as follows:

$$BUv = BUp - BLp \tag{28}$$

$$BLv = -BUv \tag{29}$$

Several additional arrays which store swarm information are then created and initialized. J stores the values of the objective function for all particles in a given generation, while V stores the velocity vectors. The array d holds for each particle the error values calculated from Equations 24, 25, and 26. PBest stores the best position ever found by each particle up until the current generation, while JBest holds the corresponding values of the objective function. Similarly, the array GBest stores the best position ever visited by the entire swarm, while GG stores the corresponding value of the objective function. GGstar contains a history of the value of GG by generation. Lastly, the array Iteration_time stores the time in seconds that it takes to run each iteration of the code. All of these arrays are initialized to zero, with the exception of JBest and GG which are initialized to infinity. This ensures that any finite value of the objective function will be an improvement over the initial value.

After all the necessary variables have be initialized, the code begins looping through iterations. For each generation, the functions EvalJ, EvalPGBest, UpdateV, and UpdateP are called. In addition, if the initial swarm size is not 50, the function Eval_Best_50 is called during the first iteration. A description of each of these functions can be found below. After each iteration, the global best value of the objective function, GG, is logged in the array GGstar. Finally, once all 1000 iterations have completed, a log file is saved which contains the arrays GGstar, Iteration_time, and GBest.

Evaluate Objective Function

The function EvalJ.m loops through each of the particles in the swarm, evaluates the objective function using the governing equations presented in Chapter 2 of this document, and stores the result in the variable J. The integrator used for the thrust trajectory segments is ode45, a variable time-step solver

that uses a fourth and fifth order Runge-Kutta method and is built in to MATLAB. The absolute and relative error tolerances for both thrust arcs are set to 10^{-7}.

It should be noted that if the semi-major axis for the coast portion is calculated to be negative (Equation 11), the value of the objective function for that particle is set to infinity, and its velocity is set to 0. A negative semi-major axis violates the constraint that the coast portion is an elliptic Keplerian arc, which mandates that the semi-major axis be positive. The particle's position thus corresponds to an invalid trajectory, and making the above changes helps to lead the swarm away from it.

Assuming the value of the semi-major axis is greater than zero, however, the objective function is evaluated as normal using Equation 24. If the errors calculated in Equations 21, 22 and 23 are smaller than 10^{-3}, the corresponding multiplier is set to 0. However, if any of the errors are greater than 10^{-3}, the corresponding multiplier is 100, and the penalty term in the objective equation becomes non-zero.

Evaluate Best 50 Particles

For cases when the initial swarm size is greater than 50 particles, the function Eval_Best_50.m is called immediately after EvalJ.m in the first iteration. This function simply finds and keeps the best 50 particles, while discarding the rest. The particles are judged based on the values of the objective function that were calculated in EvalJ. The PSO algorithm then proceeds as usual. Eval_Best_50.m is not called again.

Evaluate Best Particle and Global Position

The function EvalPGBest.m loops through each particle in the swarm and determines whether its current position produces a smaller value of the objective function than its previous best position. If it does, PBest for that particle is set equal to the particles current position P, and JBest is set to

corresponding value of the objective function J. If not, the previous values of PBest and JBest are retained. Similarly, the particle's current position is also compared to the previous best global position. Again, if the current value of J is smaller than the global best value of GG, GBest is set equal to the particles position P and GG is set equal to J. If not, the previous values of GBest and GG are retained.

Update Velocity

The function UpdateV.m updates the velocity vector for all particles in the swarm. The updated velocity V_{new} is calculated using

$$V_{new} = c_I V + c_C(P_{Best} - P) + c_S(G_{Best} - P) \tag{30}$$

where V and P are the particle's current velocity and position, c_I is the inertial weighting coefficient, c_C is the cognitive weighting coefficient, and c_S is the social weighting coefficient. The weighting coefficients are defined as

$$c_I = \frac{1 + rand}{2} \tag{31}$$

$$c_C = 1.49445 \times rand \tag{32}$$

$$c_S = 1.49445 \times rand \tag{33}$$

where $rand$ is a uniform random number between 0 and 1. These coefficients were used by Pontani and Conway when they applied PSO to solve this same problem [2]. The same set of weighting coefficients is used for all particles in the swarm, but are re-calculated for each generation.

Once the coefficients have been calculated, the velocity vector is updated for each particle in the swarm. Lastly, each element in the new velocity vector is checked to make sure it does not exceed any of the bounds specified in BUv or BLv. If it does, that element is set equal to the bound it violated.

Update Position

In UpdateP.m, each particle's position is updated using the velocity that was calculated in UpdateV. The equation for the position update is

$$P_{new} = P + V_{new} \tag{34}$$

Like the new velocity vector, each element in the new position vector is checked to make sure that it does not exceed any of the bounds specified in BUp or BLp. If it does, that element is set equal to the bound it violated, and the corresponding element in the velocity vector is reset to zero. This helps to prevent any of the position elements from getting stuck up against a bounding value.

ODE for First Thrust Arc

The function ThrustArc1_ODE.m is solved using MATLAB's ode45 integrator to calculate the spacecraft's trajectory during the first thrust arc. Equations 6, 7, 8 and 9 are the equations of motion that are integrated, while the thrust-to-mass ratio $\frac{T}{m}$ and the thrust pointing angle δ are defined by Equations 5 and 10, respectively. The initial conditions are the spacecraft's radial velocity, tangential velocity, orbital radius, and total angular displacement at time t_0. After running, ThrustArc1_ODE.m returns those same variables as functions of time to EvalJ.m.

ODE for Second Thrust Arc

The function ThrustArc2_ODE.m is solved using MATLAB's ode45 integrator to calculate the spacecraft's trajectory during the second thrust arc. Equations 6, 7, 8 and 9 are the equations of motion that are integrated, while the thrust-to-mass ratio $\frac{T}{m}$ and the thrust pointing angle δ are defined by Equations 5 and 23, respectively. The initial conditions are spacecraft's radial velocity, tangential

velocity, orbital radius, and total angular displacement at the end of the coast segment of the transfer trajectory. After running, ThrustArc2_ODE.m returns those same variables as functions of time to EvalJ.m.

Create Sobol Sequence Arrays

When executed, the script create_sobol_arrays.m creates the Sobol sequence arrays that are then mapped using the bounds BLp and BUp to become the initial positions of all particles in the swarm. There are a total of 21 arrays that are created using this function, each one having dimensions of 3000x11. This corresponds to a maximum initial swarm size of 3000 particles, with 11 elements in each particle's position vector. MATLAB's Statistics Toolbox includes functions that allow a user to automatically generate an n-dimensional set of points from a Sobol sequence. The user can also choose to skip a certain number of terms in the Sobol sequence before values are drawn. Thus, the first array is created after skipping 0 terms in an 11-dimensional Sobol sequence (i.e., the terms were drawn from the beginning of the sequence). Each successive n-th array is created after skipping an additional $3000n$ terms. The arrays are then saved under filenames that include the value of n. For the Sobol case, array 0 is used for all 20 runs. In the Sobol skip case, arrays 1-20 are used, depending on the run number.

Map Sobol Sequence Arrays

By default, Sobol sequences generate values that fall between 0 and 1. Thus, before they become particle position vectors, the arrays containing terms from an 11-dimensional Sobol sequence need to be mapped on to the search space defined by BLp and BUp. This mapping is performed by the function map_array.m. Each row of the Sobol array (containing 11 elements) is linearly mapped from the range [0,1] to the range $[BLp, BUp]$.

Chapter 4

Results

Overall, the PSO algorithm detailed in the previous section was run 1260 times. It was run with three different values of β, three different swarm creation techniques, and seven different initial swarm sizes, for a total of 63 unique cases. A breakdown of the different cases can be found in Table 1.

Table 1: Summary of Cases Run

Variable	Number of Different Values	Values
Ratio of Orbital Radii $\beta = \frac{R_2}{R_1}$	3	2
		4
		8
Swarm Creation Method	3	Uniform random
		Sobol
		Sobol skip
Initial Swarm Size (no. of particles)	7	50
		100
		200
		500
		1000
		2000
		3000

Each of those cases was run 20 times so that an average performance for each case could be attained, resulting in 1260 total runs. Unless explicitly stated otherwise, all the results presented below are averages across the 20 repeated runs.

Figure 2 below shows a plot of the time for the first iteration of the PSO algorithm vs. initial swarm size for a uniform random initial swarm and $\beta = 4$. Only the first iteration is considered because

for all subsequent iterations in all cases, the swarm is reduced to 50 particles and the computation time would remain constant. Figure 2 demonstrates a directly proportional relationship between the number of initial particles in the swarm and the computation time for the first iteration. This is expected, since it is simply a matter of additional particles being evaluated in the EvalJ function. This relationship was consistent across all values of β and for all swarm creation techniques.

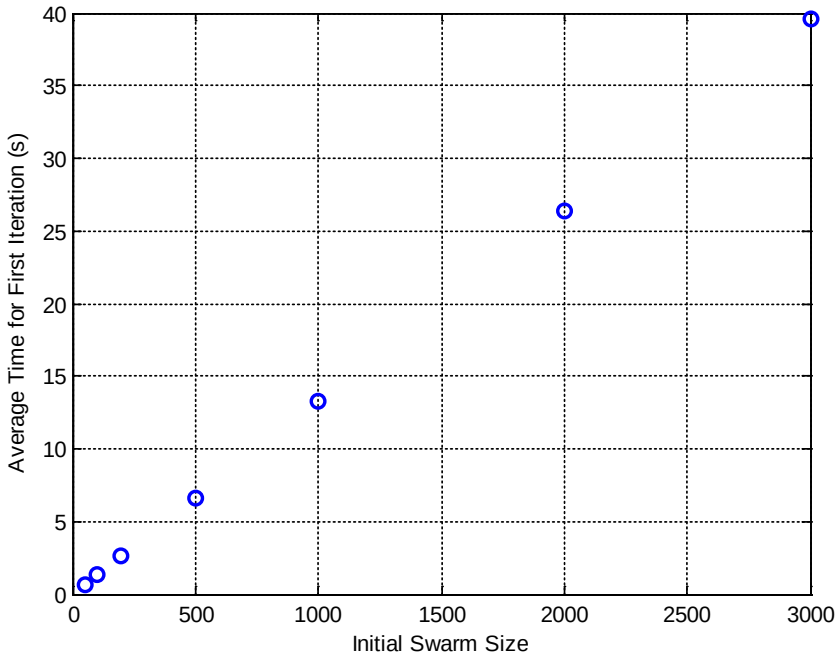

Figure 2: First Iteration Time vs. Initial Swarm Size, Uniform, β = 4

Next, the final value of the objective function, GG, is examined for each value of β. All three swarm creation techniques are shown on the same plot, with different line colors. Figure 3 shows the plot of GG vs initial swarm size for $\beta = 2$. In the figure, it can be seen for both the Sobol and uniform cases, increasing the size of the initial swarm generally decreases the final value of the cost function. The greatest impact is seen in the uniform case, which saw up to a 20% improvement by using an initial swarm larger than 50 particles. For the Sobol skip case, however, the value of GG increased as much as 7.5% from the addition of more initial particles. When comparing the swarm creation techniques, the Sobol cases achieved the lowest value of GG for initial swarms with greater than 500 particles. The uniform and Sobol skip cases performed similarly for the middle-sized initial swarms, with the Sobol skip

performing much better than the uniform case for smaller initial swarms, and slightly worse for the largest initial swarms.

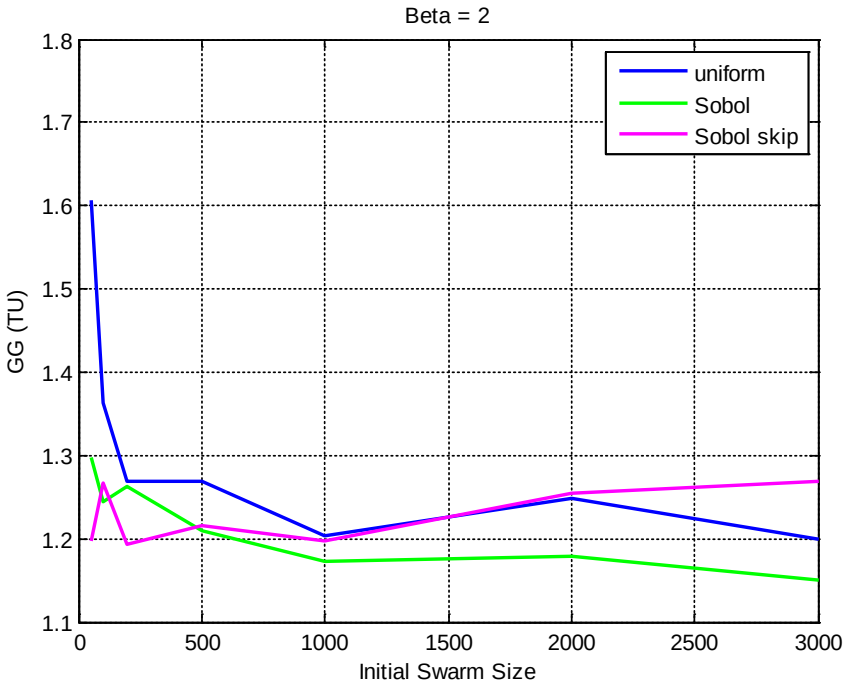

Figure 3: GG vs. Initial Swarm Size- β = 2

Figures 4 shows the plot of GG vs. initial swarm size for $\beta = 4$. In Figure 5, it can be seen that the Sobol skip case sees a drastic improvement in GG when the initial swarm is increased from 50 to 100 particles. Beyond that, however, additional increases in initial swarm size appear to have little effect. For the Sobol technique, increasing the initial swarm size up to 500 particles results in a small improvement in GG, but increasing the initial swarm size further does practically nothing. On the other hand, for the uniform case, increasing the initial swarm size from 50 to 100 or 200 particles actually caused an increase in GG. This increase in GG disappeared when the initial swarm size was increased to 500 particles, and further increases again appeared to have minimal effect. Like the $\beta = 2$ case, when comparing between swarm creation techniques, the Sobol method performed slightly better than the other two.

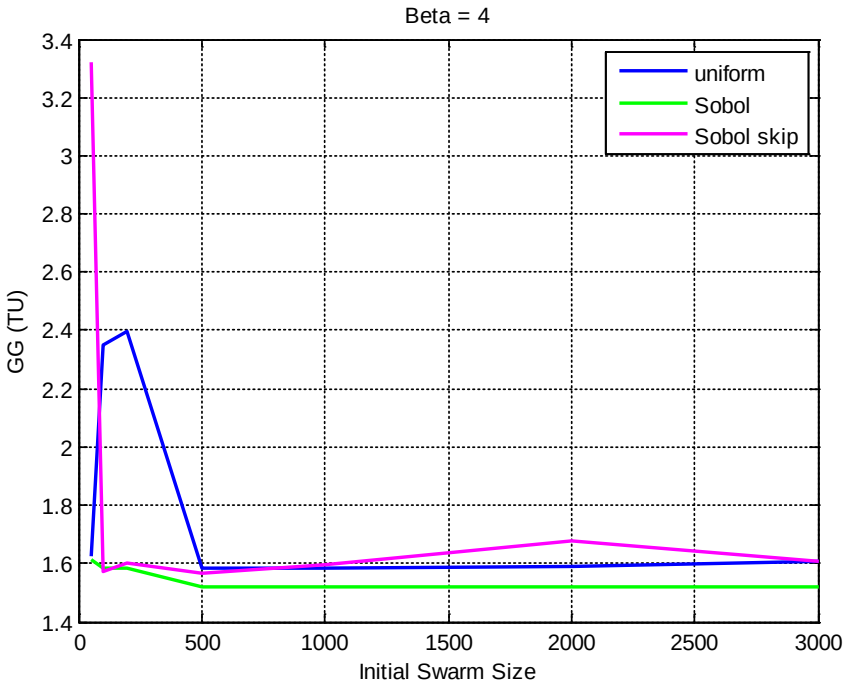

Figure 4: GG vs. Initial Swarm Size- β = 4

Next, the same plot showing GG vs. initial swarm size for the case when $\beta = 8$ can be seen in Figure 5. In general, there appears to be significantly more scatter in all of the data for this value of β. Looking at Figure 7, it is obvious that the largest improvement in GG with increasing initial swarm size was seen when the Sobol swarm creation technique was used. For that case, increasing the initial swarm size to 1000 particles or more resulted in an 80% decrease in GG compared to an initial swarm of 50 particles. Although they did not see nearly that level of decrease in GG, the uniform and Sobol skip cases generally saw smaller values of GG with increased initial swarm size. For initial swarm sizes up to 1000 particles, the Sobol skip method out-performed the other two. However, at 2000 particles, the Sobol case showed the lowest value of GG. For 3000 particles, the lowest value of GG was seen in the uniform case.

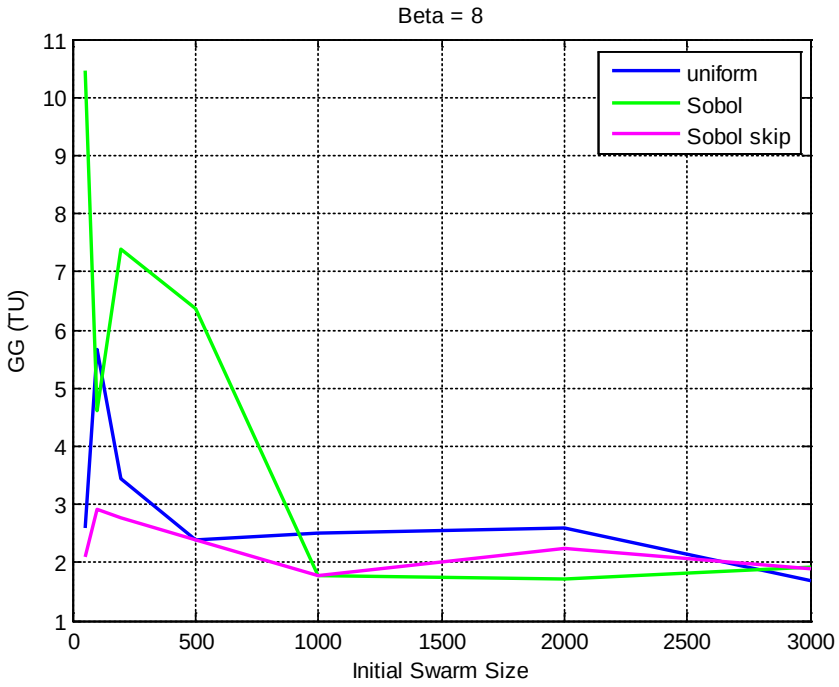

Figure 5: GG vs. Initial Swarm Size- β = 8

Figures 6, 7, and 8 show plots of the added benefit per second of additional computation time vs. initial swarm size for $\beta = 2$, $\beta = 4$, and $\beta = 8$, respectively. The added benefit per second of additional computation time (henceforth simply referred to as "added benefit") is calculated as follows:

$$Added\ Benefit_n = \frac{GG_{50} - GG_n}{t_{i,n}},\ \ for\ n = 100, 200, 500, 1000, 2000, 3000 \qquad (34)$$

where GG_{50} is the value of GG for an initial swarm of 50 particles, GG_n is the value of GG for the larger initial swarms, and $t_{i,n}$ is the additional computation time in seconds that it takes to complete the first iteration of the PSO algorithm. In all three plots, it can be seen that for all swarm creation techniques, the added benefit gets smaller and approaches zero as the initial swarm size is increased. This is expected, since the true optimal solution is a constant finite value, while the first iteration computation time increases proportionally with initial swarm size. Another interesting results that can best be seen on these three plots is that, for each value of beta, there is one swarm creation technique that sees significantly larger added benefits from increasing initial swarm size. When $\beta = 2$, the uniform case sees the largest

additional benefits. For $\beta = 4$ and $\beta = 8$, the largest benefits are seen in the Sobol skip and Sobol cases, respectively. These results seem to suggest that the swarm creation method is sensitive to the problem to which it is applied.

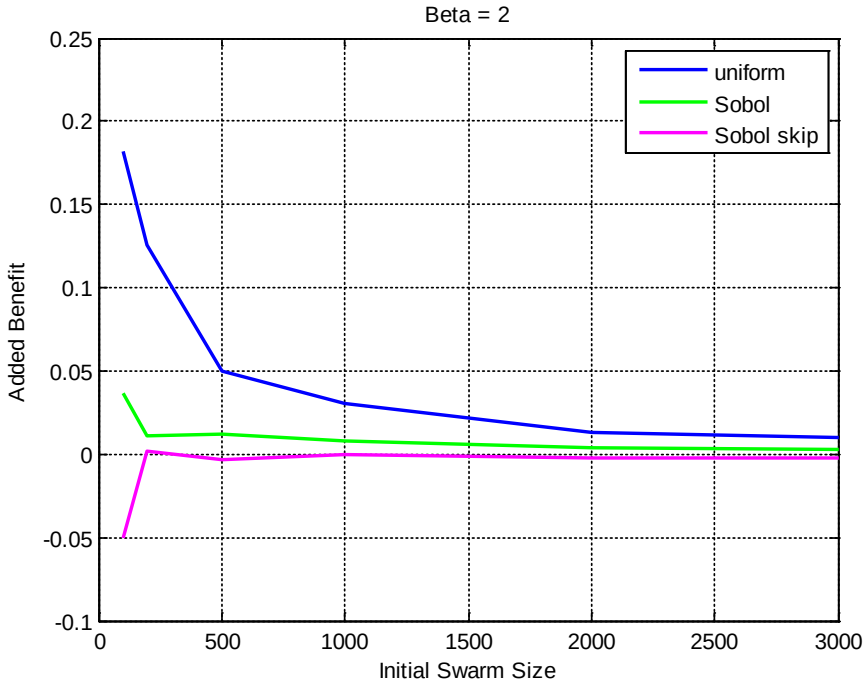

Figure 6: Added Benefit vs. Initial Swarm Size- β = 2

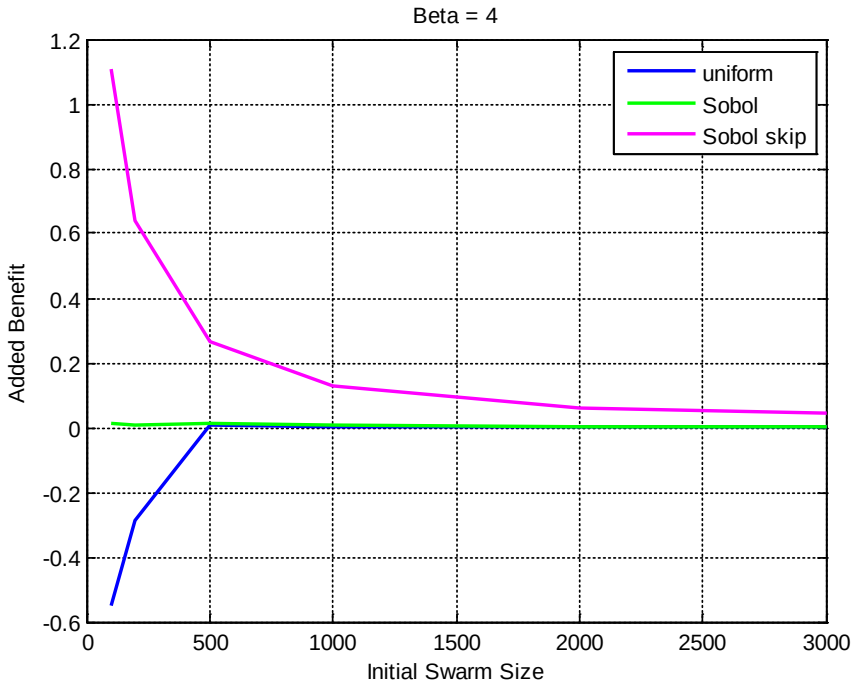

Figure 7: Added Benefit vs. Initial Swarm Size- β = 4

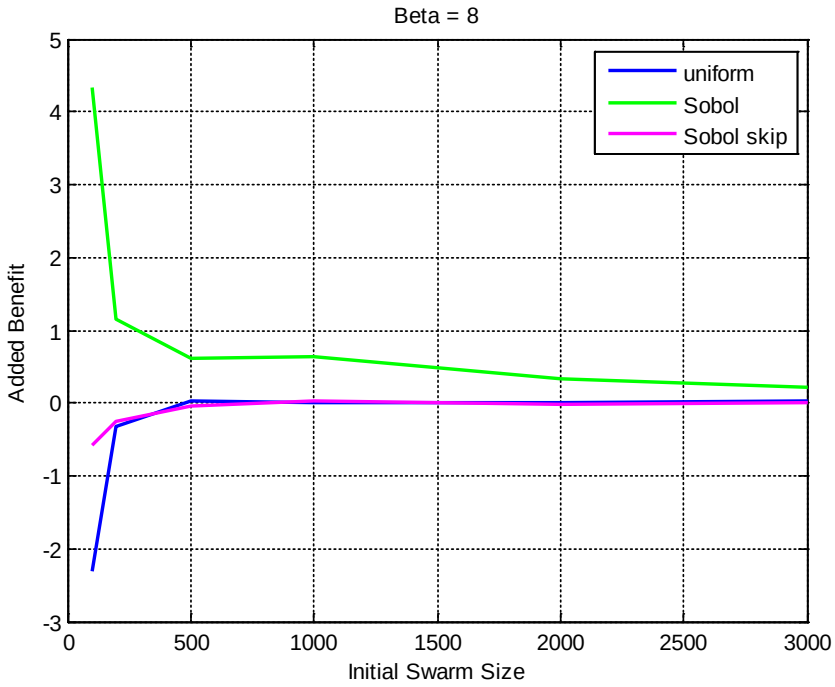

Figure 8: Added Benefit vs. Initial Swarm Size- β = 8

When examining individual runs from each of the 63 cases it becomes apparent that the PSO algorithm often finds and gets stuck on local minimum solutions instead of the global optimum. These local minima still satisfy the necessary conditions and constraints in the problem, but simply correspond to higher values for the objective function. This tendency can best be visualized using the following figures. Figure 9 shows a plot of GGstar vs Iteration number for the Sobol case when $\beta = 2$ and the initial swarm contains 50 particles. Recall that GGstar contains the value of GG for each generation. In that figure, it can be seen from the spread of final values that many different families of local minima were found. Figure 10 shows a plot of GGstar vs iteration number for Sobol case when $\beta = 2$ as well, except now there are 3000 particles in the initial population. Compared to Figure 9, Figure 10 shows a much smaller spread in final values. This difference can really be noticed in Figure 11, which shows the trajectories from all 20 runs of each case side by side (50 initial particles on the left, 3000 initial particles on the right). Similar trends were seen for the cases with the other creation techniques and different values of β. This suggests that increasing the initial swarm size may help the PSO algorithm to avoid local minima.

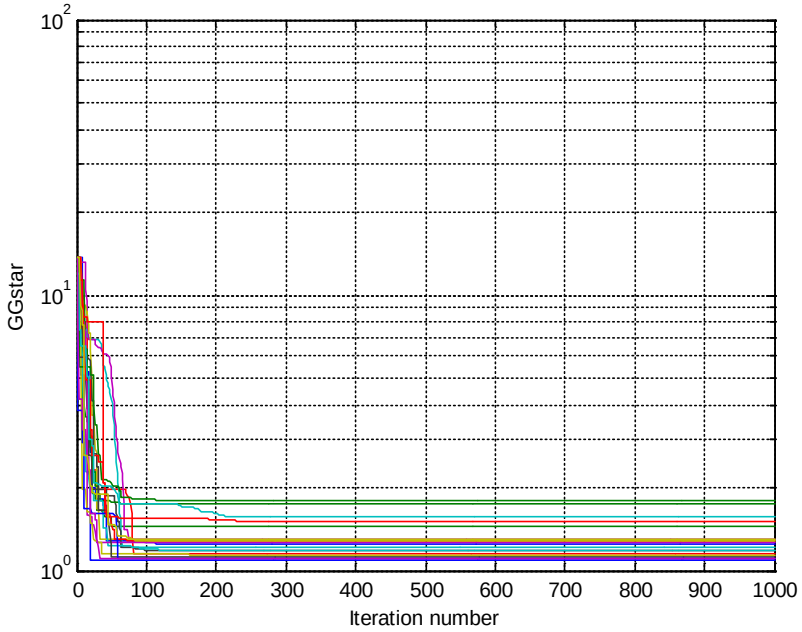

Figure 9: GGstar vs. Iteration Number- Sobol, β = 2, 50 initial particles

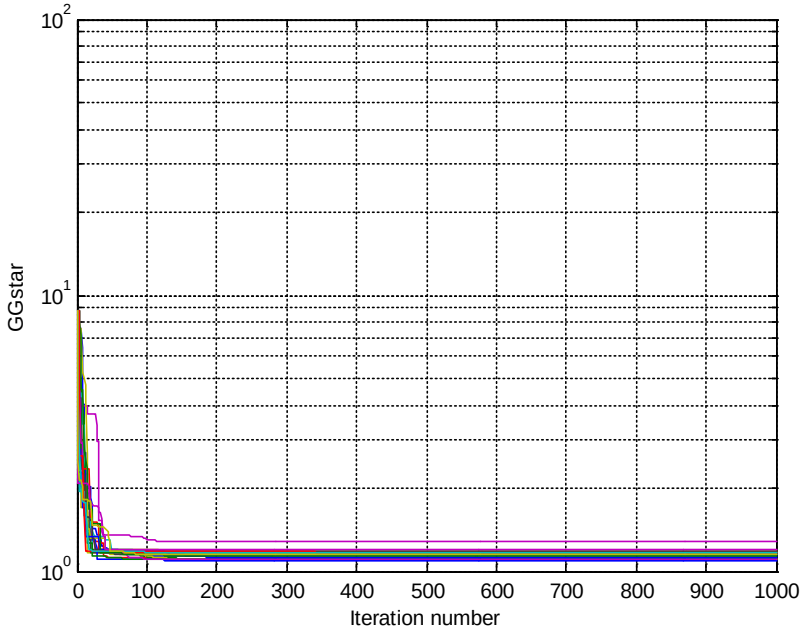

Figure 10: GGstar vs. Iteration Number- Sobol, β = 2, 3000 initial particles

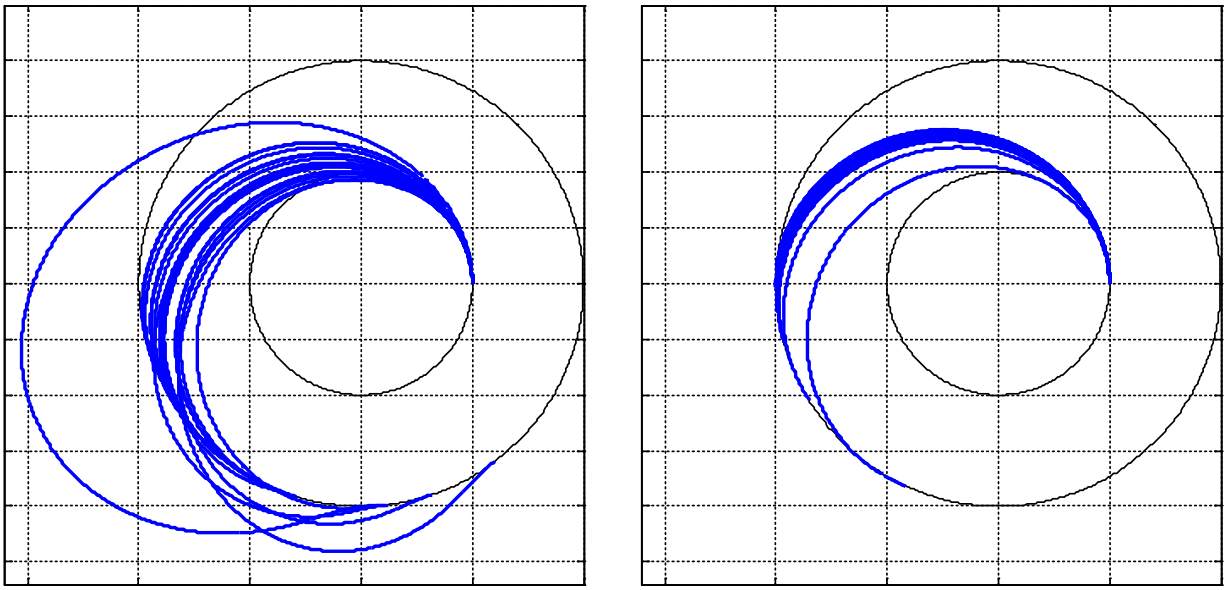

Figure 11: Optimal Transfer Trajectories Found Using PSO- Sobol, β = 2

Chapter 5

Conclusion

Overall, it appears that changing the swarm initialization technique can have an impact on the performance of the Particle Swarm Optimization method, at least when it is applied to optimize the finite thrust orbital transfer between two circular orbits. A PSO algorithm is used to optimize this problem for three different ratios of orbital radii β, using three different swarm creation techniques and seven different initial swarm sizes. Unfortunately, the scatter seen in the data across all three values of β makes it difficult to draw broad conclusions about which combination of initial swarm size and swarm creation method is best. Of the three swarm creation techniques, the Sobol method produced the smallest values of the objective function for the cases when $\beta = 2$ and $\beta = 4$. When $\beta = 8$, however, none of the methods emerged as the consistent top performer. Increasing the initial swarm size generally reduced the final value of the objective function achieved by the PSO algorithm, and the amount of improvement decreased as the number of particles in the initial swarm increased. In addition, for each value of β, increasing the initial swarm size had the largest effect on a different swarm creation technique. This indicates that the best combination of initial swarm size and swarm creation technique is problem dependent, instead of a general solution. Nevertheless, this study shows at least preliminary evidence that changing the way the swarm is initialization can have an effect on the quality of solution that is found.

In the future, each of the cases examined in this thesis should each be run several more times. It would be interesting to see if the data trends would smooth out after a sufficiently large number of runs. This would potentially allow for stronger conclusions to be drawn. In addition, the different swarm initialization techniques investigated in this thesis should be applied to other problems. If those

applications also demonstrate a performance improvement, it would strengthen the case for the adjustment of the swarm initialization as a means of tuning the Particle Swarm Optimization method.

Appendix A

PSO MATLAB Code

Included in this section is the complete set of MATLAB code that was used to run the PSO algorithm detailed throughout this thesis. There are two scripts and nine functions for a total of 11 MATLAB .m files.

run_multiple.m

```
clc
clear all
close all

beta = [2 4 8];
initial_population_size = [50 100 200 500 1000 2000 3000];
nruns = 20;
dist = {'uniform','sobol','sobol_skip'};

results_dir = 'results_test/';

if ~exist(results_dir,'dir')
    mkdir(results_dir)
end

for i = 1:length(dist)

    dist_dir = [results_dir,dist{i},'/'];
    if ~exist(dist_dir,'dir')
        mkdir(dist_dir)
    end

    for j=1:length(beta)

        beta_dir = [dist_dir,'beta',num2str(beta(j)),'/'];
        if ~exist(beta_dir,'dir')
            mkdir(beta_dir)
        end
```

```
        for k=1:length(initial_population_size)

            population_dir =
[beta_dir,'Initial_Swarm_',num2str(initial_population_size(k)),'/'];
            if ~exist(population_dir,'dir')
                mkdir(population_dir)
            end

            for m=1:nruns
                fprintf('\nDistribution: %s\nPopulation Size: %d\nBeta:
%d\nRun: %d\n',dist{i},initial_population_size(k),beta(j),m)

                if strcmp(dist{i},'sobol_skip')
                    sobol_input_filename =
['sobol_input/sobol_set_skip_',num2str(m),'.mat'];
                    fprintf('Filename: ')
                    fprintf(sobol_input_filename)
                    fprintf('\n')

PSO_Vary_Initial_Swarm(initial_population_size(k),beta(j),m,'Distribution',di
st{i},'SobolFilename',sobol_input_filename)
                else

PSO_Vary_Initial_Swarm(initial_population_size(k),beta(j),m,'Distribution',di
st{i})
                end
            end
        end
    end
end
```

PSO_Vary_Initial_Swarm.m

```
function PSO_Vary_Initial_Swarm(N_particles_initial,beta,run,varargin)
% PSO With Larger Initial Swarm

p = inputParser;
paramName1 = 'Distribution';
default1 = 'uniform';
validationFcn1 = @(x) ischar(x);
addParameter(p,paramName1,default1,validationFcn1);

paramName2 = 'SobolFilename';
default2 = 'sobol_input/sobol_set_skip_0.mat';
validationFcn2 = @(x) ischar(x);
addParameter(p,paramName2,default2,validationFcn2);

parse(p,varargin{:});

distribution = p.Results.Distribution;
```

```
sobol_input_filename = p.Results.SobolFilename;

swarm.N_particles = 50;
swarm.N_particles_initial = N_particles_initial;
swarm.N_elements = 11;
swarm.N_iterations = 10;

constants.beta = beta;
constants.mu = 1;
constants.R1 = 1;
constants.c = 0.5;
constants.n0 = 0.2;

% set lower and upper bounds on unknowns (particle elements)
swarm.BLp = [-1, -1, -1, -1, -1, -1, -1, -1, .06,    0,  .04];
swarm.BUp = [ 1,  1,  1,  1,  1,  1,  1,  1,  1.4,  2*pi, 1.0];

if strcmp(distribution,'sobol')
    % create initial population from Sobol set
    fprintf('\n sobol \n');
    fprintf(' Filename = %s',sobol_input_filename);
    sobol_set = load(sobol_input_filename);
    swarm.P =
map_array(sobol_set.data(1:swarm.N_particles_initial,:),swarm.BLp,swarm.BUp,'
col');
    results_dir = 'results_test/sobol/';

elseif strcmp(distribution,'sobol_skip')
    % create initial population from Sobol set
    fprintf('\n sobol skip \n');
    fprintf(' Filename = %s',sobol_input_filename);
    sobol_set = load(sobol_input_filename);
    swarm.P =
map_array(sobol_set.data(1:swarm.N_particles_initial,:),swarm.BLp,swarm.BUp,'
col');
    results_dir = 'results_test/sobol_skip/';

elseif strcmp(distribution,'uniform')
    % create uniform random initial population
    fprintf('\n uniform \n');
    swarm.P = zeros(swarm.N_particles_initial,swarm.N_elements);
    for i = 1:8
        swarm.P(:,i) = -1 + 2*rand(swarm.N_particles_initial,1);
    end

    swarm.P(:,9) = .06 + (1.3-.06)*rand(swarm.N_particles_initial,1);
    swarm.P(:,10) = 2*pi*rand(swarm.N_particles_initial,1);
    swarm.P(:,11) = .04 + (1.1-.04)*rand(swarm.N_particles_initial,1);
    results_dir = 'results_test/uniform/';

else
    fprintf('\nInvalid distribution type. Aborting...\n')
    return
end
```

```
swarm.J = zeros(swarm.N_particles_initial,1);
swarm.d = zeros(swarm.N_particles,3);
swarm.V = zeros(swarm.N_particles, swarm.N_elements);
swarm.PBest = zeros(swarm.N_particles,swarm.N_elements);
swarm.JBest = zeros(swarm.N_particles,1);
swarm.GBest = zeros(1,swarm.N_elements);
swarm.GGstar = zeros(swarm.N_iterations,1);
swarm.Iteration_time = zeros(swarm.N_iterations,1);


% determine velocity bounds
swarm.BUv = swarm.BUp - swarm.BLp;
swarm.BLv = -swarm.BUv;

for i = 1:swarm.N_particles
    swarm.JBest(i) = inf;
end
swarm.GG = inf;

tic

for j = 1:swarm.N_iterations

    if (j == 1) && (swarm.N_particles_initial ~= swarm.N_particles)
        fprintf('\n Not 50\n')
        swarm = EvalJ(swarm,constants,swarm.N_particles_initial);
        swarm = Eval_Best_50(swarm);
    else
        swarm = EvalJ(swarm,constants,swarm.N_particles);
    end
    swarm = EvalPGBest(swarm,constants);
    swarm = UpdateV(swarm,constants);
    swarm = UpdateP(swarm,constants);
    swarm.GGstar(j) = swarm.GG;
    swarm.Iteration_time(j) = toc;
end

if ~exist(results_dir,'dir')
    mkdir(results_dir)
end

beta_dir = [results_dir,'beta',num2str(beta),'/'];

if ~exist(beta_dir,'dir')
    mkdir(beta_dir)
end

initial_swarm_dir =
[beta_dir,'Initial_Swarm_',num2str(swarm.N_particles_initial),'/'];

if ~exist(initial_swarm_dir,'dir')
    mkdir(initial_swarm_dir)
```

```
end

log_filename = [initial_swarm_dir,'log_',sprintf('Run%03d',run),'.mat'];

log.GGstar = swarm.GGstar;
log.Iteration_time = swarm.Iteration_time;
log.GBest = swarm.GBest;

save(log_filename,'log');

end
```

EvalJ.m

```
function updated_swarm = EvalJ(swarm,constants,N_particles)
% EvalJ evaluates J for each particle in current iteration

for i = 1:N_particles

    R2 = constants.beta*constants.R1;
    xi0 = 0;
    vr0 = 0;
    vt0 = sqrt(constants.mu/constants.R1);
    t1 = swarm.P(i,9);

    statevec0 = [vr0; vt0; constants.R1; xi0];

    options = odeset('RelTol', 1e-7, 'AbsTol', 1e-7);
    tspan1 = [0,t1];
    [~,statevec1] =
ode45(@ThrustArc1_ODE,tspan1,statevec0,options,swarm,constants,i);

    last = length(statevec1);

    vr1 = statevec1(last,1);
    vt1 = statevec1(last,2);
    r1 = statevec1(last,3);
    xi1 = statevec1(last,4);

    a = constants.mu*r1/(2*constants.mu - r1*(vr1^2 + vt1^2));
    e = sqrt(1 - r1^2*vt1^2/(constants.mu*a));

    if(a<0)
        swarm.J(i) = inf;
        swarm.V(i,:) = 0;

    else
        if(vr1>=0)
            TA1 = acos(vt1/e*sqrt(a*(1-e^2)/constants.mu)-1/e);
        else
```

```
            TA1 = 2*pi - acos(vt1/e*sqrt(a*(1-e^2)/constants.mu)-1/e);
        end

        E1 = 2*atan(sqrt((1-e)/(1+e))*tan(TA1/2));

        if(E1 < 0)
           E1 = E1 + 2*pi;
        end

        E2 = E1 + swarm.P(i,10);

        deltat_coast = sqrt(a^3/constants.mu)*(E2 - E1 - e*(sin(E2) -
sin(E1)));

        t2 = deltat_coast + t1;

        TA2 = 2*atan(sqrt((1+e)/(1-e))*tan(E2/2));

        if(TA2 < 0)
            TA2 = TA2 + 2*pi;
        end

        vr2 = sqrt(constants.mu/(a*(1-e^2)))*e*sin(TA2);
        vt2 = sqrt(constants.mu/(a*(1-e^2)))*(1+e*cos(TA2));
        r2 = a*(1-e^2)/(1+e*cos(TA2));
        xi2 = xi1 + (TA2- TA1);

        statevec2 = [vr2; vt2; r2; xi2];
        tspan2 = [t2,t2+swarm.P(i,11)];
        [~,statevec3] =
ode45(@ThrustArc2_ODE,tspan2,statevec2,options,swarm,constants,i,t1,t2);

        last = length(statevec3);

        vr3 = statevec3(last,1);
        vt3 = statevec3(last,2);
        r3 = statevec3(last,3);
        xi3 = statevec3(last,4);

        swarm.d(i,1) = vr3;
        swarm.d(i,2) = vt3-sqrt(constants.mu/R2);
        swarm.d(i,3) = r3-R2;
        alpha = zeros(3,1);
        penalty = zeros(3,1);

        %Equality constraint penalties
        for m = 1:3
            if(abs(swarm.d(i,m)) > 1e-3)
                alpha(m) = 100;
            else
                alpha(m) = 0;
            end
            penalty(m) = abs(swarm.d(i,m))*alpha(m);
```

```
        end
        swarm.J(i) = swarm.P(i,9) + swarm.P(i,11) + sum(penalty);
    end
end

updated_swarm = swarm;

end
```

Eval_Best_50.m

```
function updated_swarm = Eval_Best_50(swarm)

[tempJ,index] = sortrows(swarm.J);
tempJ = tempJ(1:swarm.N_particles);
index = index(1:swarm.N_particles);

tempP = swarm.P(index,:);

swarm.J = tempJ;
swarm.P = tempP;

updated_swarm = swarm;

end
```

EvalPGBest.m

```
function updated_swarm = EvalPGBest(swarm,constants)
% EvalP&GBest determines the best position visited by each particle up
%   through the current iteration

for i = 1:swarm.N_particles
    if swarm.J(i) < swarm.JBest(i)
        swarm.PBest(i,:) = swarm.P(i,:);
        swarm.JBest(i) = swarm.J(i);
    end
end

for i = 1:swarm.N_particles
    if swarm.J(i) < swarm.GG
       swarm.GG = swarm.J(i);
       swarm.GBest = swarm.P(i,:);
       swarm.dBest = swarm.d(i,:);
    end
end
```

```
updated_swarm = swarm;

end
```

UpdateV.m

```
function updated_swarm = UpdateV(swarm,constants)
% UpdateV updates the velocity vector V

c_I = (1 + rand)/2;
c_C = 1.49445*rand;
c_S = 1.49445*rand;

for i =1:swarm.N_particles

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

        if swarm.V(i,k) > swarm.BUv(k)
            swarm.V(i,k) = swarm.BUv(k);
        end
    end
end

updated_swarm = swarm;

end
```

UpdateP.m

```
function updated_swarm = UpdateP(swarm,constants)
% UpdateP updates the position vector

for i =1:swarm.N_particles
    swarm.P(i,:) = swarm.P(i,:) + swarm.V(i,:);
    for k = 1:swarm.N_elements
        if swarm.P(i,k) < swarm.BLp(k)
            swarm.P(i,k) = swarm.BLp(k);
            swarm.V(i,k) = 0;
        end
        if swarm.P(i,k) > swarm.BUp(k)
            swarm.P(i,k) = swarm.BUp(k);
            swarm.V(i,k) = 0;
```

```
        end
    end
end

updated_swarm = swarm;
end
```

ThrustArc1_ODE.m

```
function xdot = ThrustArc1_ODE(t,x,swarm,constants,index)

%x(1) = vr
%x(2) = vt
%x(3) = r
%x(4) = xi

xdot = zeros(4,1);

T_M = constants.c * constants.n0 / (constants.c - constants.n0 * t);
delta = swarm.P(index,1) + swarm.P(index,2)*t + swarm.P(index,3)*t^2 +
swarm.P(index,4)*t^3;

xdot(1) = -(constants.mu - x(3) * x(2)^2) / (x(3)^2) + T_M*sin(delta);
xdot(2) = -x(1) * x(2) / x(3) + T_M*cos(delta);
xdot(3) = x(1);
xdot(4) = x(2) / x(3);

end
```

ThrustArc2_ODE.m

```
function xdot = ThrustArc2_ODE(t,x,swarm,constants,index,t1,t2)

%x(1) = vr
%x(2) = vt
%x(3) = r
%x(4) = xi

xdot = zeros(4,1);

T_M = constants.c * constants.n0 / (constants.c - constants.n0 * (t1 + t -
t2));
delta = swarm.P(index,5) + swarm.P(index,6) * (t-t2) + swarm.P(index,7) * (t-
t2)^2 + swarm.P(index,8) * (t-t2)^3;

xdot(1) = -(constants.mu - x(3) * x(2)^2) / (x(3)^2) + T_M*sin(delta);
xdot(2) = -x(1) * x(2) / x(3) + T_M*cos(delta);
```

```
xdot(3) = x(1);
xdot(4) = x(2) / x(3);

end
```

create_sobol_set_arrays.m

```
clc
clear

skip = 0:3000:60000;

for i=1:length(skip)
    p = sobolset(11,'Skip',skip(i));
    data = net(p,3000);
    filename = ['sobol_input/sobol_set_skip_',num2str(i-1),'.mat'];
    save(filename,'data');
end
```

map_array.m

```
function y = map_array(x,ymin,ymax,dim)

yrange = ymax-ymin;

if strcmp(dim,'row')
    x = x';
elseif strcmp(dim,'col')
    x = x;
else
    fprintf('Invalid mapping dimension.  Operation cancelled.\n\n')
    return
end

if (length(ymin) ~= length(ymax)) || (length(ymin) ~= size(x,2))
    fprintf('Dimension mismatch.  Operation cancelled.\n\n')
    return
end

y = NaN(size(x));

for i=1:size(x,2)
    xmin = min(x(:,i));
    xmax = max(x(:,i));
    xrange = xmax-xmin;
    xoffset = xmin;
    mapped_array = x(:,i) - xoffset;
```

```
    gain = yrange(i)/xrange;
    mapped_array = mapped_array.*gain + ymin(i);
    y(:,i) = mapped_array;
end

end
```

BIBLIOGRAPHY

[1] Eberhart, Russ C., and James Kennedy. "A new optimizer using particle swarm theory." *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*. Vol. 1. 1995.

[2] Pontani, Mauro, and Bruce A. Conway. "Particle swarm optimization applied to space trajectories." *Journal of Guidance, Control, and Dynamics.* 33.5 (2010): 1429-1441.

[3] Pontani, Mauro, and Bruce A. Conway. "Particle swarm optimization applied to impulsive orbital transfers." *Acta Astronautica.* 74 (2012): 141-155.

[4] Angeline, Peter J. "Using selection to improve particle swarm optimization." *Proceedings of IEEE International Conference on Evolutionary Computation*. Vol. 89. 1998.

[5] Mark Richards and Dan Ventura. “Choosing a starting configuration for particle swarm optimization.” *Proceedings of the International Joint Conference on Neural Networks*. Vol. 3. 2004.

[6] I.M. Soboĺ. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics.* 7.4 (1967): 86–112.

ACADEMIC VITA

Matthew Honeychuck
436 Lakeside Drive
New Freedom, PA 17349
matthew.honeychuck@gmail.com

EDUCATION

Bachelor of Science Degree in Aerospace Engineering
The Pennsylvania State University, Fall 2014
Honors in Aerospace Engineering
Schreyer Honors College

EXPERIENCE

Co-op, NASA Johnson Space Center (JSC), Houston, TX

Flight Mechanics and Trajectory Design Branch
Summer 2014, Summer 2013

- Developed six degree of freedom (6-DOF) simulation for a sounding rocket test of the Maraia capsule using the Flight Analysis Simulation Tool (FAST)
- Used Copernicus trajectory optimization software to determine the performance characteristics of a new design reference mission option for Orion's Exploration Mission 2

Trajectory Operations and Analysis Branch
Spring 2014

- Used COMPASS 6-DOF simulation to perform a sensitivity analysis of vehicle touchdown points for the Orion spacecraft's Ascent Abort 2 mission

Applied Aeroscience and Computational Fluid Dynamics (CFD) Branch
Fall 2012

- Constructed surface meshes for the rotocapsule vehicle and completed CFD simulations with different vehicle configurations
- Used Direct Simulation Monte Carlo method to characterize low density flow fields around International Space Station (ISS) visiting vehicles

Cargo Integration and Operations Branch
Spring 2012

- Developed "Router Command Reference" manual- lists the name, syntax, and description of Linux commands used for configuration changes and system maintenance of the ISS Local Area Network

AWARDS AND HONORS

JSC Outstanding Co-op Award
PSU Department of Aerospace Leonhard Scholarship
National Merit Scholarship sponsored by Northrop Grumman
Sigma Gamma Tau National Aerospace Engineering Honor Society
PSU College of Engineering Dean's List