

THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF AEROSPACE ENGINEERING

Parallelized Particle Swarm Optimization for Minimum-Time Satellite Orientation Maneuvers

Sean Rich
Spring 2020

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 Advisor

Amy R. Pritchett
Professor and Head of the Department of Aerospace Engineering
Faculty Reader

*Electronic approvals are on file in the Schreyer Honors College.

Abstract

Particle swarm optimization has developed as a popular method of solution discovery for many numerical problems. The finite thrust transfer problem employs this metaheuristic algorithm for trajectory optimization. This method is computationally expensive and requires numerous runs to confidently discover a quasi-optimal solution. These major bottlenecks - execution time and a low probability of optimal solution convergence - make real-time, on-board calculations utilizing this algorithm impractical.

The research presented within this thesis studies two methods to significantly improve each of these bottlenecks. This thesis develops results that show increased algorithm performance of well over an order of magnitude, and demonstrably improves the average solution discovered. In employing these two methods in concert, this research suggests the potential feasibility of real-time particle swarm optimization techniques.

Table of Contents

List of Figures	iv
List of Tables	v
Acknowledgements	vi
Nomenclature	vii
1 Introduction and Literature Review	1
1.1 Particle Swarm Optimization	2
1.2 Finite Thrust Transfer	2
1.3 Thesis Outline	3
2 Problem Statement	4
2.1 Particle Swarm Optimization Overview	4
2.1.1 Algorithm Definition	4
2.1.2 Algorithm Challenges	5
2.2 Speedup Test Case Problem Selection	6
2.3 Finite Thrust Problem Definition	6
2.4 Canonical Units Definition	10
2.5 Finite Thrust Arc Implementation	10
2.6 Parallelization	11
2.7 Rehydration Definition	12
3 Problem Implementation	13
3.1 Overview	13
3.2 MATLAB	14
3.2.1 Integrator	14
3.2.2 Platform Specific Implementation Details	14
3.2.3 Thrust Arcs	14
3.2.4 Thrust Angles	14
3.2.5 Rehydration	14
3.3 C++ Single Threaded	15
3.3.1 Integrator	15
3.3.2 Platform Specific Implementation Details	15

3.4	C++ Parallelization	16
4	Results	17
4.1	Numerical Results	17
4.2	Rehydration Results	22
4.3	Algorithm Benchmarking	24
4.3.1	MATLAB	25
4.3.2	C++ Single Threaded	26
4.3.3	C++ Parallelization	27
5	Conclusions and Recommendations for Future Work	28
5.1	Conclusions	28
5.2	Recommendations For Future Work	29
	References	30

List of Figures

4.1	Numerically computed optimal transfer trajectory, $\beta = 2$	18
4.2	Numerically computed optimal transfer trajectory, $\beta = 10$	19
4.3	Numerically computed thrust-angle time histories for optimal $\beta = 2$ solutions . . .	20
4.4	Global Best J Value Per Iteration: 100 Particles over 50 iterations	21
4.5	Global Best J Value Per Iteration: 110 Particles over 500 iterations	22

List of Tables

4.1	Numerical results for the optimum orbital transfer for different β values	17
4.2	Rehydration results for a 25% population reset	23
4.3	Rehydration results for a 33% population reset	23
4.4	Rehydration results for a 50% population reset	23
4.5	MATLAB Numerical Results	25
4.6	C++ Single Threaded Speedup Results	26
4.7	OpenMP Speedup Results	27

Acknowledgements

The creation of this thesis involved significant support from many throughout this journey. While lots have undoubtedly helped me grow both academically and personally, there are a few I would like to express my sincere gratitude to.

First, to my thesis advisor, Dr. Melton. His subject matter expertise is obvious, yet, perhaps more importantly, so is his passion for teaching others. Whether through classes or this research, Dr. Melton consistently strives for the best for his students. Dr. Melton has been a beacon in my academic and personal undergraduate experience and I cannot thank him enough.

Second, to my mother Brenda and sister Lindsey. You two have unwaveringly supported me in every one of my undertakings for as long as I can remember. You have pushed me, loved me, and helped shape me into the person I am today. You two are my rocks. I love you both endlessly.

Lastly, to my friends. You make every day enjoyable and have shown me what happiness looks and feels like. I am so thankful to have you by my side every day; you know I will always be by yours.

Nomenclature

α_k	Penalty coefficients	h	Integration step size
\bar{J}_{best}	Average global best cost function value over multiple algorithm executions	I_{num}	Number of algorithm Iterations
$\Delta \bar{J}_{best,\%}$	Percent change of J_{avg} for rehydration executions relative to non-rehydration	J	Cost function value
ΔE	Eccentric anomaly change during t_{co}	m	Spacecraft mass
Δt_1	Thrust arc 2 duration	m_0	Initial spacecraft mass
Δt_2	Thrust arc 2 duration	m_f	Final spacecraft mass
Δt_{co}	Coasting time interval	n_{iter}	Iterations to average δ_j over
δ	Thrust pointing angle	$n_{su,st\%}$	Percent speed-up relative to C++ single threaded
$\delta_{J_{avg}}$	Average J change over previous n_{iter}	$n_{su\%}$	Percent speed-up relative to MATLAB
$\delta_{J_{crit}}$	The critical $\delta_{J_{avg}}$ value for swarm stagnation	n_{su}	$\frac{\text{MATLAB execution time}}{\text{C++ execution time}}$
μ_B	Gravitational parameter	P_{num}	Number of particles in the swarm
$\nu_0, \nu_1, \nu_2, \nu_3$	Thrust pointing angle coefficients for thrust arc 2	$P_{r,\%}$	The percentage of the population reset upon swarm stagnation
ξ	Spacecraft angular displacement from the x axis	r	Spacecraft position
$\zeta_0, \zeta_1, \zeta_2, \zeta_3$	Thrust pointing angle coefficients for thrust arc 1	R_1	Initial orbit radius
a	semi-major axis	R_2	Final orbit radius
c	Effective exhaust velocity	T	Spacecraft thrust
d_k	Error terms	t_0	Initial time
e	Eccentricity	t_f	Final time
f	True anomaly	v_θ	Transverse velocity component
		DU	One canonical distance unit
		TU	One canonical time unit

Chapter 1

Introduction and Literature Review

1.1 Particle Swarm Optimization

Since its introduction in 1995 by Kennedy and Eberhart [1], particle swarm optimization has had numerous implications in various aerospace fields and beyond. The algorithm was originally designed to simulate social behavior within a flock of birds but has since been adapted as a stochastic solution for problems requiring numerical algorithms. Particle swarm optimization consists of a set of possible solutions, referred to as particles, which then explore the solution search space due to computed velocity update parameters. The velocity change of each particle per algorithm iteration depends on three things: the particle's best historical position, the best historical solution within the entire solution set (referred to as the swarm), and the particle's current velocity. In forming the algorithm in this way, many particle swarm optimization techniques require no prior knowledge of the solution space and do not necessitate the problem to be differentiable.

Particle swarm optimization's metaheuristic nature has lent itself to the application of various problems, including structural design optimization, responsive theater maneuvers, corrosion fatigue, and path-constrained minimal time satellite reorientation [2, 3, 4, 5]. As a stochastic algorithm, however, a global optimal solution within the search space is not guaranteed for every execution. This requires multiple runs of the algorithm and increased computation. As such, significant effort has been devoted to reducing the computational time for this class of algorithms. One proposed such method is parallel computation. Various papers detail problem-specific parallel particle swarm optimization implementations and the resulting effect on computation time [6, 7, 8]. In addition to algorithm parallelization, this thesis implemented particle swarm optimization within C++. This development sought to reduce computational burden with the finer grain memory control synonymous with lower-level programming languages.

One caveat of particle swarm optimization algorithms is premature stagnation: a condition in which the swarm converges to a non-optimal solution. The attempt to maintain population diversity has been a research area of interest and has seen various approaches in stagnation determination and correction [9, 10]. This thesis introduces a new concept, dubbed *rehydration*, in which a portion of the population is randomly reset when the population meets the specified stagnation criteria. Analysis of the effect of this *rehydration* on the search for an optimal solution is explored and presented.

1.2 Finite Thrust Transfer

Impulsive thrust approximations are used to simplify the computations used in modeling orbital transfers under high-thrust assumptions. More rigorous analysis of orbital transfers, in correspondence with actual spacecraft maneuvers, requires thrust to be finite. Numerous algorithms have been proposed to model these finite thrust arcs. Grund and Pitkin proposed an iterative method for computing the optimal thrust arcs requiring the impulsive transfer trajectory as an initial solution [11]. Further literature introduced an indirect solution applying optimal control theory using nonlinear programming from an initially guessed solution [12]. Pontani and Conway then proposed a stochastic algorithm, particle swarm optimization, for the computation of the optimal transfer trajectory [13]. The problem was formulated as a system of 11 unknown parameters and modeled

with an objective function aimed at minimizing fuel consumption and solution error. Consisting of two numerical integrations and requiring lengthy computation time, this finite thrust method was thus selected as this thesis' test case for minimizing particle swarm optimization algorithm runtimes.

1.3 Thesis Outline

Chapter 2 of this thesis defines the generalized particle swarm optimization algorithm and its challenges. This chapter introduces the development of the finite thrust arc problem and the particle swarm optimization implementation designed to explore its solutions. Additionally, this chapter details proposals to reduce both the algorithm runtime and the probability of premature swarm convergence. Chapter 3 then delves into the problem-specific implementations within MATLAB and C++. Included within this chapter are details of the parallelized C++ version of the algorithm. Chapter 4 then covers the results of these implementations, including speedup comparisons and analysis of the *rehydration* method. Finally, chapter 5 draws conclusions from these results and offers recommendations for future research.

Chapter 2

Problem Statement

2.1 Particle Swarm Optimization Overview

2.1.1 Algorithm Definition

Particle swarm optimization is a class of algorithm that employs statistical methods to locate optimal solutions minimizing an objective function in the global search space. The algorithm was originally designed to model the behavior of bird flocks but has since been adapted to solve a variety of mathematical problems. Inherently metaheuristic, particle swarm optimization requires no prior knowledge of the search space.

Potential solutions for particle swarm optimization methods exist within an n -dimensional unknown parameter space. The algorithm consists of a population, or swarm, of particles initially seeded with randomly assigned parameter sets. A particle maintains both its position - an n -dimensional parameter array - and a corresponding n -dimensional velocity vector. For each iteration, the fitness of every particle's candidate solution is evaluated as defined by the objective function J . Following the iteration, every particles' position is updated with its velocity terms, and velocity vector updated according to

1. How far each of the particle's parameters differs from the global best particle ever recorded's parameters
2. How far each of the particle's parameters differs from the parameters of its historical personal best values
3. Its current velocity, where velocity refers to the rate of change of a particle's position in the search space per iteration

This process continues for the set number of iterations. The global best particle at algorithm completion is the discovered solution. Pseudocode for the algorithm is as follows

Algorithm 1 General PSO Algorithm

```

Set Lower and Upper bounds on position and velocity
for particle in Swarm do
    Assign initial position values for each particle
end for
for iteration in Num Iterations do
    for particle in Swarm do
        Evaluate objective function  $J$ 
    end for
    Record best set of parameters each particle has achieved  $\leftarrow pBest$ 
    Record best overall position as global best  $\leftarrow gBest$ 
    for particle in Swarm do
        Update particle velocity based on  $pBest$ ,  $gBest$ , and current velocity
        if  $v_i(k) < \text{velocity lower bound}$  then
             $v_i(k) = \text{velocity lower bound}$ 
        else if  $v_i(k) > \text{velocity upper bound}$  then
             $v_i(k) = \text{velocity upper bound}$ 
        end if
        Update particle Position
        if  $p_i(k) < \text{position lower bound}$  then
             $p_i(k) = \text{position lower bound}$ 
             $v_i(k) = 0$ 
        else if  $p_i(k) > \text{position upper bound}$  then
             $p_i(k) = \text{position upper bound}$ 
             $v_i(k) = 0$ 
        end if
    end for
end for
Use  $gBest$  as solution

```

2.1.2 Algorithm Challenges

Premature Swarm Convergence

Particle swarm optimization imposes no requirements on the solution search space. Thus, a variety of local minima may exist, potentially leading to premature swarm convergence on non-optimal solutions. As such, globally optimal solutions are never guaranteed. This requires running the algorithm many times to increase the probability of finding a near-optimum parameter set. This thesis studies a proposed method to limit this premature stagnation.

Long Computation Time

Particle swarm optimization methods require the evaluation of many candidate solutions throughout the algorithm's duration. Computational complexity in the solution evaluation step is thus amplified, leading to numerically expensive algorithms with long execution durations. Additionally, the many runs required for probabilistic quasi-optimal solution discovery multiplies the overall algorithm runtime. This limits the use of these stochastic methods in time-sensitive calculations. Thus, minimizing algorithm execution time is paramount, and is chosen as a core study of this paper.

2.2 Speedup Test Case Problem Selection

To gauge the effectiveness of execution time minimization methods a computationally complex problem is necessary. A problem with larger runtimes allows for the effective benchmarking of these reduction strategies, and presents a non-trivial application of the proposed techniques.

These criteria make the finite thrust arc problem an apt test case for speedup benchmarking. Particles within this problem require lots of execution time for two main reasons

1. Each particle requires two separate numerical integrations, one for each thrust arc
2. Numerical integrations for many potential solutions within the global search space do not converge. This requires drastically reduced integration step sizes and massively increases computational complexity.

Additionally, this problem consists of a non-trivial 11 unknowns. This broad 11 dimensional search space necessitates a comparatively large population in the search for optimal values. The combination of complex solution evaluation and large swarm sizes make the problem explored within this thesis sufficient for evaluating potential speedup methods.

2.3 Finite Thrust Problem Definition

This thesis analyzes the optimal finite thrust transfer problem between two circular orbits as a benchmark case for proposed particle swarm optimization improvements. The development of this problem formulation is derived from [13].

This transfer is defined with respect to an initial circular orbit of radius R_1 and a final circular orbit of radius R_2 , subject to the conditions $R_1 > R_2$, where the parameter $\beta = R_2/R_1 > 1$. The inertial reference frame is centered at the attracting body. The corresponding coordinate frame definition is as follows: The x axis is aligned with the spacecraft at the initial time t_0 and the y axis located in the orbital plane. The z axis is determined by the right hand rule of these two basis axes. Within this problem v_r denotes the spacecraft radial velocity, v_θ the horizontal component of velocity, r the radius, and ζ the angular displacement from the x axis. The gravitational parameter

of the attracting body is μ_B .

Initial conditions at time t_0 are given by

$$v_r(t_0) = 0 \quad v_\theta(t_0) = \sqrt{\mu_B/R_1} \quad r(t_0) = R_1 \quad \xi(t_0) = 0 \quad (2.1)$$

And final conditions given by

$$v_r(t_f) = 0 \quad v_\theta(t_f) = \sqrt{\mu_B/R_2} \quad r(t_f) = R_2 \quad (2.2)$$

The problem assumes a successive transfer trajectory of an initial thrust arc, coasting arc, and second thrust arc. Optimization of this problem corresponds to determining the thrust pointing angle function requiring minimum propellant consumption subject to the constraints of Eq. (2.2).

Additional assumptions are made to simplify the analysis:

1. Throughout the duration of each thrust arc maximum thrust is generated
2. Each thrust arc pointing angle is represented as a third-degree polynomial as a function of time.

T and c are used to represent the spacecraft thrust level and effective exhaust velocity respectively. The previous assumptions indicate that the thrust-to-mass ratio has the form

$$\frac{T}{m} = \begin{cases} \frac{T}{m_0 - \frac{T}{c}t} = \frac{cn_0}{c - n_0t} & 0 \leq t \leq t_1 \\ 0 & t_1 \leq t \leq t_2 \\ \frac{T}{m_0 - \frac{T}{c}(t_1 + t - t_2)} = \frac{cn_0}{c - n_0(t_1 + t - t_2)} & t_2 \leq t \leq t_f \end{cases} \quad (2.3)$$

m_0 is the initial spacecraft mass and n_0 is the initial thrust to mass ratio at t_0 . The state space equations of motion for the spacecraft are

$$\dot{v} = -\frac{\mu_B - rv_\theta^2}{r^2} + \frac{T}{m} \sin \delta \quad (2.4)$$

$$\dot{v}_\theta = -\frac{v_r v_\theta}{r} + \frac{T}{m} \cos \delta \quad (2.5)$$

$$\dot{r} = v_r \quad (2.6)$$

$$\dot{\xi} = \frac{v_\theta}{r} \quad (2.7)$$

where $\frac{T}{m}$ is given in Eq. (2.3) and δ the thrust pointing angle represented by a third-order polynomial of time. The state vector used in this problem is $[x_1 \ x_2 \ x_3 \ x_4]^T = [v_r \ v_\theta \ r \ \xi]^T$.

The thrust pointing angle δ is defined as

$$\delta = \begin{cases} \zeta_0 + \zeta_1 t + \zeta_2 t^2 + \zeta_3 t^3 & 0 \leq t \leq t_1 \\ \nu_0 + \nu_1(t - t_2) + \nu_2(t - t_2)^2 + \nu_3(t - t_2)^3 & t_2 \leq t \leq t_f \end{cases} \quad (2.8)$$

The optimum thrust pointing angle coefficients $\{\zeta_0 \ \zeta_1 \ \zeta_2 \ \zeta_3\}$ and $\{\nu_0 \ \nu_1 \ \nu_2 \ \nu_3\}$ are parameters determined by particle swarm optimization.

During the coasting arc the problem consists of a Keplerian orbit. As such, the semimajor axis a and eccentricity e of the coasting arc can be computed as

$$a = \frac{\mu_B r_1}{2\mu_B - r_1(v_{r1}^2 + v_{\theta1}^2)} \quad (2.9)$$

$$e = \sqrt{1 - \frac{r_1^2 v_{\theta1}^2}{\mu_B a}} \quad (2.10)$$

Provided that the orbit is elliptic ($a > 0$) the true anomaly at $t_1(f_1)$ can be computed as

$$\sin f_1 = \frac{v_{r1}}{e} \sqrt{\frac{a(1-e^2)}{\mu_B}} \quad \text{and} \quad \cos f_1 = \frac{v_{\theta1}}{e} \sqrt{\frac{a(1-e^2)}{\mu_B}} - \frac{1}{e} \quad (2.11)$$

and the eccentric anomaly E_1 as

$$\sin E_1 = \frac{\sin f_1 \sqrt{1-e^2}}{1+e \cos f_1} \quad \text{and} \quad \cos E_1 = \frac{\cos f_1 + e}{1+e \cos f_1} \quad (2.12)$$

The PSO parameter ΔE represents the variation in eccentric anomaly throughout the coasting arc. Therefore, the eccentric anomaly at t_2 is $E_2 = E_1 + \Delta E$. True anomaly can thus be determined utilizing the results of Eq. (2.12)

$$\sin f_2 = \frac{\sin E_2 \sqrt{1-e^2}}{1-e \cos E_2} \quad \text{and} \quad \cos f_2 = \frac{\cos E_2 - e}{1-e \cos E_2} \quad (2.13)$$

Furthermore, the coasting time interval t_{co} can be calculated through Kepler's law as

$$t_{co} \triangleq t_2 - t_1 = \sqrt{\frac{a^3}{\mu_B}} [E_2 - E_1 - e(\sin E_2 - \sin E_1)] \quad (2.14)$$

The results from Eqs. (2.9, 2.10, and 2.13) provide the initial conditions required to numerically integrate the spacecraft equations of motion for the second thrust arc beginning at time t_2 . These initial conditions for the second thrust arc are computed as

$$v_{r2} = \sqrt{\frac{\mu_B}{a(1-e^2)}} e \sin f_2 \quad (2.15)$$

$$v_{\theta 2} = \sqrt{\frac{\mu_B}{a(1-e^2)}}(1 + e \cos f_2) \quad (2.16)$$

$$r_2 = \frac{a(1-e^2)}{1 + e \cos f_2} \quad (2.17)$$

$$\xi_2 = \xi_1 + (f_2 - f_1) \quad (2.18)$$

Integrating the second thrust arc with these initial conditions over the given time duration yields the final orbital characteristics.

This system depends on the eight coefficients representing the thrust pointing angles of the first and second thrust arcs, $\{\zeta_0 \ \zeta_1 \ \zeta_2 \ \zeta_3\}$ and $\{\nu_0 \ \nu_1 \ \nu_2 \ \nu_3\}$ respectively, and the time intervals for the first thrust arc $\Delta t_1 \triangleq t_1$, the coasting arc Δt_{co} , and the second thrust arc $\Delta t_2 \triangleq t_f - t_2$. The selection of these 11 unknowns is sought to minimize the objective function

$$J = \Delta t_1 + \Delta t_2 \quad (2.19)$$

Optimizing the unknown coefficients to minimize J corresponds to the minimization of propellant consumption. Minimizing propellant consumption results in a maximization of the final-to-initial mass ratio given by

$$\frac{m_f}{m_0} = \frac{m_0 - \frac{T}{c}\Delta t_1 - \frac{T}{c}\Delta t_2}{m_0} = 1 - \frac{n_0}{c}(\Delta t_1 + \Delta t_2) \quad (2.20)$$

The coasting arc is required to be elliptic, therefore any particles with $a \leq 0$ are assigned an infinite objective function value.

Since Δt_{co} can be computed using ΔE , the eccentric anomaly variation replaces the coasting time interval in the problem's 11 unknown parameters. Each particle in the swarm thus consists of the following

$$\chi = [\zeta_0 \ \zeta_1 \ \zeta_2 \ \zeta_3 \ \nu_0 \ \nu_1 \ \nu_2 \ \nu_3 \ \Delta t_1 \ \Delta E \ \Delta t_2]^T \quad (2.21)$$

To enforce the constraints set forth in Eq. (2.2) three penalty terms are added to the objective function Eq. (2.19).

$$\tilde{J} = \Delta t_1 + \Delta t_2 + \sum_{k=1}^3 \alpha_k |d_k| \quad (2.22)$$

where

$$d_1 = v_r(t_f) \quad d_2 = v_\theta(t_f) - \sqrt{\frac{\mu_B}{R_2}} \quad d_3 = r(t_f) - R_2 \quad (2.23)$$

A maximum acceptable error of 10^{-3} is used and the α_k coefficients assigned as

$$\alpha_k = \begin{cases} 100 & |d_k| > 10^{-3} \\ 0 & |d_k| < 10^{-3} \end{cases} \quad (2.24)$$

2.4 Canonical Units Definition

This problem employs a canonical set of units to simplify the analysis. One distance unit (DU) is defined as the radius of the initial orbit, and one time unit (TU) defined such that $\mu_B = 1 \frac{DU^3}{TU^2}$. The unknown coefficients are thus sought within the following ranges

$$\begin{aligned} 0 \text{ TU} \leq t_1 \leq 3 \text{ TU} \quad 0 \leq \Delta E \leq 2\pi \quad 0 \text{ TU} \leq \Delta t_2 \leq 3 \text{ TU} \\ -1 \leq \xi_k \leq 1 \quad -1 \leq \nu_k \leq 1 \quad (k = 0, 1, 2, 3) \end{aligned} \quad (2.25)$$

The effective exhaust velocity c is set to $0.5 \frac{DU}{TU^2}$ and the initial thrust-to-mass ratio set to $0.2 \frac{DU}{TU^2}$.

2.5 Finite Thrust Arc Implementation

The finite thrust arc problem explored within this thesis is implemented using the 11 unknown parameters in Eq. (2.21) and corresponding objective function Eq. (2.22). Position and velocity bounds for the particle swarm optimization algorithm are given by Eq. (2.25). Particle position updating uses the standard form

$$p_i = p_i + v_i \quad (2.26)$$

i ranges from 1 to the 11 unknowns.

Velocity updating is implemented as

$$v_i = v_i c_i + c_c(pBest_i - p_i) + c_s(gBest - p_i) \quad (2.27)$$

where the three accelerator coefficients are defined as

$$c_i = \frac{(1 + rand(0, 1))}{2} \quad c_c = 1.49445(rand(0, 1)) \quad c_s = 1.49445(rand(0, 1)) \quad (2.28)$$

and $rand(0, 1)$ is a uniformly distributed random number generated between 0 and 1.

A detailed look at the main portion of the PSO pseudocode is shown below

Algorithm 2 Main Finite Thrust Transfer PSO Algorithm

```

for iteration in Num Iterations do
  for particle in Swarm do
    Numerically integrate first thrust arc using state-space equations 2.4, 2.5, 2.6, and 2.7
    Compute initial conditions for second thrust arc using equations 2.9, 2.10, and 2.13
    Numerical integrate second thrust arc using state-space equations
    Evaluate  $\tilde{J}$  (2.22) numerically integrated second thrust arc values and penalty terms 2.23
    and penalty coefficients 2.24
  end for
  for particle in Swarm do
    Update particle velocity with equation 2.27
    Check velocity bounds
    Update particle Position with equation 2.26
    Check position bounds
  end for
end for
Use  $gBest$  as solution

```

2.6 Parallelization

The conditions referenced within section 2.2 make the finite thrust transfer problem a good candidate for parallelization. Within each iteration, each particle is evaluated for its value of the objective function J . This contains the computationally heavy numerical integrations. Within this section of the algorithm each particle is evaluated independently. As such, parallel processing can be used to evaluate multiple particles simultaneously. Only after each iteration completes and position and velocity updating occurs do the particles become dependent on each others' results. Since the swarm updating portion of the algorithm is not computationally expensive, a large potential benefit can be gained from parallelizing the inter-iteration computations. A simplified version of the parallelized algorithm is shown below.

Algorithm 3 Simplified parallel PSO Pseudocode

```

for iteration in Num Iterations do
  for particle in Swarm do
    Evaluate objective function  $J$ 
  end for
  Update particle velocity based on  $pBest$ ,  $gBest$ , and current velocity
  Update particle position
end for

```

$\left. \vphantom{\begin{array}{l} \text{Evaluate objective function } J \\ \text{end for} \end{array}} \right\} \text{ in parallel}$

2.7 Rehydration Definition

This thesis also considers the problem of limiting swarm stagnation as mentioned in section (2.1.2). *Rehydration* is this thesis' term for the random resetting of a percentage of the population under premature convergence. This method seeks to expand the search space explored throughout the duration of the algorithm.

Certain stagnation criteria are defined to aid in stagnation determination. To quantify population diversity, the algorithm measures the average percentage change of J , deemed δ_{Javg} , over a pre-defined previous number of iterations, η_{iter} . The critical δ_{Javg} value for stagnation is pre-defined as δ_{Jcrit} . If $\delta_{Javg} < \delta_{Jcrit}$, the swarm is considered stagnated and a portion of the population, $P_{r,\%}$, is randomly reset. This resetting does not affect the particle's knowledge of its historical best set of parameters, nor its knowledge of the historically computed global best solution for a given run.

Chapter 3

Problem Implementation

3.1 Overview

The finite thrust arc problem consists of 11 unknowns and requires two numerical integrations for each solution candidate within the particle swarm. Additionally, it necessitates very small absolute and relative tolerances when integrated with variable step-size integrators. These components serve to make this problem a suitable benchmark for the performance of different implementations of the particle swarm optimization algorithm.

To maintain consistency between different implementation targets, the numerical integration algorithm, absolute and relative tolerance values, and minimum step size constraints are held constant. All implementations employ a fifth order Runge-Kutta Dormand-Prince variable step-size integrator throughout both thrust arcs. The tolerances are defined as

$$AbsTol = 10^{-9} \quad \text{and} \quad RelTol = 10^{-9} \quad (3.1)$$

And the minimum step size set to the MATLAB default of

$$h_{min} = 7.105427 * 10^{-15} \quad (3.2)$$

Execution time is measured as wall-clock time and is computed from swarm initialization to algorithm completion. File operations are not included within this measurement.

The problem is benchmarked using the $\beta = 2$ case with various swarm sizes and iteration numbers. Results from this benchmarking can be seen in Tables (4.5, 4.6, 4.7). Unless otherwise specified, all data within this thesis is computed as the average of 30 algorithm executions held constant at the given parameters. All benchmarking is run without any *rehydration* methods. Numerical

analysis is additionally performed on the $\beta = 4, 6, 8$ and 10 cases, though these solutions are not benchmarked for execution time.

3.2 MATLAB

3.2.1 Integrator

MATLAB implements a fifth-order Runge-Kutta Dormand-Prince variable step-size integrator with the native ODE45 function. ODE45 defaults to a minimum step size of h_{min} as given in Eq. (3.2). Additionally, ODE45 presents configuration options for both *AbsTol* and *RelTol*, utilizing the larger of the two values. Integrations for both thrust arcs are configured with the values given in Eq. (3.1).

3.2.2 Platform Specific Implementation Details

As a stochastic algorithm, particle swarm optimization requires uniformly distributed random numbers to update particle velocities and explore the search space. MATLAB provides a `rand` function to generate uniformly distributed psuedo-random numbers between 0 and 1. These numbers are used to generate the initial swarm and compute velocity accelerator coefficients as given by Eq. (2.28). Solutions are computed using the default 16 digits of precision. Execution time is measured using the standard MATLAB functions `tic` and `toc`.

3.2.3 Thrust Arcs

Further analysis of the numerically determined optimal thrust arcs is computed within MATLAB. The transfer is broken down into its three distinct components: the first thrust arc, the coasting arc, and the second thrust arc. A set of r and θ values throughout these three durations are joined together for graphically modeling the given particle's transfer solution. Results of these computations can be seen in Figs. (4.1, 4.2).

3.2.4 Thrust Angles

Analysis of the finite thrust transfer problem yields the optimal thrust-pointing-angle time history for a given transfer. From Eq. (2.8) the thrust angle δ is computed for both the first and second thrust arcs. MATLAB is then used to graphically model the thrust angle as a function of time for various quasi-optimal solutions. These results are seen in Fig. (4.3).

3.2.5 Rehydration

Analysis of the rehydration technique focuses on determining the effect the parameters η_{iter} , δ_{Jcrit} , and $P_{r,\%}$ have on the solution. For each set of these parameters, the algorithm is ran 30 times and a \bar{J}_{best} computed. This \bar{J}_{best} is then compared against a \bar{J}_{best} from the result of 300 executions of the

algorithm without the *rehydration* method. Results from these comparisons can be seen in Tables (4.2, 4.3, 4.4). All *rehydration* cases are ran with $P_{num} = 100$ and $I_{num} = 1000$ for the $\beta = 2$ case.

3.3 C++ Single Threaded

A C++ version of the main particle swarm optimization algorithm is implemented to benchmark execution time across targets. C++ provides finer-grain memory management and allows for cache optimization and faster computations. While the subject of these optimizations is beyond the scope of this thesis, they can be used to greatly speedup the execution time of numerical algorithms.

This research utilizes the C++ `Boost` libraries for numerical computations. `Boost` implements the aforementioned cache and memory optimizations and is a de-facto standard for high performance C++.

3.3.1 Integrator

`Boost` provides `odeint`, a library for solving ordinary differential equations. Within `odeint` there exists a fifth-order Runge-Kutta Dormand-Prince variable step-size integrator. This implements the same algorithm as ODE45 and thus was chosen as the integrator for both finite thrust arcs. The method exposes references to the system state which is updated and overwritten at each successive time step. Absolute and relative tolerance options are provided, and are set to the values specified in Eq. (3.1).

By default `odeint` does not provide any minimum step size checks nor options to set h_{min} . As such, a custom integration observer is implemented to compute and limit step size in accordance with Eq. (3.2). This observer evaluates solution time at successive steps in the integration process and computes their delta for h_{min} regulation.

3.3.2 Platform Specific Implementation Details

To reduce computational complexity and integrate with `Boost` libraries, all 2-D arrays are flattened to their one-dimensional counterparts. The corresponding one-dimensional index is computed as

$$index = r(w + y) \quad (3.3)$$

where r is the zero-based two-dimensional row number, y the zero-based two-dimensional column number, and w the two-dimensional array width. All flattened one-dimensional arrays are represented with double precision, dynamically allocated, contiguous blocks of memory.

The C++ standard library provides a `rand` function for generating psuedo-random numbers. To ensure randomness between successive executions, the `rand` function is seeded with the number of seconds since the Epoch at the start of the program. This function returns an integer between 0

and `RAND_MAX`, where `RAND_MAX` is architecture specific. For use in particle swarm optimization, this number is cast as a double precision floating point number and bound between 0 and 1. This creates the uniformly distributed random number generator required for the algorithm.

Program execution time is measured using successive system calls to `gettimeofday` at swarm initialization and algorithm completion. Each system call returns the number of seconds and microseconds since the Epoch, enabling timing accuracy of up to 10^{-6} seconds.

3.4 C++ Parallelization

To reduce complexity and limit memory passing overhead, this thesis employs a shared-memory model of parallelization. In this model, memory is shared between all cores and threads on a singular node. Standard C++ provides a POSIX compliant threading system to architecture parallel code with this structure. While this offers the lowest level of program management, its implementation is arduous and error prone. As such, the OpenMP specification for creating parallelized programs is utilized.

OpenMP provides a set of compiler directives for creating multi-threaded applications. Within parallelized regions there must be no inter-thread memory dependency. Naturally, this allows OpenMP to parallelize particle evaluation as seen in algorithm (3). This research uses a simple `parallel for` directive to run four concurrent threads within this parallel region.

Chapter 4

Results

4.1 Numerical Results

Implementation of the particle swarm optimization algorithm provides a way to solve for the optimal orbital transfer between two concentric circular orbits. Table (4.1) shows the results of these computations. The ratio $(\frac{m_f}{m_0})_h$ represents the final to initial mass ratio for a Hohmann transfer between orbits, whereas $\frac{m_f}{m_0}$ is the mass ratio of the algorithm determined optimal solution.

β	Δt_1	Δt_{co}	Δt_2	J	m_f/m_0	$(m_f/m_0)_h$
2	0.671	5.229	0.411	1.082	0.567062	0.56614
4	1.044	11.834	0.442	1.487	0.405365	0.407642
6	1.178	20.006	0.413	1.59	0.3638	0.368367
8	1.25	24.87	0.403	1.652	0.339164	0.353299
10	1.282	41.08	0.365	1.647	0.34106	0.346603

Table 4.1: Numerical results for the optimum orbital transfer for different β values

For all values of β in Table (4.1) the computed final to initial mass ratio is similar to the final to initial mass ratio for a Hohmann transfer. For $\beta = 2$, this value exceeds its Hohmann counterpart. This indicates this algorithm is effective in determining optimal orbit trajectories, without requiring the impulsive thrust assumption as made in the Hohmann transfer calculations.

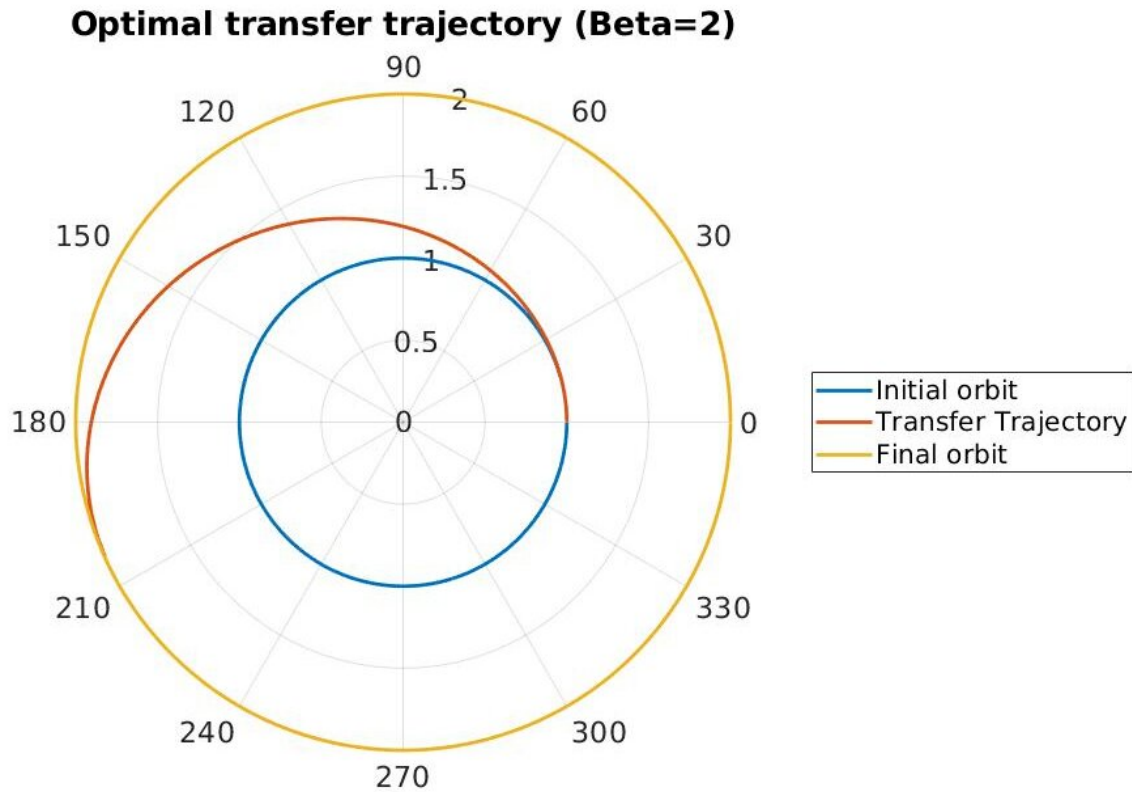
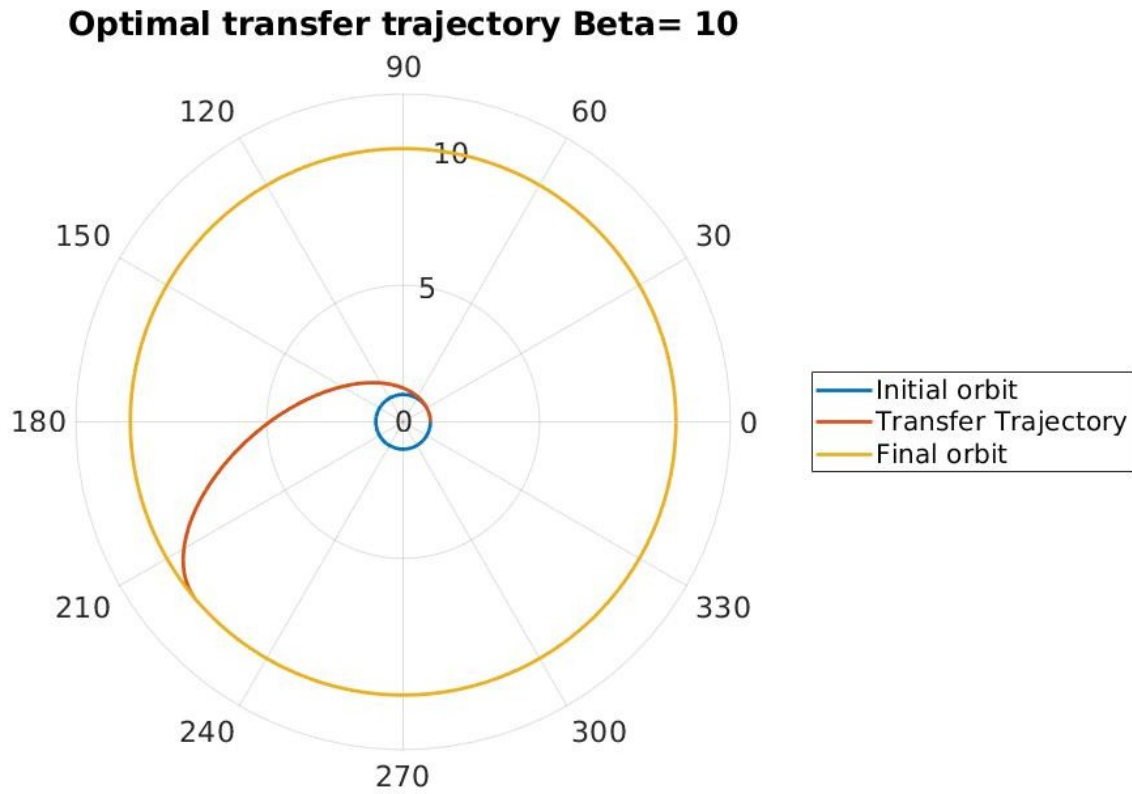


Figure 4.1: Numerically computed optimal transfer trajectory, $\beta = 2$.

Fig. (4.1) displays the numerically computed optimal transfer trajectory for $\beta = 2$. This yields insight into the optimal shape of the transfer orbit; a low eccentricity, relatively short orbital path connects the $R_2 = 2LU$ and $R_1 = 1LU$ circular orbits.



Analyzing Fig. (4.2) shows how the transfer orbit trajectory differs between two orbits with higher $\beta = \frac{R_2}{R_1}$ values. In comparison to Fig. (4.1), the transfer trajectory has a much higher eccentricity and longer duration. It can thus be graphically and logically inferred that transfers between orbits with increasingly larger β values are increasingly elliptical.

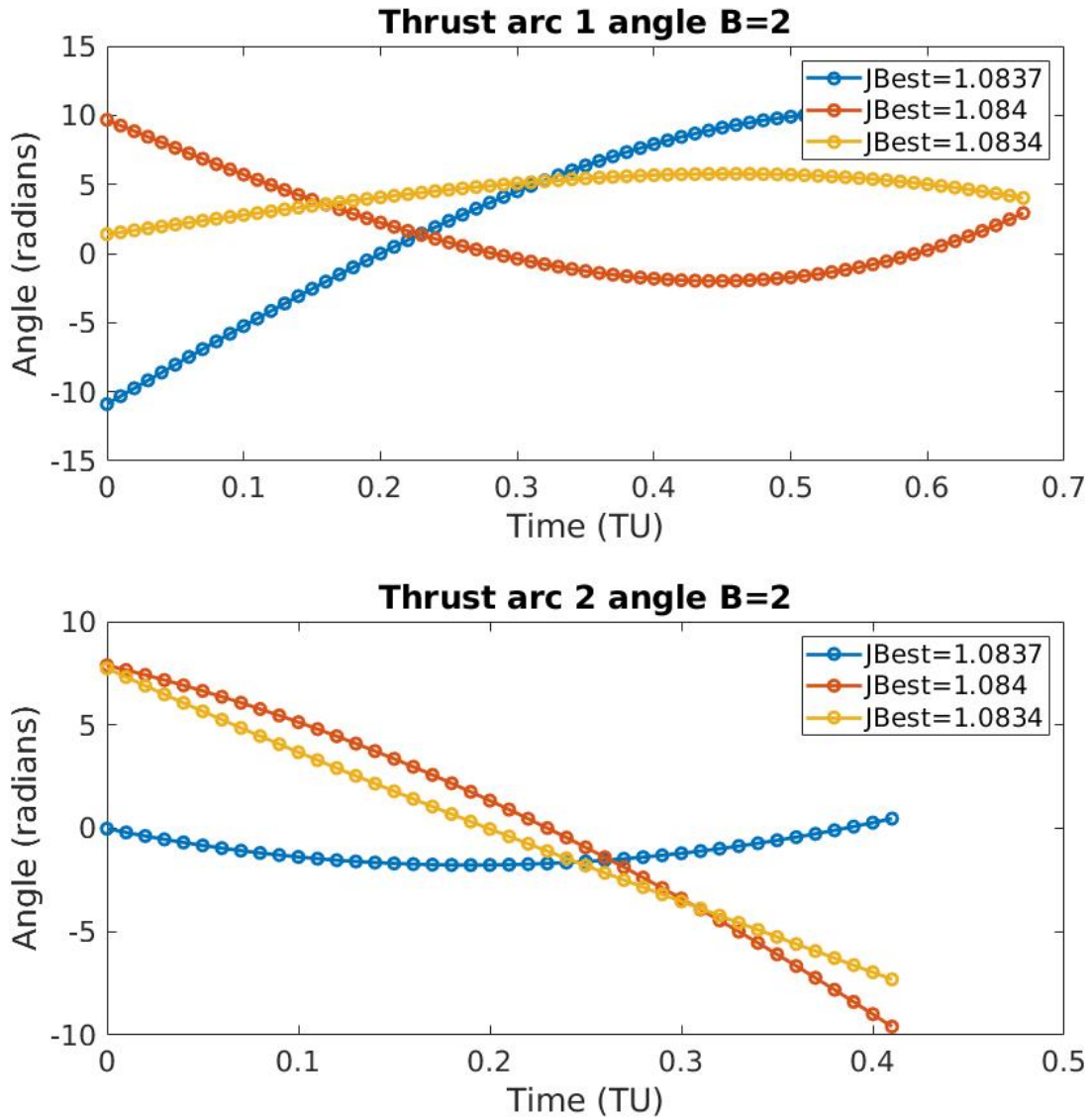


Figure 4.3: Numerically computed thrust-angle time histories for optimal $\beta = 2$ solutions

Particle swarm optimization's stochastic nature does not guarantee optimal results every run, nor does it guarantee that particles with similar cost function values represent similar solutions. Fig. (4.3) shows the thrust-pointing-angle time histories for different particles with quasi-optimal cost function values. While similar in cost function value, the thrust-pointing-angle histories vary greatly for these parameter sets. Thus, it can be observed that, within a search space of 11 unknowns, near optimal solutions may vary greatly in their characteristics.

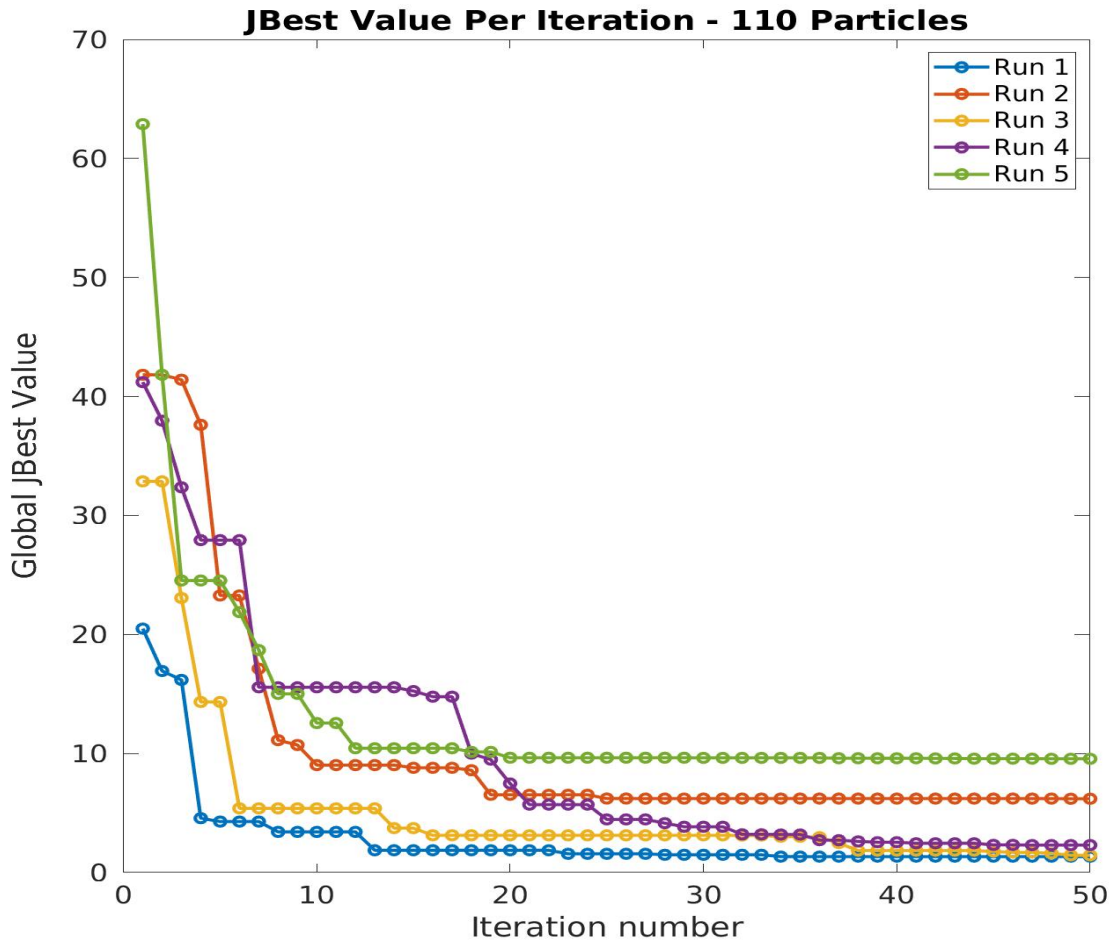


Figure 4.4: Global Best J Value Per Iteration: 100 Particles over 50 iterations

Fig. (4.4) demonstrates the evolution of the best solution found throughout the first 50 iterations of the algorithm. Two key results can be ascertained through this plot: First, the lowest found cost function value decreases rapidly over the first few iterations. Second, the swarm may appear to stagnate at a cost function value then make a jump to a better solution. This notion makes it difficult to discern when a population has truly stagnated.

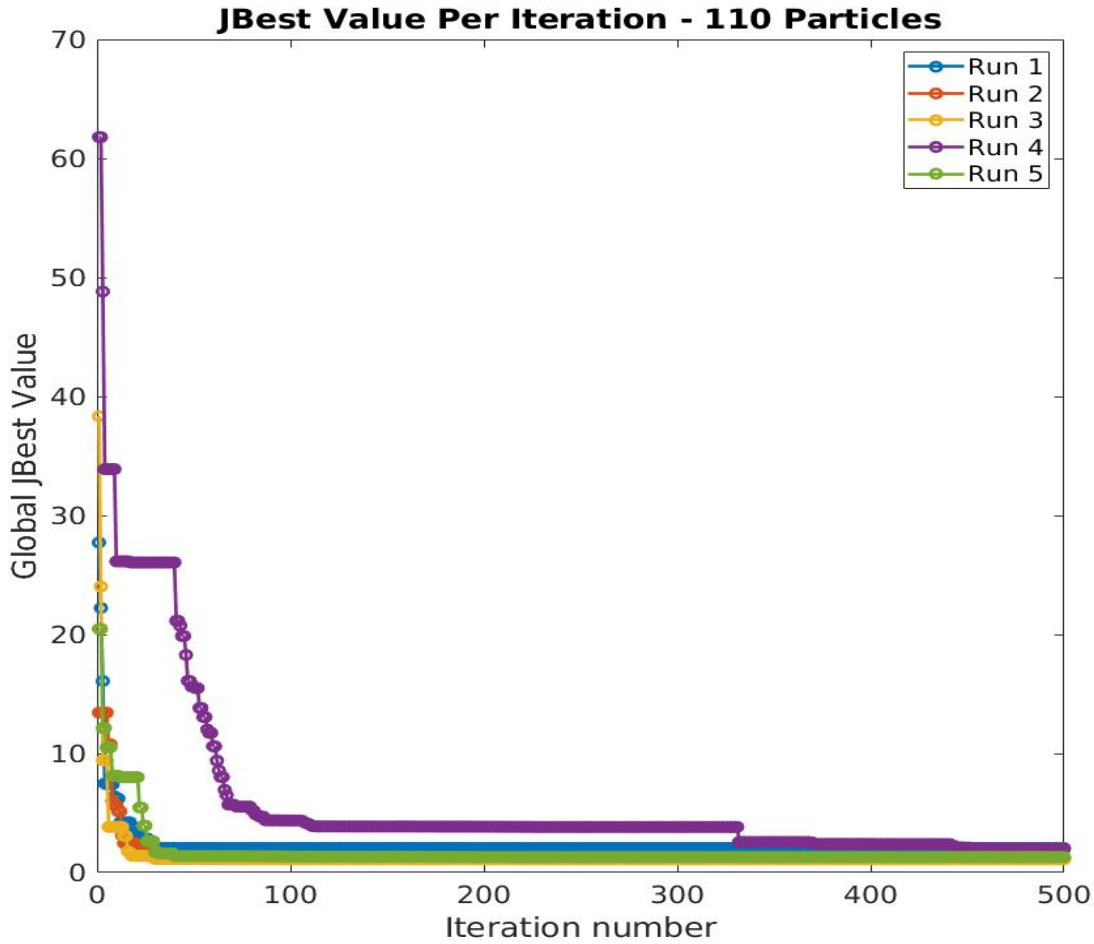


Figure 4.5: Global Best J Value Per Iteration: 110 Particles over 500 iterations

Fig. (4.5) showcases the evolution of global best cost function values over a larger range of iterations. This figure reinforces the observations from Fig. (4.5). The rapid decrease in cost function value is generally limited to the beginning iterations, and cost function value plateaus in the later stages of the algorithm. One notable portion is the significant decrease in the global best value for run four following over 300 iterations. This event demonstrates the inherent randomness of the particle swarm optimization algorithm, and once again illustrates how population stagnation can be difficult to predict.

4.2 Rehydration Results

Rehydration is a method proposed within this thesis that resets a portion of the swarm if the population is estimated to be stagnated. This method is designed to inject more randomness into the population to explore more of the solution search space, i.e. *rehydrate*, while simultaneously exploiting previously computed knowledge of the solution set.

Results within this section are derived from the variation of the three main *rehydration* parameters: δ_{Jcrit} , η_{iter} , and $P_{r,\%}$. Data from each combination of these parameters are presented. The quantity \bar{J}_{best} represents the average cost function value of 30 runs with the specified *rehydration* parameters. The quantity $\Delta\bar{J}_{best,\%}$ is a measure of the percent change of \bar{J}_{best} with *rehydration* compared to 300 runs of the same P_{num} and I_{num} without the reset method.

$P_{r,\%} = 25$	$\delta_{Jcrit} = .1\%$		$\delta_{Jcrit} = 1\%$		$\delta_{Jcrit} = 5\%$	
	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$
$n_{iter} = 5$	1.347	44.177%	1.54	36.179%	1.438	40.406%
$n_{iter} = 10$	1.421	41.11%	1.379	42.851%	1.594	33.941%
$n_{iter} = 20$	1.331	44.840%	1.652	31.538%	1.503	37.712%

Table 4.2: Rehydration results for a 25% population reset

Table (4.2) shows the *rehydration* results with $P_{r,\%}$ held constant at 25%. Every combination of parameters produces a major improvement over the standard particle swarm optimization algorithm, with percent improvement ranging from 31 to over 44 percent. In general, it can be observed that parameter sets with lower values of δ_{Jcrit} perform better.

$P_{r,\%} = 33$	$\delta_{Jcrit} = .1\%$		$\delta_{Jcrit} = 1\%$		$\delta_{Jcrit} = 5\%$	
	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$
$n_{iter} = 5$	1.371	43.18%	1.475	38.872%	1.412	41.484%
$n_{iter} = 10$	1.364	43.47%	1.558	35.433%	1.473	40.448%
$n_{iter} = 20$	1.412	41.48%	1.531	36.552%	1.578	34.604%

Table 4.3: Rehydration results for a 33% population reset

Data from Table (4.3) displays results with $P_{r,\%}$ set to 33%. These data indicate similarly improved algorithm performance for any parameter combination. However, just as in Table (4.2), the best results originate from parameter combinations with $\delta_{Jcrit} = .1\%$.

$P_{r,\%} = 50$	$\delta_{Jcrit} = .1\%$		$\delta_{Jcrit} = 1\%$		$\delta_{Jcrit} = 5\%$	
	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$	\bar{J}_{best}	$\Delta\bar{J}_{best,\%}$
$n_{iter} = 5$	1.58	34.52%	1.469	39.12%	1.425	40.94%
$n_{iter} = 10$	1.396	42.15%	1.306	45.88%	1.473	38.96%
$n_{iter} = 20$	1.373	43.10%	1.39	42.40%	1.33	44.88%

Table 4.4: Rehydration results for a 50% population reset

Table (4.4) showcases *rehydration* performance with $P_{r,\%}$ maintained at 50%. Once again, every combination of parameters leads to significant reduction in \bar{J}_{best} relative to the *non-rehydration* algorithm. Within these results it can be observed that higher values of η_{iter} generally give the best

performance. Unlike Tables (4.2, 4.3), the value of $\delta_{J_{crit}}$ appears to be less correlated to algorithm improvement.

4.3 Algorithm Benchmarking

Evaluation of a candidate solution to the finite thrust transfer problem requires significant computational resources. These potential solutions must be calculated for every particle throughout each iteration, leading to long execution cycles. Additionally, the inherent randomness of particle swarm optimization algorithms require them to be ran multiple times in the pursuit of a quasi-optimal solution. The combination of these factors prove execution time to be paramount, thus the benchmarking analysis done within this section.

To aid in the presentation of the benchmarking results the additional parameters $\eta_{su\%}$, η_{su} , and $\eta_{su,st\%}$ are defined. These quantities relate the speedup of different implementations to the base MATLAB version, with all parameters held constant. They are computed as

$$\eta_{su\%} = 100 * \frac{\text{MATLAB Time} - \text{Execution Time}}{\text{MATLAB Time}} \quad (4.1)$$

$$\eta_{su} = \frac{\text{MATLAB Time}}{\text{Execution Time}} \quad (4.2)$$

$$\eta_{su,st\%} = \frac{\text{C++ Single Threaded Time} - \text{Execution Time}}{\text{C++ Single Threaded Time}} \quad (4.3)$$

These parameters serve to yield an insight into how algorithm performance differs between implementations with the variation of P_{num} and I_{num} .

4.3.1 MATLAB

P_{num}	I_{num}	Time (s)
25	250	30.527
25	500	39.15
25	1000	68.88
50	250	30.527
50	500	80.4246
50	1000	141.151
100	250	95.956
100	500	152.22
100	1000	306.732
150	250	161.3
150	500	195.87
150	1000	388.03
200	250	157.061
200	500	251.575
200	1000	457.55

Table 4.5: MATLAB Numerical Results

Table (4.5) presents base MATLAB implementation data throughout the range of P_{num} and I_{num} to be benchmarked. Execution time increases rather linearly with iteration number, but scales slower as P_{num} grows. Results within this data set form the basis of the comparisons presented within Tables (4.6, 4.7).

4.3.2 C++ Single Threaded

P_{num}	I_{num}	Time (s)	$n_{su}\%$	n_{su}
25	250	1.732	94.32633406	17.62528868
25	500	2.39	93.89527458	16.38075314
25	1000	3.92	94.30894309	17.57142857
50	250	2.96	90.30366561	10.31317568
50	500	3.26	95.94651388	24.6701227
50	1000	5.695	95.96531374	24.78507463
100	250	6.659	93.060361	14.40997147
100	500	7.21	95.2634345	21.11234397
100	1000	11.83	96.14321297	25.92831784
150	250	7.26	95.49907006	22.21763085
150	500	12.64	93.54674018	15.4960443
150	1000	16.68	95.7013633	23.26318945
200	250	7.77	95.05287754	20.21377091
200	500	11.423	95.45940574	22.02354898
200	1000	21.06	95.39722435	21.72602089

Table 4.6: C++ Single Threaded Speedup Results

Data within Table (4.6) illustrate how the single threaded C++ implementation of the particle swarm optimization algorithm compares to the MATLAB version. It is evident that the C++ version is clearly more performant, topping out at 96.1% greater efficiency and 29.1 times faster than its MATLAB counterpart. An analysis of these results indicates that, in general, the improvement differential becomes greater as I_{num} increases.

4.3.3 C++ Parallelization

P_{num}	I_{num}	Time (s)	$n_{su}\%$	$n_{su,st}\%$	n_{su}
25	250	1.78	94.16909621	-2.771362587	17.15
25	500	3.4	91.31545338	-42.25941423	11.51470588
25	1000	5.23	92.40708479	-33.41836735	13.17017208
50	250	2.81	90.7950339	5.067567568	10.86370107
50	500	4.77	94.06897889	-46.3190184	16.86050314
50	1000	8.51	93.9709956	-49.42932397	16.58648649
100	250	5.71	94.04935595	14.2513891	16.80490368
100	500	6.6	95.66417028	8.460471567	23.06363636
100	100	11.73	96.17581472	0.8453085376	26.14936061
150	250	4.92	96.94978301	32.23140496	32.78455285
150	500	9.77	95.01199775	22.7056962	20.04810645
150	1000	13.93	96.41007139	16.48681055	27.85570711
200	250	5.88	96.25623166	24.32432432	26.71105442
200	500	10.46	95.84219418	8.4303598	24.05114723
200	1000	22.42	95.09998907	-6.457739791	20.40811775

Table 4.7: OpenMP Speedup Results

Table (4.7) displays the speedup of the OpenMP implementation relative to the base C++ and MATLAB versions. At peak efficiency, this implementation achieves a 96.49% speedup at 32.78 times faster than the MATLAB execution. However, under certain conditions, the parallelized version of the algorithm is slower than its single threaded equivalent. Analysis of the data suggests this occurs in cases with a lower P_{num} and higher I_{num} .

The reasoning for this is likely two-fold: First, parallelization improves the algorithm during early iterations, where the numerical integrations for many particles of the population do not converge, requiring greater computation. In these instances, the improvements from parallelizing the particle evaluation step overcome the multi-threaded overhead for the iteration. Throughout the later iterations of the algorithm, where most numerical integrations converge quickly, the multi-threaded overhead outweighs the parallelization improvements. Second, as P_{num} decreases, so does the benefit arising from parallelizing particle evaluation. Thread setup overhead, however, remains constant independent of P_{num} , leading to net decreased efficiency. One potential way to maintain increased efficiency throughout may be to parallelize the process in early iterations and run the algorithm single threaded in later stages.

Chapter 5

Conclusions and Recommendations for Future Work

5.1 Conclusions

The finite thrust transfer problem models orbital transfers without impulsive velocity change assumptions. As such, it provides higher fidelity solutions for minimum propellant consumption maneuvers. An analysis of this problem can direct thrust angles on a currently orbiting spacecraft throughout the transfer arc. However, this algorithm requires significant time to confidently discover a quasi-optimal solution. In many cases, the necessity of determining optimal transfer parameters has minimal advanced notice. Thus, reducing the algorithm runtime and increasing the probability of optimal solution discovery is fundamental to enabling time-sensitive results. Significant improvements in these bottlenecks could potentially enable on-board computers to calculate these transfer parameters autonomously. This thesis explores improvements to both of these barriers through algorithm benchmarking and a swarm stagnation prevention technique deemed *rehydration*.

While the particle swarm optimization algorithm runtime is inherently constrained by the calculations it entails, optimizing memory transfer, caching, and computation can potentially speed up its execution. To explore this possibility, three different implementations of the algorithm were developed: single threaded MATLAB, single threaded C++, and parallel C++. Each implementation ran for the $\beta = 2$ transfer over a range of P_{num} and I_{num} values. Results clearly indicate that C++ implementations are over an order of magnitude faster, peaking at a multiple of 26 times single threaded, and 32 times faster parallelized. It can be observed that parallelizing the implementation improves efficiency throughout the beginning portion of the algorithm, but has overhead outweighing the benefits during the exploitation phase of particle swarm optimization. Toggling

parallelization during program execution may allow for peak efficiency throughout. The results from this benchmarking demonstrate a significant improvement in algorithm performance. It is not unreasonable to consider real-time trajectory optimization maneuvers using similar C++ implementations in the future.

The second problem this paper evaluates is increasing the probability of quasi-optimal solution discovery. As a stochastic algorithm, particle swarm optimization is prone to swarm stagnation, where the population prematurely converges on a non-optimal solution. *Rehydration* is an attempt to rectify this behavior, resetting some of the population upon stagnation determination to inject more randomness into the algorithm. *Rehydration* results are tabulated for a series of the method's core parameters: $P_{r,\%}$, δ_{Jcrit} , η_{iter} , and compared against the average of 300 runs sans the reset technique. For each parameter combination, the *rehydration* algorithm demonstrably improved the average solution discovered. This indicates that this method can improve the probability of quasi-optimal solution discovery, thus reducing the amount of algorithm runs necessary to find optimal transfer parameters. In combination with C++ implementations, this significantly improves the two bottlenecks preventing real-time trajectory optimization.

5.2 Recommendations For Future Work

The core of this thesis proposes methods to improve the overall effectiveness of particle swarm optimization algorithms. While applied to the finite thrust transfer problem, their breadth is much further reaching; these techniques can be used to optimize the solutions to a wide variety of problems solved by computationally intensive particle swarm methods. Future work may be done in generalizing the C++ and *rehydration* implementations and abstracting them for easy configuration and use in different problems.

Additionally, this paper's exploration of concurrent programming as an algorithm optimization method would benefit from a greater depth of research. As previously mentioned, developing an implementation with automatically toggling parallelization based on swarm convergence state could help maintain peak efficiency throughout. Other parallel implementations, namely graphical processing unit (GPU) programming, are also recommended to be considered. GPUs benefit from an immensely high thread count, enough for a thread per particle, but add increased overhead in copying memory between host and device. An analysis of the tradeoffs between the increased multi-threaded efficiency and increased memory overhead is yet to be seen.

References

- [1] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.
- [2] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. Particle swarm optimization. *AIAA Journal*, 41(8):1583–1589, 2003.
- [3] Daniel J. Showalter and Jonathan T. Black. Responsive theater maneuvers via particle swarm optimization. *Journal of Spacecraft and Rockets*, 51(6):1976–1985, 2014.
- [4] R. M. Pidaparti and S. Jayanti. Corrosion fatigue through particle swarm optimization. *AIAA Journal*, 41(6):1167–1171, 2003.
- [5] Dario Spiller, Luigi Ansalone, and Fabio Curti. Particle swarm optimization for time-optimal spacecraft reorientation with keep-out cones. *Journal of Guidance, Control, and Dynamics*, 39(2):312–325, 2016.
- [6] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. Parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *Journal of Aerospace Computing, Information, and Communication*, 3(3):123–137, 2006.
- [7] Young Ha Yoon, Jong Kuen Moon, Eun Suk Lee, Seung Jo Kim, and Jin Hee Kim. Parallel optimal design of satellite bus structures using particle swarm optimization. *48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*.
- [8] Petri Kere and Jussi Jalkanen. Parallel particle swarm-based structural optimization in a distributed grid computing environment. *48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*.
- [9] Weiyang Tong, Souma Chowdhury, and Achille Messac. Multi-domain diversity preservation to mitigate particle stagnation and enable better pareto coverage in mixed-discrete particle swarm optimization. *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*.
- [10] Souma Chowdhury, Jie Zhang, and Achille Messac. Avoiding premature convergence in a mixed-discrete particle swarm optimization (mdpso) algorithm. *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*.
- [11] EKKEHART GRUND and EDWARD T. PITKIN. Iterative method for calculating optimal finite-thrust orbit transfers. *AIAA Journal*, 10(2):221–223, 1972.

- [12] GUIDO COLASURDO. Optimal finite-thrust spacecraft trajectories. *Astrodynamics Conference*.
- [13] Mauro Pontani and Bruce A. Conway. Particle swarm optimization applied to space trajectories. *Journal of Guidance, Control, and Dynamics*, 33(5):1429–1441, 2010.