INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Francisco Bruno Dias Ribeiro da Silva

EFFICIENT ALGORITHMS FOR SOLVING OPTIMIZATION PROBLEMS IN AEROSPACE TRAJECTORIES

Final Paper (Undergraduate study)
2023

Course of Aerospace Engineering

629.783:523.3

Francisco Bruno Dias Ribeiro da Silva

EFFICIENT ALGORITHMS FOR SOLVING OPTIMIZATION PROBLEMS IN AEROSPACE TRAJECTORIES

Advisor

Prof. Dr. Maisa de Oliveira Terra (ITA)

Aerospace Engineering

SÃO JOSÉ DOS CAMPOS
INSTITUTO TECNOLÓGICO DE AERONÁUTICA

2023

Cataloging-in Publication Data
Documentation and Information Division

Dias Ribeiro da Silva, Francisco Bruno
Efficient Algorithms for Solving Optimization Problems in Aerospace Trajectories/
Francisco Bruno Dias Ribeiro da Silva.
São José dos Campos, 2023.
83f.

Final paper (Undergraduation Study) – Course of Aerospace Engineering – Instituto Tecnológico de Aeronáutica, 2023. Advisor: Prof. Dr. Maisa de Oliveira Terra.

1. Optimization. 2. Optimizers. 3. Aerospace. 4. Trajectory. I. Instituto Tecnológico de Aeronáutica. II. Efficient Algorithms for Solving Optimization Problems in Aerospace Trajectories

BIBLIOGRAPHIC REFERENCE

DIAS RIBEIRO DA SILVA, Francisco Bruno. **Efficient Algorithms for Solving Optimization Problems in Aerospace Trajectories**. 2023. 83f. Final paper (Undergraduate study). (Bachelor of Science in Aerospace Engineering) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

AUTHOR'S NAME: Francisco Bruno Dias Ribeiro da Silva
PUBLICATION TITLE: Efficient Algorithms for Solving Optimization Problems in Aerospace Trajectories.
PUBLICATION KIND/YEAR: Final paper (Undergraduate study) / 2023

_____

Francisco Bruno Dias Ribeiro da Silva
Rua H8A, 137 – Campus CTA
12228-460, São José dos Campos - SP

EFFICIENT ALGORITHMS FOR SOLVING OPTIMIZATION PROBLEMS IN AEROSPACE TRAJECTORIES

This publication was accepted as Final Work of Undergraduate Study

Francisco Bruno Dias Ribeiro da Silva

Francisco Bruno Dias Ribeiro da Silva

Author

Prof. Dr. Maisa de Oliveira Terra (ITA)

Advisor

Prof. Dr. Cristiane Aparecida Martins

Course Coordinator of Aerospace Engineering

São José dos Campos, 23 de novembro de 2023

Acknowledgments

We would like to disclose that this work was conducted without any external funding, and there are no conflicts of interest to declare.

"Two little mice fell in a bucket of cream. The first mouse quickly gave up and drowned. The second mouse wouldn't quit. He struggled so hard that eventually he churned that cream into butter and crawled out. Gentlemen, as of this moment, I am that second mouse".

Frank Abagnale

Resumo

As exigências tecnológicas do setor aeroespacial tornam-no uma das indústrias mais dispendiosas do mundo. Portanto, é de extrema importância otimizar todos os aspectos de uma missão aeroespacial, desde o projeto do hardware da espaçonave até a trajetória da missão. A otimização da trajetória geralmente requer a definição de um número finito de parâmetros. Para alguns casos, como o problema de dois corpos, é possível obter uma solução em forma fechada para as trajetórias da espaçonave. A otimização dos parâmetros, então, é uma questão de resolver algumas equações em forma fechada. No entanto, em outros casos, é necessário um otimizador para encontrar os parâmetros ideais de um voo espacial. Este trabalho analisa otimizadores comumente usados na literatura para resolver problemas relacionados a trajetórias aeroespaciais. Em seguida, aplica esses otimizadores a problemas teste e avalia o desempenho de cada um deles, comparando a solução e o número de iterações. O objetivo deste trabalho é estabelecer um padrão de quais são os melhores otimizadores para resolver problemas de trajetória na área aeroespacial e os trade-offs entre cada um deles. Descobrimos que, considerados os algoritmos investigados e as implementações disponibilizadas nas bibliotecas pesquisadas, DE, NSGA-II e GA são os melhores otimizadores para problemas de objetivo único, enquanto NSGA-II e MHACO são os melhores otimizadores para problemas de múltiplos objetivos.

Abstract

The technological demands of the aerospace field make it one of the costliest industries in the world. It is then of uttermost importance to optimize every aspect of an aerospace mission, from spaceship hardware design to the mission trajectory. Optimizing the trajectory usually requires setting a finite number of parameters. For some cases, such as the 2-body problem, we can get a closed form solution for the spaceship trajectories. Optimizing the parameters is then a matter of solving some closed-form equations. In other cases, however, an optimizer is needed to find the optimal parameters of a space flight. This work analyses commonly used optimizers in literature for solving problems related to aerospace trajectories. It then applies those optimizers to some test problems and reviews the performance of each one of them by comparing the solution found and the number of evaluations to get such solution. The aim of this work is to set a standard of what are the best optimizers to solve trajectory problems in the aerospace field and the trade-offs between each of them. We found that, considering the investigated algorithms and the implementations available in the researched libraries, DE, NSGA-II, and GA are the best optimizers for single-objective problems, while NSGA-II and MHACO are the best optimizers for multi-objective problems.

List of Figures

List of Tables

List of Abbreviations and Acronyms

ABC	Artificial Bee Colony
ASA	Adaptative Simulated Annealing
CL	Collocation
DE	Differential Algorithm
EDA	Estimation of Distribution Algorithm
GA	Genetic Algorithm
GPS	Global Positioning System
ICA	Imperialist Competitive Algorithm
IICA	Improved Imperialist Competitive Algorithm
ITA	Instituto Tecnológico de Aeronáutica
MHACO	Multi-Objective Hypervolume-Based Ant Colony Optimizer
MJD2000	Modified Julian Calendar 2000
MOEA/D	Multi-objective Evolutionary Algorithm Based on Decomposition
MOPSO	Multi-Objective Particle Swarm Optimization
MPSO	Modified Particle Swarm Optimization
NSGA-II	Nondominated Sorting Genetic Algorithm II
PSO	Particle Swarm Optimization
QL	Quasilinearization
SA	Simulated Annealing
SD	Steepest Descent
VE	Variation of Extremals
VLDE	Violation Learning Differential Evolution

Contents

1 Introduction

The aerospace industry is renowned for its remarkable achievements and groundbreaking innovations that have revolutionized human technology. Thanks to this field, we have GPS (Global Positioning System) service [1], which provides us with accurate global positioning and navigation capabilities; we have private companies, such as SpaceX, providing low latency internet access anywhere in the world using LEO (Low Earth Orbit) satellites [2]; we have the FireSat system [3], which aims to increase safety by allowing fire fighters to detect long-range forest fires earlier, reducing damage to properties and nature. All those advancements make it clear that the aerospace field is of great strategic relevance to governments and modern society. Yet, the entire space economy is only valued at 469 billion USD [4]. By comparison, Apple's market value reached 3 trillion USD on July 3, 2023 [5]. The main cause of the relatively low valuation of the space economy is the high cost of aerospace technology projects. For instance, the whole Apollo program spent roughly 25.8 billion USD, or approximately 257 billion USD when adjusted for inflation to 2020 dollars [6]. Because of those high costs, optimizing every possible variable in a space mission is of uttermost importance.

One of the variables we want to optimize in a space mission is the orbit, or flight trajectory, of the spacecraft. The orbit selection process is highly complex, involving many parameters [7]. In some cases, such as the two-body problem where only one gravitational body influences the spacecraft's motion, closed-form solutions for the spacecraft orbit exist, simplifying the optimization process. However, for more complex scenarios involving multiple gravitational bodies or other factors, such as the oblateness of the orbited celestial body, there is no closed form solution for the orbit. In those cases, it is not clear what orbit a given choice of parameters outputs. Therefore, a tool to calculate the trajectory and a method to evaluate its "quality" (according to the mission requirements) is needed. Notice it becomes much more difficult to find the optimal parameters for the mission in that case: as the influence of each parameter becomes unclear in the orbit format, we could need to simulate all possible combinations of parameters to find the one that minimizes the mission cost, which may be unfeasible. For example, Lima dos Santos [8] needed 4 parameters to optimize a two-stage solar sail trajectory. One of those parameters was an angular position α that could assume values from -90° to +90°. If we set a step of 5°, that means 37 possible values of α. Assuming a similar number of values for the other variables, we have more than a million possible states to evaluate. This means

running a costly simulation a million times. If each simulation took 10 seconds to finish using the available propagator, the optimizer would take almost 4 months to converge if running non-stop. This makes it clear we have a necessity to use efficient optimizers for this problem (and other multidimensional optimization problems in the aerospace field).

This work reviews commonly used optimizers in literature for space trajectory problems. It analyses how those optimizers work and compares them through some test problems. The comparison considers the solution found and the number of evaluations of an objective function. The aim of this work is to set a standard for the best optimizers to solve trajectory problems in the aerospace field and the trade-offs between each of them.

The remainder of this paper is organized as follows: Section 2 reviews commonly used optimizers in the literature. Section 3 reviews the theory behind optimizers and shows how the optimizers found in the previous section work. Section 4 presents the test problems. Section 5 presents the methodology used to run the test problems. Section 6 tests the optimizers against the test problems and compares their solutions and the number of evaluations of the objective function. Finally, Section 7 concludes and shares our ideas for future work.

2 Literature Review

We did a literature review of all the papers that compared different optimizers on space trajectory problems. The objective of this review was to search for similar work done before and gather their results and insights. This review was done over the following databases: IEEE Xplore [9], Google Scholar [10], and Springer Link [11]. We believe those databases suffice when searching for relevant work on this subject. The time interval we considered in this review was between January 2012 and August 2023 inclusive. We did not directly search for works before 2012, as they may be outdated, either by not considering new state of the art optimizers (the ones published after 2011) or by not taking better implementations of existing optimizers into account (for example, better parallel computing using more recent CUDA technology [12]), which may lead to an unfair comparison for some methods.

To perform this review on IEEE Xplore, we performed the following query:

Search Terms: (aerospace in "All Metadata") AND (optimization in "All Metadata") AND (optimizers in "All Metadata") OR (orbital in "All Metadata") AND (trajectory in "All Metadata").

Filters: 2012-2024.

We got 499 results. Of those, the following 8 works met our criteria, by either comparing different optimizers or citing works that do so. We did not include works that simply presented a new optimizer and showed its convergence / acceptable solution.

1. Acciarini [13] cited works by the ESA (European Space Agency) affirming that MPSO (Modified Particle Swarm Optimization), DE (Differential Evolution) and ASA (adaptative simulated annealing) are the most promising trajectory optimizers [14] and [15]. They also presented their own optimizer MHACO (Multi-Objective Hypervolume-Based Ant Colony Optimizer) and compared it to MOEA/D [16] and NSGA-II (Nondominated Sorted Genetic Algorithm-II) [17] optimizers. They concluded their optimizer converges better than the other two for their test problems.
2. Wang [18] affirmed their optimizer IICA (Improved Imperialist Competitive Algorithm) performs better than other common algorithms, such as PSA, DE and GAs (Genetic Algorithms) for solar sail spacecraft trajectory optimization.

3. Shirazi [19] compared EDAs (Estimation of Distribution Algorithms) to GAs and PSO in low-thrust trajectories corrections. They affirmed EDAs are more reliable than GAs and PSO for the problems they analyzed.
4. Samsam [20] affirms GAs are the most suitable to handle multi-objective optimization problems with constraints for optimal trajectories for spacecrafts (particularly when compared to PSO and SA).
5. Navabi [21] compares a different set of algorithms than the ones found in the works cited above. They compare SD (Steepest Descent), VE (Variation of Extremals), QL (Quasilinearization) and CL (Collocation) methods to solve low-thrust trajectory optimization problems. They concluded CL has the least error and is a more practical method than the others for the problem they tested.
6. Chai [22] presents a method called VLDE (Violation Learning Differential Evolution algorithm) and states that this method is superior to PSO, DE, GA and ABC (Artificial Bee Colony) methods for trajectory optimizations.
7. In another work, Shirazi [23] employed an ICA (Imperialist Competitive Algorithm) for solving a multi-objective high thrust acceleration and compared it to a GA. They found the ICA converged faster than the GA.
8. Shirazi [24] also analyzed a hybrid genetic simulated annealing strategy in high thrust orbital maneuvers optimization. They used GA and SA as basis for comparison and found that their hybrid approach converges faster.

To perform this review on Google Scholar, we performed the following query:

With all the words: trajectory optimization spacecraft.

Where my words occur: in the title of the article.

Date: 2012 – 2024

We got 116 results. Of these, we found and were able to review 3 new papers:

9. Shirazi [25] refers to GA, PSO and DE as the most used optimizers in aerospace trajectory problems. They also highlight that GAs are the first choice for most of the spacecraft trajectory optimization problems, perhaps due to their availability and ease of use.
10. In another work, Chai [26] reviews optimization techniques used in spacecraft flight trajectory design. They enumerate popular optimization algorithms, including GA, DE, VLDE, PSO, AC, ABC and SA for single-objective optimization, and NSGA-II, MOPSO (Multi Objective Particle Swarm

Optimization) and MOEA/D for multi-objective optimization. This work does not compare optimizers using test problems.

11. Chai [27] also motivates the use of NSGA-II in multi-objective optimization problems in comparison to other derivative-free optimization algorithms such as GA, ABC and PSO.

Some other works found on Google Scholar presented a comparison of different methods to solve aerospace trajectories, but we were not able to access them due to restrictions.

Finally, we performed the following query on Springer Link:

With all the words: optimization

Date: 2012-2024

We found 21 results, none of which refer to optimization of aerospace trajectories.

So, after performing our literature review, we found 11 works that cover the same subject of this work: review and find efficient optimizers for aerospace trajectory problems. Analyzed those works, we concluded the following:

- We can separate space trajectory problems into two kinds: single-objective and multi-objective problems. Single-objective problems have a single cost function which must be minimized. Multi-objective problems have multiple cost functions. Because there may not be such a solution that minimizes all of them at once, we search for pareto-optimal solutions (we will discuss that in detail on the next section).
- PSO, DE, GA, SA, and ABC are common algorithms that have been studied and used for some time in single-objective aerospace trajectory optimization problems [15]. Because they are easy to implement and are reliable, they are usually used as benchmark for newer state-of-the-art algorithms. Those optimizers are safe options for single-objective problems.
- For multi-objective problems the most used algorithms are NSGA-II, MOEA/D and MOPSO (Multi-Objective Particle Swarm Optimization).

That said, in hope to find efficient algorithms for space trajectory problems, we decided to include the following algorithms in our tests and comparisons:

Table 1 Chosen optimizers.

Optimizer Name	Problem Type
PSO	Single-Objective
DE	Single-Objective
GA	Single-Objective
SA	Single-Objective
ABC	Single-Objective
MOPSO	Multi-Objective
NSGA-II	Single-Objective and Multi-Objective
MOEA/D	Multi-Objective
MHACO	Multi-Objective

Other algorithms presented in the papers above were not included due to difficulty in using available source code. We refrained from implementing the optimizers ourselves to not introduce errors and thus biases in the analysis. Also, we believe the chosen list is enough for finding efficient algorithms to solve trajectory problems, both for single-objective and multi-objective problems, due to the amount of research that has been put into those optimizers.

3 Optimizers Theoretical Background

Optimizers are used to minimize (or maximize) objective functions. Often, a single objective function is not enough to describe a problem: say you want to minimize the cost of a space mission, but the found solution implies the mission will last much longer than desired. In that case, you can better describe the problem by using two objectives: the cost and the duration of the mission. Because the theory behind single-objective minimization and multi-objective minimization has a lot of differences, we will study them separately.

3.1 Single-Objective Minimization

Single-objective minimization can be summarized as follows: Find the points that provide the minimum function value of f over the constraint set Ω: $\underset{x}{\operatorname{argmin}}\{f(x) : x \in \Omega\}$. Note that this set can be empty. For example, let $f(x) = x$ and $\Omega = \mathbb{R}$. No input $x \in \mathbb{R}$ minimizes $f(x)$, as $f(x-1) < f(x)$. This happens because Ω is unbounded and $f(x)$ does not have a lower bound. In the context of space mission parameters (that is, choosing a tuple of parameters x to define some aspect of a space mission), Ω will usually be bounded. Also, having $\lim_{x \to x_0} f(x) = -\infty$ does not make sense for objective functions that have a physical meaning (such as cost or time elapsed). So, we may assume that we will always have a non-empty set as solution.

Describing the exact form of $\Omega \subseteq \mathbb{R}^n$ to a solver may be hard. For example, what if Ω is the upper surface of a unit sphere (see Figure 1)? How are we supposed to tell the solver this information? This problem becomes particularly worse as the geometry of Ω becomes less usual. To solve this problem, we define equality and inequality constraints: we define a list of equality constraints $g_i(x) = 0$ and a list of inequality constraints $h_i(x) \geq 0$.

Figure 1 Upper surface of a unit sphere.

For example, if we desire to minimize a function $f(x)$: $\Omega \to \mathbb{R}$, $x = (a, b, c)$, where Ω is the upper surface of a unit sphere, as shown in Figure 1, we can set the lower bound of x to $[-1, -1, 0]$, the upper bound to $[1, 1, 1]$ and an equality constraint $g(x) = a^2 + b^2 + c^2 - 1 = 0$. This gives the optimizer the necessary information to search over Ω. Also, the fact that $g(x)$ is smooth helps the optimizer converge to the domain whenever it evaluates points outside of it. By using lower and upper bounds, along with equality and inequality constraints, we can tell the optimizer the domain Ω over which we want to search for an optimal value. Now we must define a strategy to do it.

3.1.1 Complete Search / Grid Search

The most straightforward strategy to minimize an objective function is to evaluate all possible values in the domain and return the input that outputs the minimum value found. Considering the input is defined over a continuous domain, this is impossible. What is done instead is segmenting the domain into finite steps and evaluating the objective function at each step. For example, if the input is a $\mathbb{R}^6$ vector with lower bound [0,0,0,0,0,0] and upper bound [1,1,1,1,1,1], we can break each dimension into steps of size 0.01, so we have 101 values to evaluate in each dimension $(0, 0.01, 0.02, \dots, 1)$. This accounts for a total of $101^6 \approx 10^{12}$ evaluations. If each evaluation takes 10,000 CPU cycles (for comparison just calling a C/C++ function takes 25-250 CPU cycles depending on the number of parameters of the function [28]. Objective functions for space missions tend to perform costly computations to simulate the environment, so 10,000 CPU cycles is a much conservative value), we have a total of 10^{16} CPU cycles. Modern CPUs' speed is around 4 GHz. This means it would take an entire month to

optimize this lightweight objective function with 6 parameters. If the objective function does a lot of computations, such as running a Runge-Kutta method, it may take years to run this optimization. This problem gets worse as we increase the dimension of the problem and reduce the size of each step. That is why doing a complete search is strongly advised against, unless we have only one or two parameters to optimize, or we have a bit more parameters and the objective function runs fast. To illustrate this method, consider Figure 2.

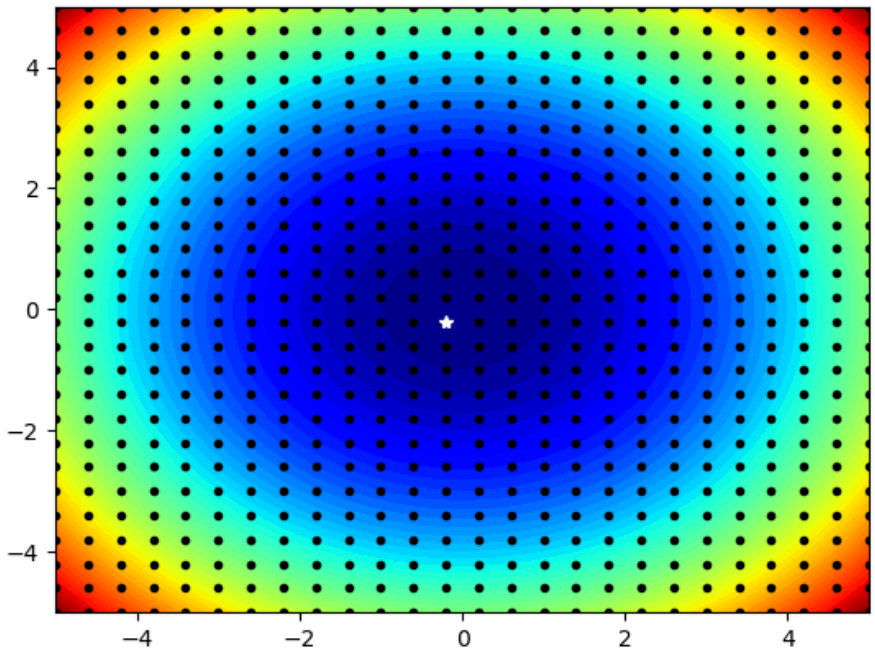

Figure 2 Complete Search over a function.

The evaluated function of the Figure 2 takes two parameters $a \in [-5, 5]$ and $b \in [-5,5]$ and returns the value $f(a, b) = a^2 + b^2$. We segmented each parameter into steps of size 0.4. Each black dot in the figure is a point where the function was evaluated. The star marks the point that outputs the minimum value found by the search. Doing that, the solution we found is $x = (-0.2, -0.2)$, which outputs $f(x) = 0.08$. Notice the actual minimum of the function is at $f(0,0) = 0$. By taking a bigger step we reduce the number of evaluations, but our solution may get further from the global minimum.

Our advice for this optimization technique is: if the problem has only a single parameter, a complete search is probably the best option. If the problem has two parameters, a complete search is still useful, specially to make plots like the one in Figure 2, where the user can easily

visualize the value of the objective function at different points. If the problem has three or more dimensions, we strongly advise against this method.

3.1.2 Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) is a heuristic optimization technique inspired by the social behavior of birds flocking or fish schooling. It was introduced by Kennedy and Eberhart in 1995 [29]. The algorithm can be described by the following sequence of steps:

1. **Initialization:** The algorithm starts with a group of random solutions called particles. Each particle represents a potential solution to the optimization problem. Each particle has a position, which represents the current solution, and a velocity, which determines how much the particle will move in the next iteration.
2. **Evaluation:** Each particle's position is evaluated using the objective function to determine its fitness.
3. **Update Best:** For each particle, if the current position is better (i.e., has a better fitness) than its previous best-known position, then this position becomes the new personal best for that particle.
4. **Update Global Best:** From all the personal best positions of the particles, the one with the best fitness is chosen as the global best position.
5. **Update Velocities and Position:** For each particle, its velocity and position are updated based on three components:
    a. Inertia: This component represents the particle's previous velocity. It ensures that the particle doesn't change its direction abruptly.
    b. Cognitive Component: This is the knowledge that the particle has from its own experience. It is the difference between the particle's personal best position and its current position.
    c. Social Component: This is the knowledge that the particle gains from its neighbors. It is the difference between the global best position and the particle's current position.

    The updated velocity is a weighted sum of these three components. The weights (often called coefficients) are parameters of the PSO algorithm and can be adjusted to change the behavior of the particles. The position of the particle is then updated by adding the new velocity to its current position.

6. **Termination:** The algorithm repeats steps 2-5 until a stopping criterion is met.

Figure 3 illustrates how the algorithm works.

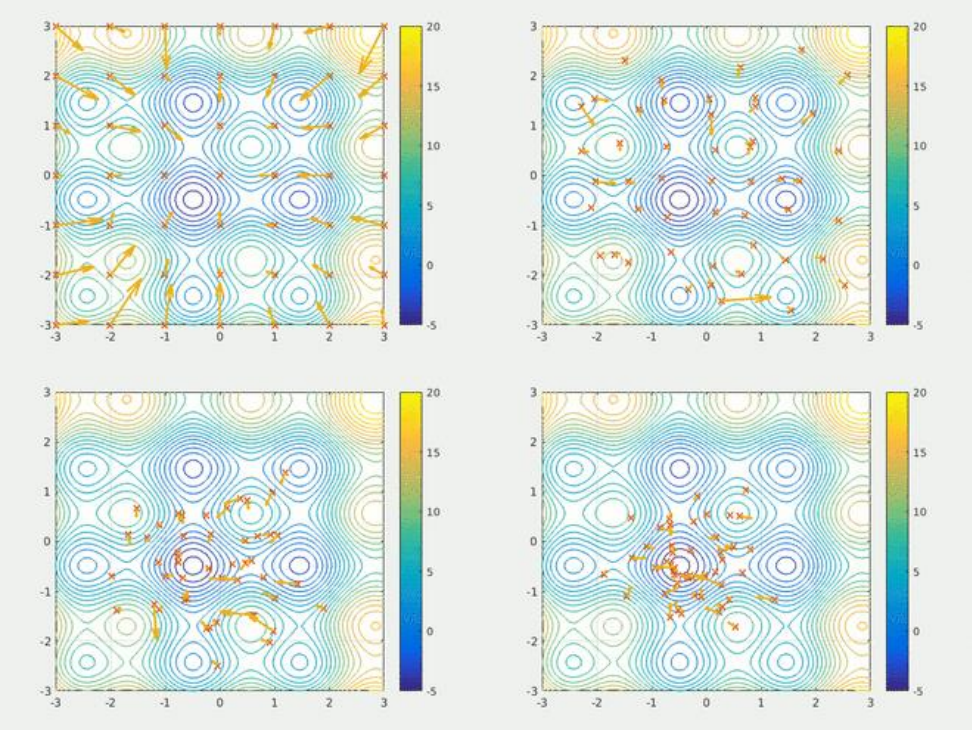

Figure 3 Illustration of PSO algorithm. Image taken from Wikipedia [30].

The algorithm starts on the top left image of Figure 3. The velocity of each point is illustrated by an arrow. As the iterations progress, the algorithm eventually converges to the global minimum.

Notice this is a heuristic approach. The balance between exploration (searching new areas of the search space) and exploitation (refining the search around the best-found solutions) is achieved through the inertia, cognitive, and social components of the algorithm. High inertia can lead to more exploration, while high cognitive and social components can lead to more exploitation. We will see how this algorithm performs in aerospace mission trajectory optimization in the results section.

The implementation of PSO we used in this paper was provided by the Python library pymoo [31] version 0.6.0.

3.1.3 Differential Evolution (DE)

Differential Evolution (DE) is a population-based optimization algorithm that belongs to the family of evolutionary algorithms. It was introduced by Storn and Price in 1997 [32]. The algorithm can be described by the following sequence of steps:

1. **Initialization:** The algorithm starts with a randomly generated population of potential solutions, called individuals or agents. Each individual represents a potential solution to the optimization problem.
2. **Mutation:** For each individual x, a trial vector is generated using the following steps:
    a. Randomly select three individuals from the population. They should be distinct from each other as well as from x. Let's call them a, b, c.
    b. Build the mutant vector $v = a + F \cdot (b - c)$, where F is a scaling parameter.
    c. Build a trial vector u the following way: for each component i, $u_i = v_i$ with CR probability and $u_i = x_i$ with $1 - CR$ probability, where CR is the crossover parameter.
3. **Selection:** If $f(u) < f(x)$, we update x to u. Otherwise, we discard the update.
4. **Termination:** The algorithm repeats steps 2-3 until a stopping criterion is met.

It is hard to illustrate this algorithm intuitively, so we won't provide a figure for it. This algorithm is also based on heuristic approaches. The mutation operation, which uses differences between randomly selected individuals ensures diversity in the population. This helps the algorithm explore the search space effectively.

The implementation of DE we used in this paper was provided by the Python library pymoo [31] version 0.6.0.

3.1.4 Genetic Algorithm (GA)

Genetic Algorithms are a subset of evolutionary algorithms inspired by the process of natural selection. The exact algorithm we will use in our analysis is a basic $(\mu + \lambda)$ genetic algorithm [31]. The algorithm can be described by the following steps:

1. **Initialization:** A starting population is sampled in the beginning.

2. **Evaluation:** Each individual is evaluated using the objective function.
3. **Survival:** It is often the core of the genetic algorithm used. For a simple single-objective genetic algorithm, the individuals can be sorted by their fitness, and survival of the fittest can be applied.
4. **Selection:** At the beginning of the recombination process, individuals need to be selected to participate in mating. Depending on the crossover, a different number of parents need to be selected. Different kinds of selections can increase the convergence of the algorithm.
5. **Crossover:** When the parents are selected, the actual mating is done. A crossover operator combines parents into one or several offspring. Commonly, problem information, such as the variable bounds, is needed to perform the mating. For more customized problems, even more information might be necessary (e.g. current generation, diversity measure of the population, …).
6. **Mutation:** It is performed after the offspring are created through the crossover. Usually, the mutation is executed with a predefined probability. This operator helps to increase the diversity in the population.
7. **Termination:** The algorithm repeats steps 2-6 until a stopping criterion is met.

Notice this algorithm is similar to Differential Evolution, as we also have the selection, crossover and mutation steps. This algorithm is illustrated by Figure 4.

Figure 4 Humanoids learned to walk via Genetic Algorithm. Image generated from [33].

In Figure 4, the yellowish humanoids are the fittest ones in each generation. It is noticeable that as the generations pass, the probability that a single humanoid walks upright increases. This happens because each generation is formed by a combination of the best agents

in the previous generation (together with some mutation). This simulation is available at https://rednuht.org/genetic_walkers/ [33].

As previously cited, the GA implementation we used in this paper was provided by the Python library pymoo [31].

3.1.5 Simulated Annealing (SA)

Simulated Annealing (SA) is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It was introduced by Scott Kirkpatrick in 1983 [34]. Annealing refers to the process where a material (like metal or glass) is heated to a high temperature and then gradually cooled to remove defects and improve the arrangement of its atoms. Similarly, SA is used to find an approximate solution to an optimization problem by iteratively exploring the solution space. The algorithm can be described by the following steps:

1. **Initialization:** Start with an arbitrary solution to the problem.
2. **Iteration:** Explore a random neighbor to the current solution. If the neighbor is fitter, accept it as the new solution. If not, accept it with $e^{-\Delta E/T}$ probability, where $\Delta E = f(neighbor) - f(curr)$ and T is the current "temperature" of the algorithm.
3. **Cooling:** Decay the value of temperature T.
4. **Termination:** Repeat steps 2-3 until a stop criterion is met.

The key feature of SA is its ability to escape local optima by accepting worse solutions with a certain probability. This probability is high at the start, when the temperature is high, and decreases as the algorithm progresses. The balance between exploration and exploitation is achieved by the cooling process: in the beginning, the algorithm explores the solution space widely, accepting many non-optimal moves. As T decreases, the algorithm becomes more conservative, refining the current best solution and exploiting the best regions of the solution space. Figure 5 illustrates this algorithm.

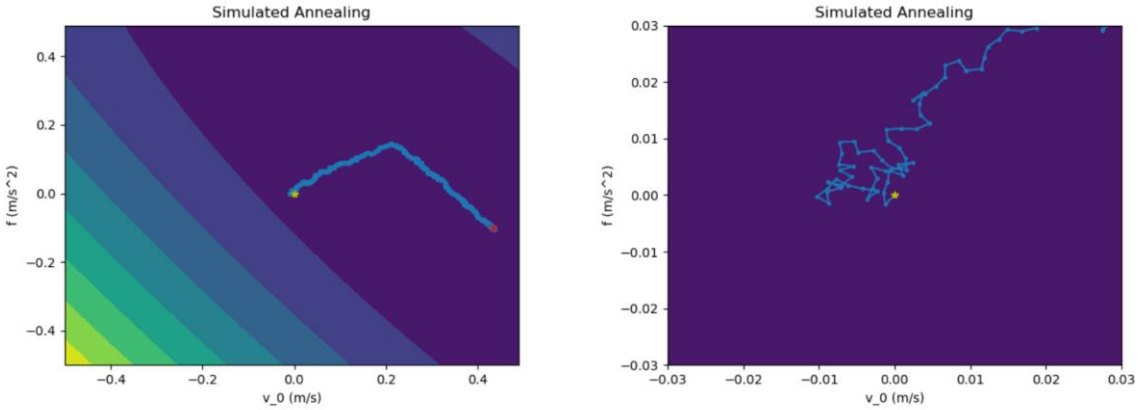

Figure 5 Simulated annealing algorithm minimizing a function.

The simulated annealing implementation we used in this paper was provided by the Python library pygmo [35] version 2.19.5.

3.1.6 Artificial Bee Colony (ABC)

Artificial Bee Colony (ABC) is an optimization algorithm based on the intelligent foraging behavior of honeybee swarms. It was introduced by Karaboğa in 2010 [36]. The algorithm simulates the foraging behavior of bees to find the optimal solution. Here is a description of the steps involved in the ABC algorithm:

1. **Initialization**: The algorithm initializes a population of solutions where each solution is referred to as a "food source." Each food source corresponds to a possible solution to the optimization problem.
2. **Employed Bees Phase:** Employed bees are associated with specific food sources, and their task is to exploit their food source. They carry out a local search near their food source and evaluate the fitness of the new solution. If the new solution has better fitness, the bee memorizes the new position and forgets the old one.
3. **Onlooker Bees Phase:** Onlooker bees watch the employed bees and choose a food source depending on the probability related to the fitness of the food source. They may also perform a local search around the selected food source to possibly find a better solution.
4. **Scout Bees Phase:** If a food source is not improved further through a predetermined number of cycles, it is abandoned, and the bee becomes a scout.

Scout bees are responsible for exploring new areas of the search space. They randomly generate new solutions without the influence of previous experience.

5. **Memorizing the Best Solution:** Throughout the foraging process, the best food source found so far is memorized.
6. **Termination:** The algorithm repeats the employed bee's phase, the onlooker bees' phase, and the scout bees' phase until a stopping criterion is met, such as a maximum number of iterations or reaching a satisfactory fitness level.

The ABC algorithm is a heuristic approach that combines local search methods, conducted by employed and onlooker bees, with global search methods, conducted by scout bees. This balance allows the algorithm to explore new areas of the search space while exploiting the best solutions found.

The ABC implementation we used in this paper was provided by the Python library pygmo [35] version 2.19.5.

3.2 Multi-Objective Minimization

When optimizing more than one objective function, there may not be such a solution that globally minimizes all the functions at the same time. In real world scenarios, it is expected for objective functions to conflict with each other. For example, minimizing the money spent on a mission usually competes with shortening the schedule. Because of that, when working with multiple objective functions, we are concerned with pareto optimal solutions; that is, solutions that cannot be improved in any of the objectives without degrading at least one of the other objectives. In other terms, $x \in \Omega$ is pareto optimal if and only if there is no other solution $y \in \Omega$ such that $f_i(y) \leq f_i(x)\ \forall\ i$ (that is, y is no worse than x in any objective function) and $\exists\ i, f_i(y) < f(x)$ (y is better than x in at least one objective function), where f_i are the objective functions.

For example, let our objective functions be $f_1(a,b) = a^2 + b^2$ and $f_2(a,b) = a + b$. Let $\Omega = [-1,1] \times [-1,1]$. f_1 has a single global minimum at $(0,0)$, so $(0,0)$ is pareto optimal, as all other solutions output a larger value for f_1. Using a similar argument, $(-1,-1)$ is also pareto optimal, as it is the only global minimum of f_2. A less trivial example are the points of the form $(-u,-u), u > 0$. All points (a,b) such that $f_1(a,b) \leq f_1(-u,-u)$ are inside a circle with center in $(0,0)$ and radius $\sqrt{2}u$. But none of those points has $f_2(a,b) < f_1(-u,-u)$. So, all points $(-u,-u)$ are pareto optimal. This can be better visualized in Figure 6.

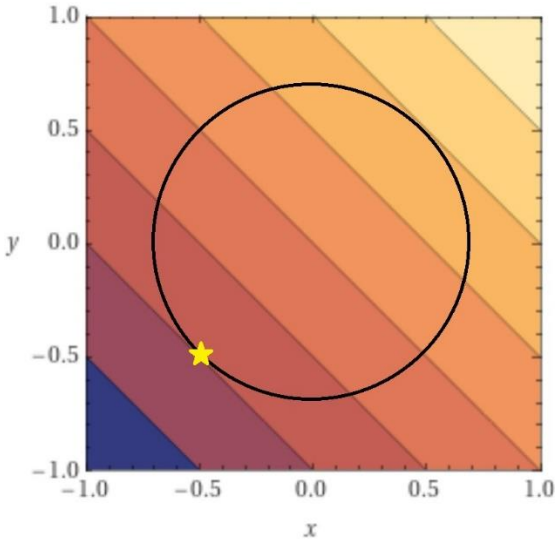

Figure 6 Visualization of a Pareto optimal solution.

One can also prove that no other solutions are pareto optimal. So, the pareto optimal solutions for this problem are the points of the form $(-u, -u)$ for $0 \leq u \leq 1$.

Finding a pareto optimal solution basically means there is no other solution the optimizer can guarantee is better than that one just by looking at the objective functions' values. The user of the optimizer must choose the pareto optimal solution that best suits their needs. That said, a multi-objective optimizer should not just return a value $x \in \Omega$ that is pareto optimal. Instead, it should return a list of all the values it found that are pareto optimal. Now that we know what are pareto optimal solutions and what their optimizers should return, we must see how the multi-objective algorithms we chose in Table 1 work.

3.2.1 MOPSO

Multi-Objective Particle Swarm Optimization (MOPSO) is an extension of the standard Particle Swarm Optimization (PSO) tailored for dealing with multi-objective problems. It was introduced by Carlos Coello in 2002 [37]. Both versions use a fitness value to update the particles' velocity. In the standard version, the particles' fitness value is determined solely by the objective function evaluation. In the multi-objective version, the particles are sorted according to their pareto dominance. The particles that are not pareto dominated by any other belong to the first front. Then, of the remaining particles, the ones that are not pareto dominated by any others belong to the second front, and so on. This way we sorted the particles by their pareto dominance. Finally, the fitness value of a particle is the front it belongs to. A list is maintained to keep all the pareto optimal solutions found by the algorithm (instead of a single

best solution). All remaining steps of the algorithm are like the single objective version, so we won't extend the description here.

The technique described above to turn a single objective optimization algorithm into a multi-objective optimization algorithm is called Non-dominated Sorting, because we sort the points giving priority to the ones that are not dominated by any other (as in pareto dominated).

The MOPSO implementation we used in this paper was provided by the Python library pygmo [35] version 2.19.5.

3.2.2 NSGA-II

NSGA-II is a genetic algorithm that uses Non-dominated Sorting (see section 3.2.1) to evaluate the fitness of the points. It was introduced by Kalyanmoy Deb in 2002 [17]. The NSGA-II algorithm can be described by the following sequence of steps:

1. **Initialization:** Begin with a randomly generated population of potential solutions. Each individual in the population represents a potential solution to the optimization problem.
2. **Non-dominated Sorting:** Sort the population into different fronts based on the concept of Pareto dominance.
3. **Crowding Distance Assignment**: For each front, calculate the crowding distance. This helps maintain diversity in the population by preserving a spread of solution.
4. **Selection**: Create a mating pool by selecting solutions based on their rank and crowding distance, ensuring that the best solutions have a higher chance of being selected for mating.
5. **Crossover and Mutation**: Apply crossover and mutation operators to the mating pool to create a child population. Crossover combines pairs of solutions to produce new offspring, while mutation introduces random changes to individuals.
6. **Combination**: Combine the parent and child population.
7. **Selection:** Sort the combined population and select the best N solutions based on nondomination and crowding distance to form a new parent population. N is the population size.
8. **Termination:** Repeat steps 2 to 7 until a stopping criterion is met.

NSGA-II specifically improves upon its predecessor by introducing a fast non-dominated sorting approach, an elitist strategy and a parameter-less diversity-preserving mechanism.

The NSGA-II implementation we used in this paper was provided by the Python library pymoo [31] version 0.6.0.

3.2.3 MOEA/D

The MOEA/D is a multi-objective evolutionary algorithm based on decomposition. The main strategy of the algorithm is decomposing the multi-objective problem into solving numerous single-objective problems. The algorithm was introduced by Qingfu Zhang in 2007 [16]. It can be described by the following sequence of steps:

1. **Initialization:**
    a. Define T: the number of subproblems considered.
    b. Define λ: a uniform spread of weight vectors.
    c. Define T': the number of weight vectors in the neighborhood of each weight vector.
    d. Initialize z: the ideal point which is the best value found so far for each objective.
    e. Generate an initial population randomly.
    f. Compute the Euclidean distances between any two weight vectors and identify the closest weight vectors for each weight vector.
2. **Update:** for each subproblem, perform the following steps:
    a. Reproduction: randomly select two indexes from the neighborhood and generate a new solution using genetic operators.
    b. Improvement: Apply a problem-specific repair/improvement heuristic to produce a better solution.
    c. Update of z: if the new solution leads to an improvement in any of the objectives, update the ideal point z.
    d. Update of neighborhood solutions: update the solutions of neighboring subproblems if the new solution is better.
    e. Update of the External Population (EP): Add the new solution to EP if it is non-dominated, and remove from EP all solutions that are dominated by the new solution.

3. **Stopping Criteria:** Repeat the update step until a stopping criterion is met.

At each generation, MOEA/D maintains:

- A population of points, each representing the current solution to a subproblem.
- The ideal point z, representing the best value found so far for each objective.
- An external population (EP) used to store non-dominated solutions found during the search.

This process aims to find a diverse set of solutions that approximate the Pareto front for the multi-objective optimization problem.

The MOEA/D implementation used in this paper was provided by the Python library pymoo [31] version 0.6.0.

3.2.4 MHACO

MHACO is a multi-objective algorithm built on top of ACO (Ant Colony Optimization). The algorithm was introduced by Giacomo Acciarini in 2020 [13]. The algorithm can be described by the following sequence of steps:

1. **Initialization:** Randomly generate the initial population of size N_p. Generate a solution archive of size $K < N_p$.
2. **Reproduction:** If the generation number is higher than 1, create a merged list of $N_p + K$ individuals by combining the archive and newly generated offspring.
3. **Ranking using the hypervolume-comparison operator**: Rank individuals of the merged list to determine which will be kept in the archive. The archive is updated only if at least one offspring outperforms the worst in the archive.
4. **Generation of new offspring**: Use the evolutionary operator to generate new individuals from those in the archive based on their positions in the archive and associated pheromone values.
5. **Algorithm Iteration**: The algorithm goes back to the reproduction step and repeats the process. At the first iteration, only the initial population is ranked, while in subsequent iterations, both the solution archive and the population are sorted. The user can tune the solution archive size (K) and the parameters T, N_G, and q to adjust the pheromone values and the optimization process.

6. **Termination:** The process continues until a stopping criterion is met, such as a certain number of function evaluations without updates to the archive or reaching a maximum number of generations.

MHACO algorithm focuses on the hypervolume metric as a performance indicator, considering both the quality and diversity of the Pareto front. The algorithm's parameters, including the solution archive size and the user-defined parameters T, N_G, and q, are crucial in controlling the exploration and exploitation balance, as well as the convergence behavior of the algorithm.

The MHACO implementation used in this paper was provided by the Python library pygmo [35] version 2.19.5.

4 Test problems

We tested the chosen algorithms (see Table 1) on 3 similar problems, called Jupiter Easy, Jupiter Hard and Cassini 1. The first two problems were designed by the author of this paper. The last is a well-known problem in literature.

4.1 Jupiter Easy

4.1.1 Single-objective version

Starting on Earth ground, minimize the Δv a spacecraft must apply to enter an orbit around Jupiter with pericenter radius equal to 600,000 km and eccentricity equal to 0.98 (any orbit meeting these criteria is acceptable). The planets' ephemerides are given. There are two parameters for this optimization: time of launch from Earth T_0 in MJD2000 (Modified Julian Date 2000) and time of flight to Jupiter Δt in days. See section 5 for a detailed explanation of how the Δv is determined from those parameters. The problem's constraints are:

Table 2 Bounds for the parameters of problem Jupiter Easy.

Lower Bound	Variable	Upper Bound
9131.5 MJD2000	T_0 (Launch from Earth)	9495.5 MJD2000
300 days	Δt (Earth – Jupiter trip)	3000 days

4.1.2 Multi-objective version

The multi-objective version adds $T_f = T_0 + \Delta t$ (that is, time of arrival) as a new objective for the mission stated in 4.1.1. That is, the algorithm must find the pareto optimal points for the function $(\Delta v, T_f)$. Notice the second objective is time of arrival, not time of flight.

4.2 Jupiter Hard

4.2.1 Single-objective version

Starting on Earth ground and doing a fly-by through Mars, calculate the minimum Δv a spacecraft must apply to enter an orbit around Jupiter with pericenter radius equal to 600,000 km and eccentricity equal to 0.98 (any orbit meeting these criteria is acceptable). The planets' ephemerides are given. There are three parameters for this optimization: time of launch from Earth T_0 in MJD2000 (Modified Julian Date 2000), time of flight Earth – Mars Δt_1 in days and time of flight Mars – Jupiter Δt_2 in days. See section 5 for a detailed explanation of how the Δv is determined from those parameters. The problem's constraints are:

Table 3 Bounds for the parameters of problem Jupiter Hard.

Lower Bound	Variable	Upper Bound
9131.5 MJD2000	T_0 (Launch from Earth)	10958 MJD2000
50 days	Δt_1 (Earth – Mars trip)	1000 days
300 days	Δt_2 (Mars – Jupiter trip)	3000 days

4.2.2 Multi-objective version

The multi-objective version adds $T_f = T_0 + \Delta t_1 + \Delta t_2$ (that is, time of arrival) as a new objective for the mission stated in 4.2.1. That is, the algorithm must find the pareto optimal points for the function $(\Delta v, T_f)$. Notice the second objective is time of arrival, not time of flight.

4.3 Cassini 1

4.3.1 Single-objective version

Starting on Earth ground and doing fly-bys through Venus, Venus, Earth and Jupiter in that order, calculate the minimum Δv a spacecraft must apply to enter an orbit around Saturn with pericenter radius equal to 108,950 km and eccentricity equal to 0.98 (any orbit meeting these criteria is acceptable). The planets' ephemerides are given. There are six parameters for

this optimization: time of launch from Earth T_0 in MJD2000 (Modified Julian Date 2000), time of flight Earth – Venus Δt_1 in days, time of flight Venus – Venus Δt_2 in days, time of flight Venus – Earth Δt_3 in days, time of flight Earth – Jupiter Δt_4 in days and time of flight Jupiter – Saturn Δt_5 in days. See section 5 for a detailed explanation of how the Δv is determined from those parameters. The problem's constraints are:

Table 4 Bounds for the parameters of problem Cassini 1.

Lower Bound	Variable	Upper Bound
-1000 MJD2000	T_0 (Launch from Earth)	0 MJD2000
30 days	Δt_1 (Earth – Venus trip)	400 days
100 days	Δt_2 (Venus – Venus trip)	470 days
30 days	Δt_3 (Venus – Earth trip)	400 days
400 days	Δt_4 (Earth – Jupiter trip)	2000 days
1000 days	Δt_5 (Jupiter – Saturn trip)	6000 days

This problem is also available at [38].

4.3.2 Multi-objective version

The multi-objective version adds $T_f = T_0 + \sum_{i=1}^{5} \Delta t_i$ (that is, time of arrival) as a new objective for the mission stated in 4.3.1. That is, the algorithm must find the pareto optimal points for the function $(\Delta v, T_f)$. Notice the second objective is time of arrival, not time of flight.

5 Methodology

We used ESA (European Space Agency) Advanced Concepts Team's Space Mechanics Toolbox software to calculate the Δv of a mission. The code we used is available at GitHub on https://github.com/fbrunodr/TestOptimizersOnSpaceTrajectoryProblems. This software takes as input the time of departure from earth T_0 in MJD2000 (Modified Julian Date 2000) and the time each trip Δt_i takes and outputs the Δv required for this mission. To calculate the Δv given this input the software uses Lambert transfers. Lambert transfers come from the study of Lambert's problem: given two points in space, a central attractor body and a time Δt, there exists a single orbit that passes through those points in Δt. We call this orbit Lambert transfer. We know the positions of the planets in the solar system through time given the planets' ephemerides. That said, given a departure time from a planet and the arrival time at another planet, we know what orbit the spacecraft should follow considering the Sun as the central attractor body and ignoring planets' gravitational influence. This is a valid approximation, as planets will be too distant from the spacecraft during almost all the transition to have a considerable influence on the motion of the spacecraft. Only when the spacecraft reaches a planet do we consider its influence. So, given the time of arrival at each planet we know what trajectory the spacecraft must follow. For example, if we set the spacecraft to departure Earth on 18th November 2023, reach Mars on 18th November 2024 and finally Jupiter on 18th November 2026 we will have a trajectory that looks like the following image:

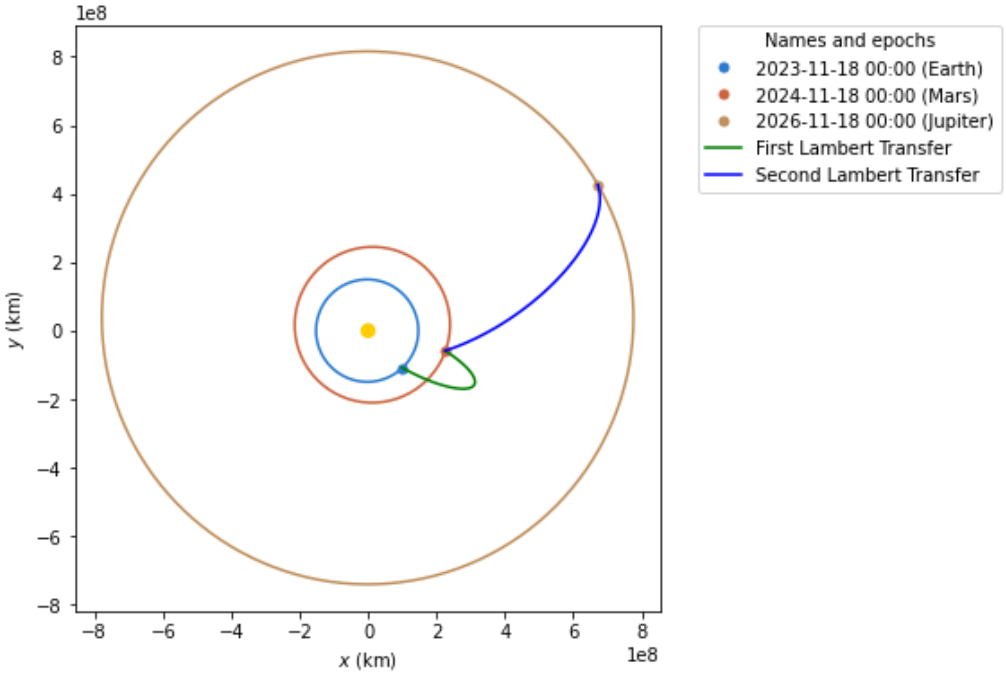

Figure 7 Lambert transfers from Earth to Mars then Jupiter given arrival times.

Now that we know what trajectory the spacecraft must follow between planets, we must know what trajectory it follows in the planet's sphere of influence and more importantly what Δv the spacecraft must output to follow the defined trajectory. For that, the ESA's software considers three cases: departure, fly-by and arrival. Let's see each one separately.

5.1 Departure

As said in the problems' statement, the spacecraft is launched from Earth. We have the first lambert transfer and thus the velocity the spacecraft should have at departure time. We also know what velocity Earth has at departure time. So, we can estimate the Earth launch Δv by calculating the difference between the required spacecraft velocity and Earth's velocity and taking the norm of this value.

5.2 Fly-by

We know the orbits the spacecraft should follow before and after passing by a planet. We also know the planet's velocity. So, we know the spacecraft's velocity relative to the planet during approach and departure. Let's call those velocities v_{in} and v_{out}. We may assume that v_{in} and v_{out} are coplanar, as the planets in the solar system and the Sun are approximately in the same plane. The spacecraft does a hyperbolic trajectory through the planet at arrival (as the spacecraft is coming from outside the planet's sphere of influence) and at departure. Let's apply an impulse at the pericenter of the incoming hyperbola tangent to the spacecraft's velocity. We can determine the speed the spacecraft has at the pericenter of the incoming hyperbola using energy conservation:

$$\varepsilon = \frac{v_{in}^2}{2} = \frac{v_{pin}^2}{2} - \frac{\mu}{r_p}$$

Where ε is the spacecraft's specific energy, v_{in} is the incoming velocity from outside the planet's sphere of influence, v_{pin} is the velocity the spacecraft has at the pericenter of the planet, r_p is the pericenter of the hyperbolic trajectory and μ is the standard gravitational parameter of the primary body. Rearranging terms we get:

$$\|v_{pin}\| = \sqrt{v_{in}^2 + \frac{2\mu}{r_p}}$$

As previously mentioned, we apply an impulse at the pericenter of the spacecraft's trajectory tangent to the spacecraft's velocity. So, the outcoming hyperbola also has a pericenter r_p. Applying conservation of energy, the speed the spacecraft must have at the pericenter so it departures from the planet with velocity (relative to the planet) v_{out} is:

$$\|v_{pout}\| = \sqrt{v_{out}^2 + \frac{2\mu}{r_p}}$$

So, we can determine the Δv we must apply during the fly-by using the following formula:

$$\Delta v = \left| \|v_{pout}\| - \|v_{pin}\| \right| = \left| \sqrt{v_{out}^2 + \frac{2\mu}{r_p}} - \sqrt{v_{in}^2 + \frac{2\mu}{r_p}} \right|$$

We do not want the spacecraft to simply exit the planet's sphere of influence with speed $\|v_{out}\|$. We want it to exit with velocity (mind the direction) v_{out}. This implies we must choose the pericenter r_p carefully so the outcoming velocity matches the expected direction. Let α be

the deflection angle after passing by the planet. We can calculate α using the dot product of v_{in} and v_{out}:

$$\alpha = \text{acos}((v_{in} \cdot v_{out})/(\|v_{in}\| \, \|v_{out}\|))$$

We have two hyperbolas to analyze, each one with its deflection angle δ (see Figure 8 for better visualization).

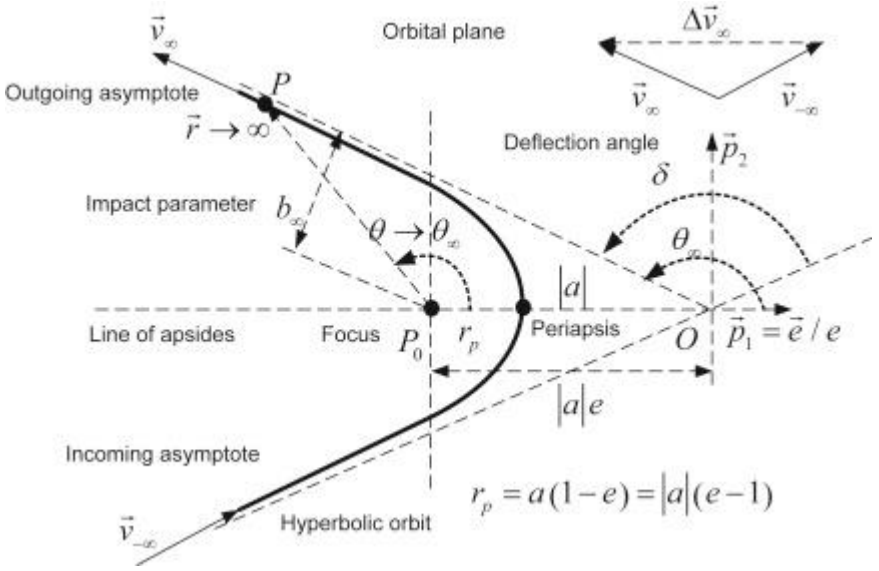

Figure 8 Diagram of a hyperbolic trajectory. Image taken from [39].

Let's call δ_{in} the deflection angle on the incoming hyperbola and δ_{out} the deflection angle of the outcoming hyperbola. We must have:

$$\alpha = \frac{\delta_{in}}{2} + \frac{\delta_{out}}{2}$$

We can determine δ_{in} and δ_{out} using only the incoming / outcoming velocities and the pericenter distance:

$$\delta = 2\sin^{-1}\left(\frac{1}{1+\frac{r_p v_\infty^2}{\mu}}\right) \Rightarrow$$

$$\alpha = \frac{\delta_{in}}{2} + \frac{\delta_{out}}{2} = \sin^{-1}\left(\frac{1}{1+\frac{r_p v_{in}^2}{\mu}}\right) + \sin^{-1}\left(\frac{1}{1+\frac{r_p v_{out}^2}{\mu}}\right)$$

ESA's software uses the Newton-Raphson method to find the value of r_p that satisfies this last equation (v_{in}, v_{out}, μ and α are known). Finally, known the value of r_p we know all the

variables needed to determine Δv. In short, we approach the planet in such a way that we can apply a tangent impulse at the pericenter of the incoming hyperbola so the outcoming hyperbola has velocity (magnitude and direction) v_{out}.

5.3 Arrival

On arrival the spacecraft will approach the final planet in a hyperbola (the spacecraft is coming from outside the planet's sphere of influence) with pericenter distance equal to the pericenter distance defined in the final orbit. Finally, the spacecraft will apply an impulsive force tangent to the spacecraft's velocity (to preserve the pericenter distance) so the final trajectory will match the orbit defined in the problem. The speed at the pericenter of the incoming hyperbola is

$$\|v_1\| = \sqrt{v_{in}^2 + \frac{2\mu}{r_p}}$$

Where v_{in} is the spacecraft's velocity relative to the planet (which can be determined by the previous lambert's orbit and the planet's velocity), μ is the standard gravitational parameter of the planet and r_p is the pericenter distance defined in the problem statement. The speed the spacecraft has at the pericenter of the final orbit is found by the following formula [40]:

$$\|v_2\| = \sqrt{\frac{2\mu}{r_p} - \frac{\mu * (1 - e)}{r_p}}$$

Where e is the eccentricity of the final orbit. So, the Δv the spacecraft must apply to enter the final orbit is determined by:

$$\Delta v = \|v_1\| - \|v_2\| = \sqrt{v_{in}^2 + \frac{2\mu}{r_p}} - \sqrt{\frac{2\mu}{r_p} - \frac{\mu * (1 - e)}{r_p}}$$

This Δv is then added to the final answer.

5.4 Observations

The software described in this section also applies penalties if the pericenter radius of each fly-by is too close to the planet, to avoid the spacecraft entering the planet's atmosphere or colliding with it. If you want to inspect the software further, visit the GitHub repository.

6 Results

As shown in Table 1, the chosen optimizers can be classified between single-objective optimizers and multi-objective optimizers. We will compare them separately to provide a fair comparison.

6.1 Single-objective optimizers

To assess the chosen single objective optimizers, we compared the number of evaluations (that is, how many times the objective function was called during optimization) and the quality of the results of the optimizers (lower values of Δv imply better results). After collecting all the data we will display in Chapter 6, we discovered that the best solution (that is, the one that minimizes Δv) for the problem Jupiter Easy outputs a Δv of $12.97\ km/s$. For the problem Jupiter Hard, the best solution outputs $9.42\ km/s$. For the problem Cassini 1 the best solution outputs $4.93\ km/s$, as seen in [38]. That is why the best Δv graphs start at those values. Each algorithm was run 100 times in each test problem (to mitigate the effects of variability and ensure statistical robustness in the analysis). The best solutions can be visualized below:

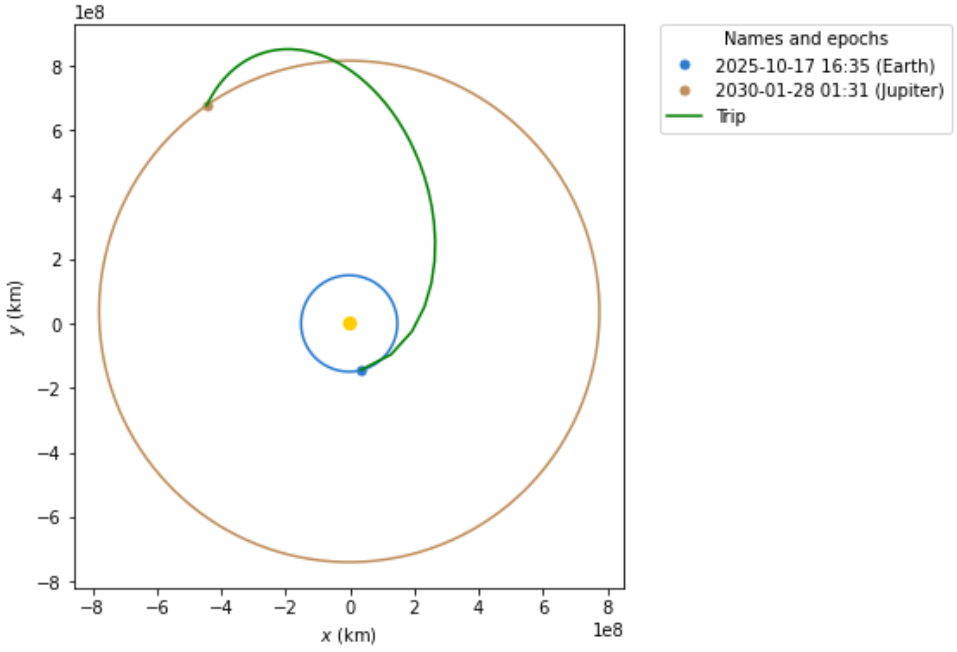

Figure 9 Best solution found for Jupiter Easy.

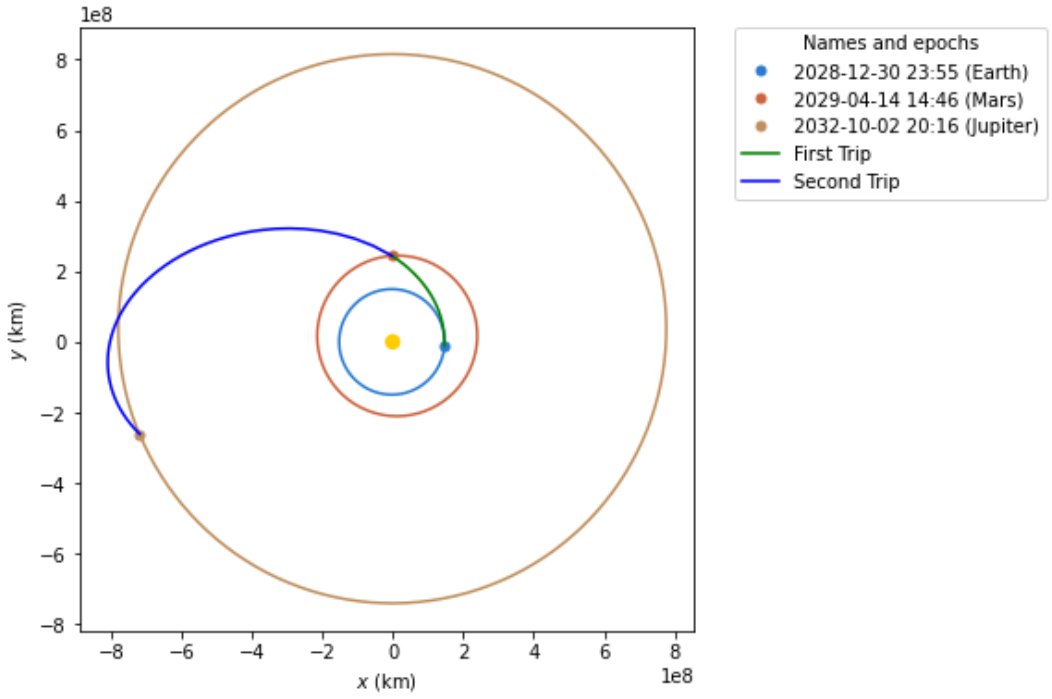

Figure 10 Best solution found for Jupiter Hard.

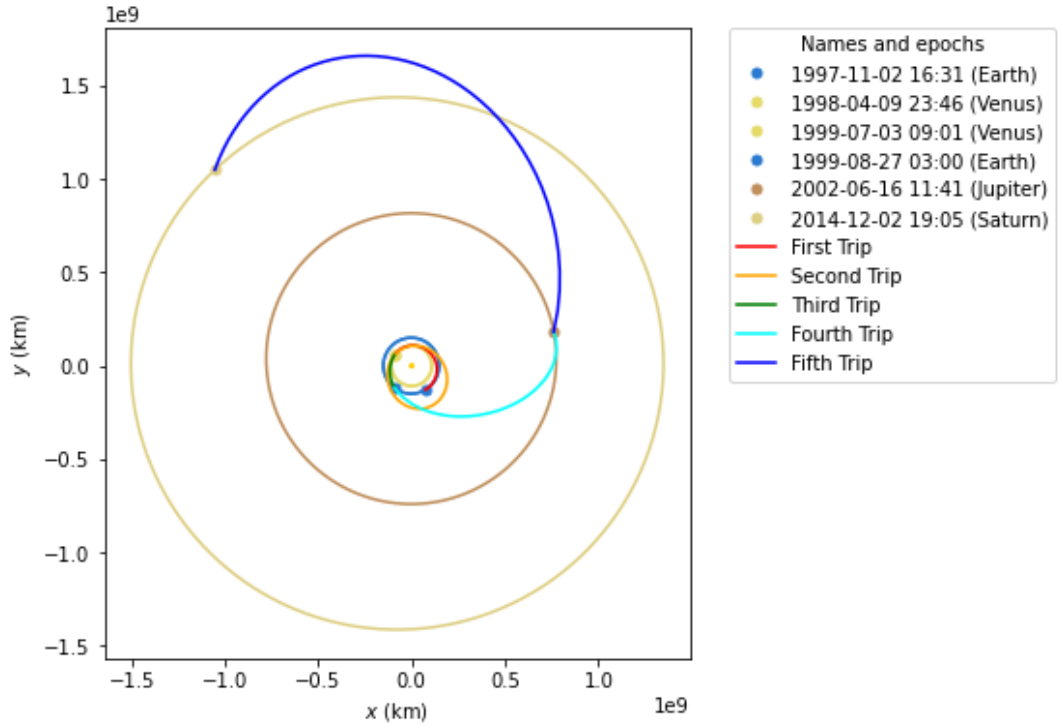

Figure 11 Best solution for Cassini 1 according to [38].

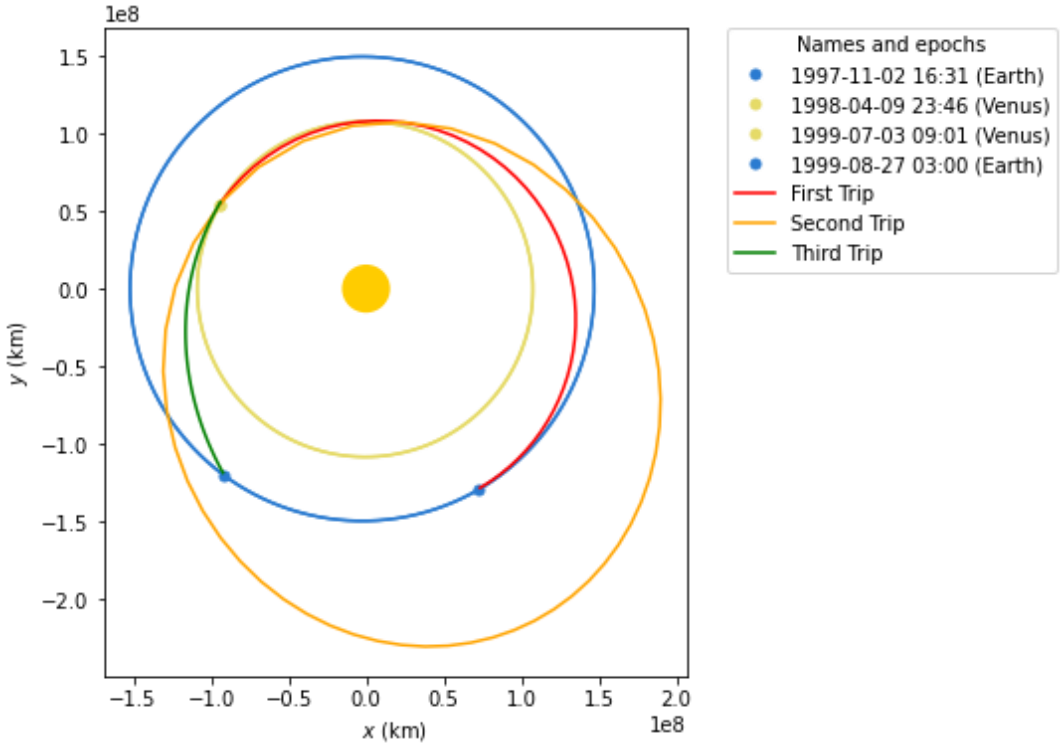

Figure 12 Best solution for Cassini 1, only first three trips.

A careful reader will notice that the solutions are not Hohmann transfers. In particular, the solution for Jupiter 1 is not a Hohmann transfer, although one expects it to be the best solution

for this problem. Indeed, the best possible solution for this problem would be a Hohmann transfer, but the constraints do not allow that to happen. To perform a Hohmann transfer between those two orbits one would need to exit Earth at time T_0 and reach Jupiter at time $T_0 + \Delta t$ such that they are perfectly aligned with the Sun in the middle. Because Earth's and Jupiter's orbits are defined, the Hohmann transfer orbit is also well defined. The time it takes for the spacecraft to do a revolution around the Sun is given by Kepler's third law:

$$\frac{a^3}{T^2} = \frac{\mu_{sun}}{4\pi^2}$$

Where a is the semi-major axis of the orbit, T is the period of the orbit and μ_{sun} is the Sun standard gravitational parameter. This means that Δt is constrained by Earth's and Jupiter's orbits themselves. So, if we wish to perform a Hohmann transfer between Earth and Jupiter, the only parameter we have is T_0. It happens that setting the departure time (from Earth) in the first problem to the year 2025 makes it impossible to find T_0 that allows for a Hohmann transfer. If we set a broader range of time for T_0, we can find the expected solution:

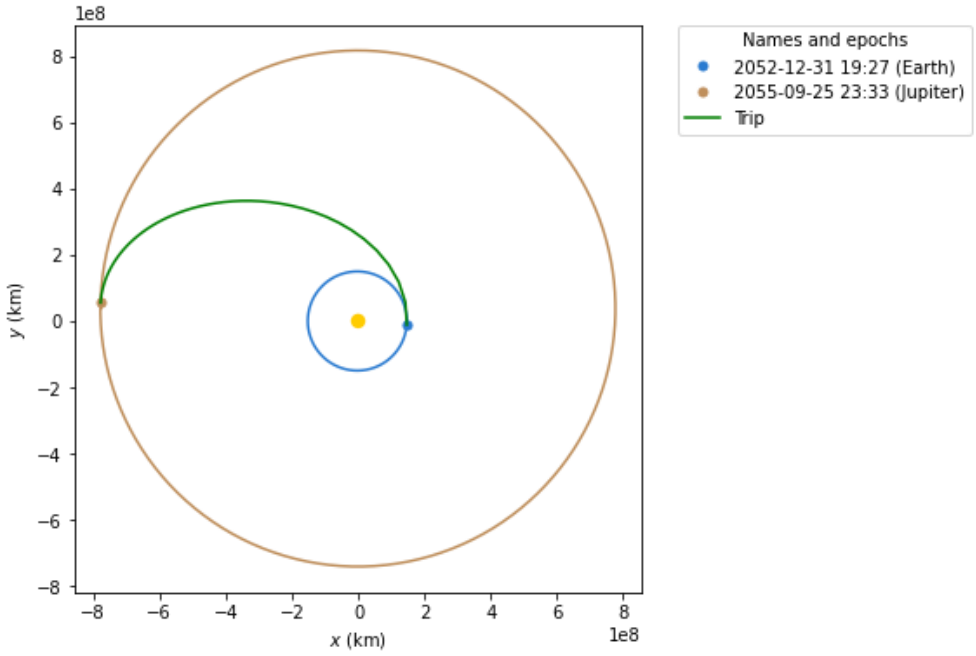

Figure 13 Hohmann transfer from Earth to Jupiter.

Notice the travel time in Figure 13 is 998.17 days. The semi-major axis of Jupiter is $7.78479 \times 10^8\ km$ and the semi-major axis of Earth is $1.49598 \times 10^8\ km$ [41]. Hence, the semi-major axis of the Hohmann transfer is $a_{transfer} = (a_{jup} + a_{earth})/2 = 4.6409 \times 10^8\ km$. Also, Sun's gravitational parameter is $1.32712 \times 10^{11} km^3/s^2$ [42]. So, the period

for the Hohmann's orbit is $T = 2\pi\sqrt{\frac{a^3}{\mu_{sun}}} = 1995.46$ days. This implies that half a revolution takes 997.72 days, which is a 0.04% difference from the solution found in Figure 13. While the Jupiter Easy solution outputs $12.97\ km/s$ of Δv, the solution provided in Figure 13 outputs $12.57\ km/s$.

6.1.1 Grid Search

As said in section 3.1.1, we recommend using grid search when the input dimension is small. Not only it is fast enough for the optimization process, but it also allows visualizing the cost of each input. The problem Jupiter Easy has a 2-dimensional input (time of departure and time of flight between Earth and Jupiter), so we can plot a heat map of the solution:

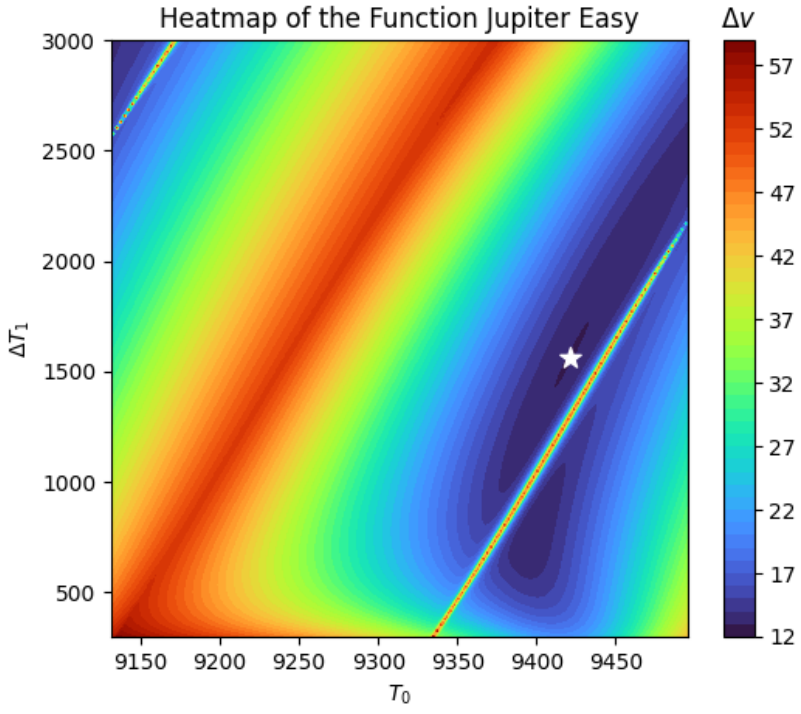

Figure 14 Δv of each input of the problem Jupiter 1.

The star on Figure 14 marks the point of minimum Δv found by the grid search, with Δv = 12.966 km/s.

6.1.2 PSO

We optimized each test problem using the PSO algorithm 100 times. The PSO implementation used in this paper was provided by the Python library pymoo [31] version 0.6.0. The PSO algorithm requires values of inertia, cognitive impact, social impact, initial velocity and maximum velocity to work. We used the default values provided by the library during the optimization process: inertia = 0.9, cognitive impact = 2.0, social impact = 2.0 and maximum velocity rate = 0.2. The maximum velocity rate is normalized over the lower and upper bounds of the problem. The initial population is sampled using Latin Hypercube Sampling over the lower and upper bounds of the problem. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization) and best Δv found per run (run meaning we executed the optimization algorithm).

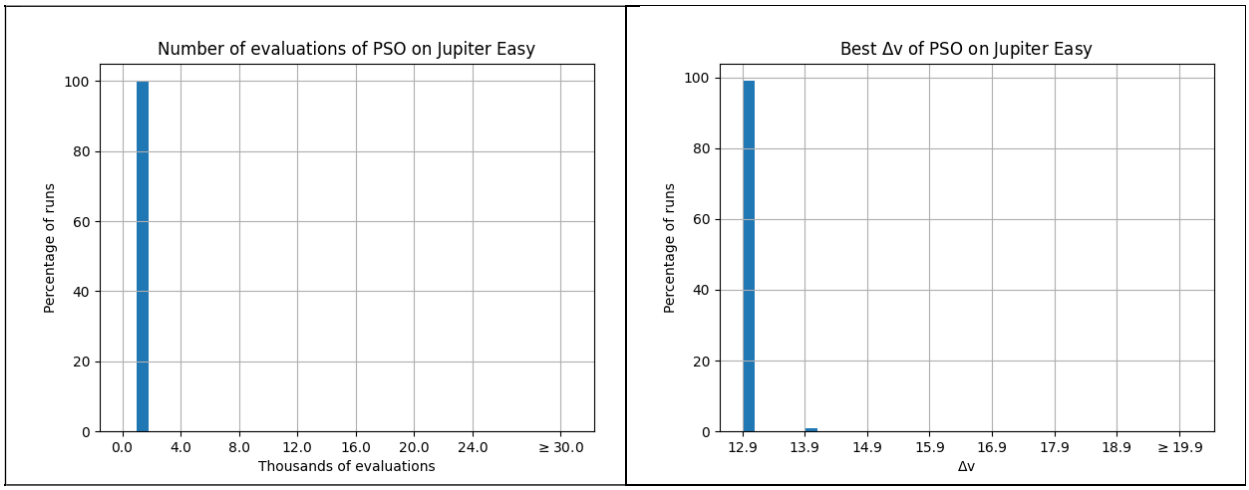

Figure 15 PSO performance on Jupiter Easy.

Figure 15 shows that on all the 100 times we run the optimizer PSO on problem Jupiter Easy it needed between 1000 and 2000 evaluations of the objective function to finish the optimization process. It also obtained the best solution (the one with $\Delta v = 12.9\ km/s$) 99 times and only a single time it finished with a suboptimal solution with $\Delta v = 13.9\ km/s$.

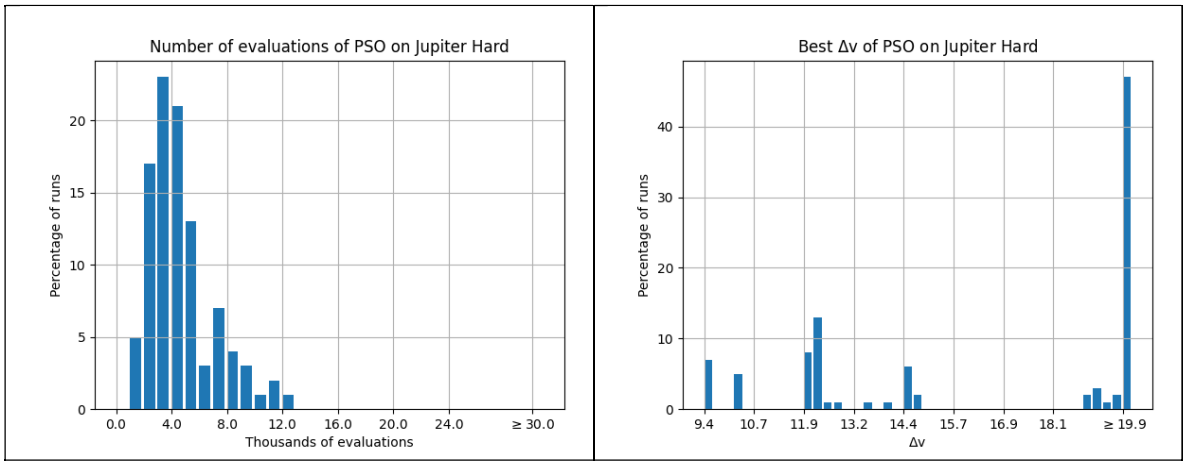

Figure 16 PSO performance on Jupiter Hard.

We can notice from Figure 16 that the performance of the algorithm PSO on problem Jupiter Hard was much worse than in the previous problem. In more than 40 % of the times we run the algorithm it obtained a solution with Δv at least 19.9 km/s, while the best solution with $\Delta v = 9.4\ km/s$ was found only in 7 % of the runs.

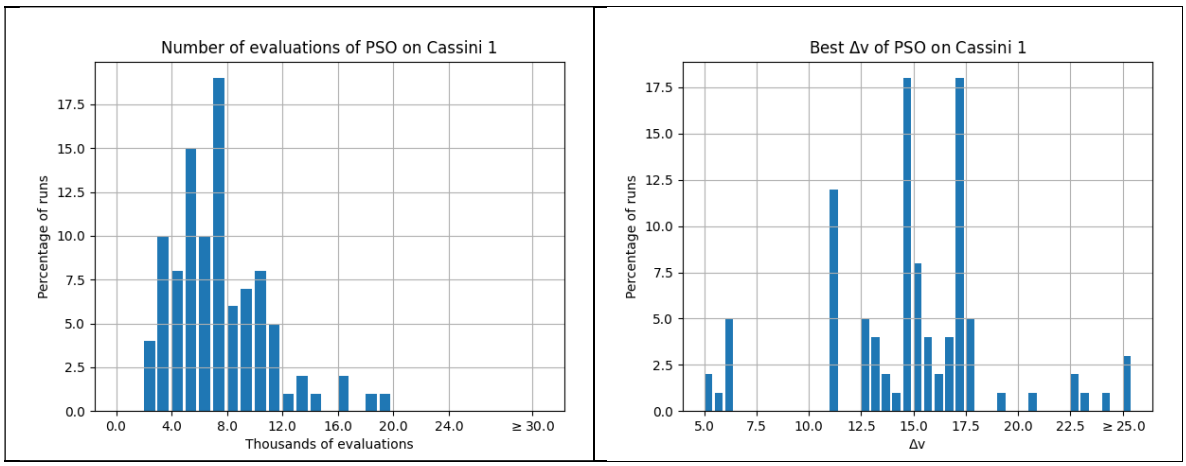

Figure 17 PSO performance on Cassini 1.

From Figure 17 we can notice that in more than 50 % of the times we run PSO it outputted a solution between 12.5 km/s and 20.0 km/s, which is far from the optimal solution 4.9 km/s. The table below compiles the performance of the algorithm PSO on all three problems:

Table 5 Average performance of PSO on test problems.

	Average number of Evaluations	Average Δv found in km/s	$\Delta v_{avg} - \Delta v_{best}$ in km/s
Jupiter Easy	1324.25	12.976	0.006
Jupiter Hard	4759.75	17.301	7.881
Cassini 1	7539.00	15.109	10.179

6.1.3 DE

We optimized each test problem using the DE algorithm 100 times. The DE implementation used in this paper was provided by the Python library pymoo [31] version 0.6.0. The DE implementation on pymoo requires the population size as a parameter. We used the default value of population size = 100. The initial population is sampled using random uniform distribution from the problem bounds. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization) and best Δv found per run (run meaning we executed the optimization algorithm).

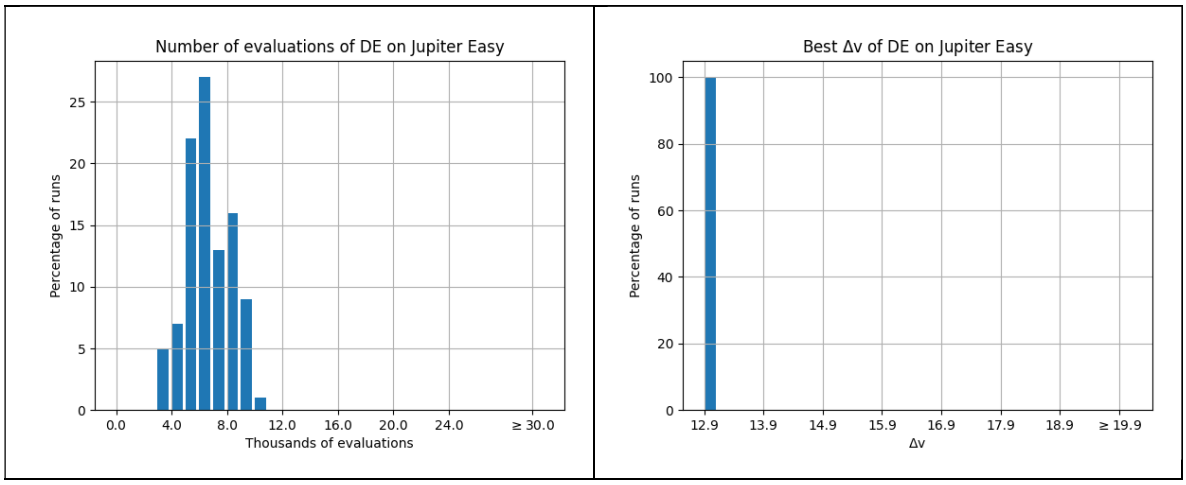

Figure 18 DE performance on Jupiter Easy.

Figure 18 shows that on all the 100 times we run the optimizer DE on problem Jupiter Easy we found the optimal solution with $\Delta v = 12.9\ km/s$. Notice though we needed more evaluations of the objective function than PSO.

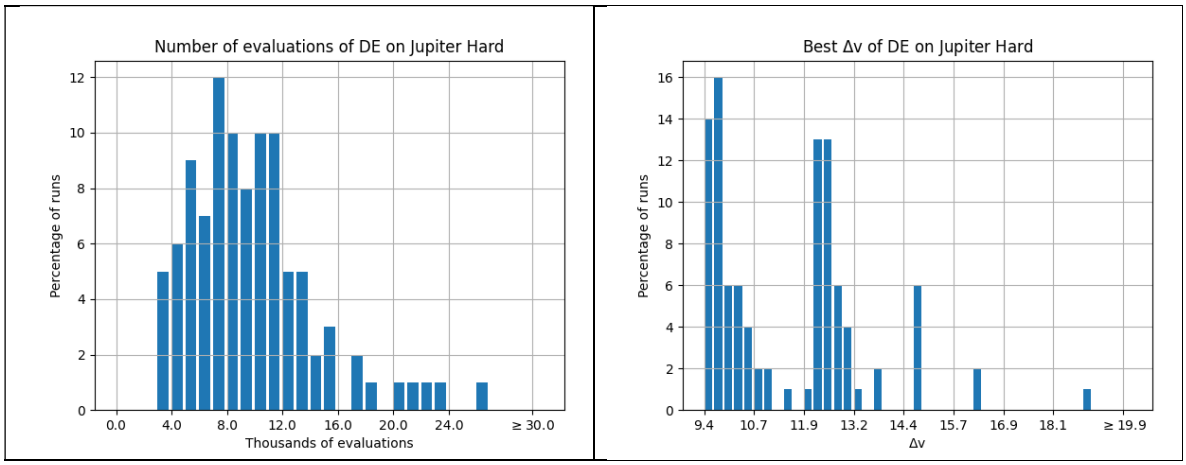

Figure 19 DE performance on Jupiter Hard.

Figure 19 shows that the performance of DE on Jupiter Hard was much better than that of PSO, as the algorithm was able to get close to the optimal Δv 30 % of the time. Also, the algorithm never returned a value of $\Delta v \geq 19.9\ km/s$, unlike PSO that returned a value in this range in more than 40 % of runs.

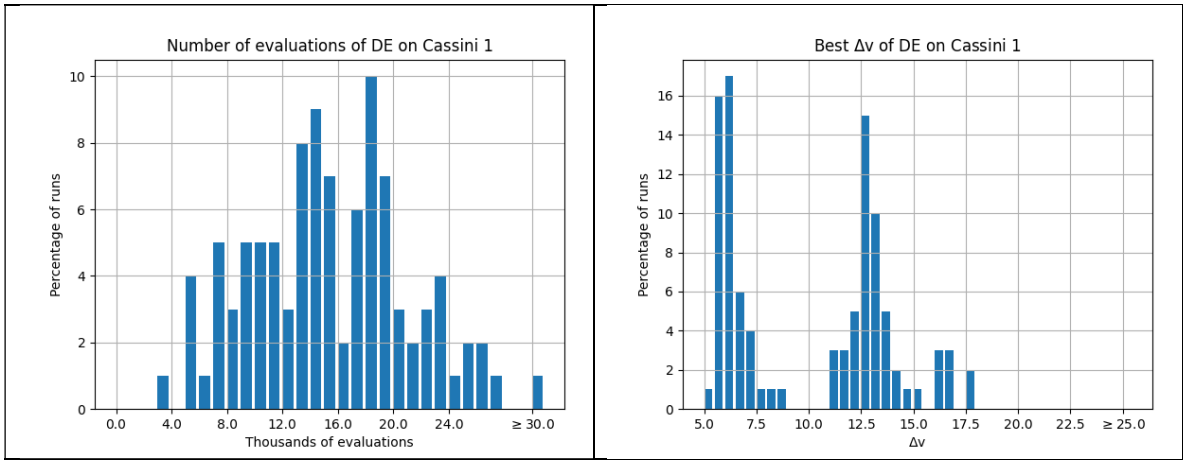

Figure 20 DE performance on Cassini 1.

Figure 20 shows that DE beats PSO in the problem Cassini 1 when it comes to finding the best Δv for the problem, although DE needs more evaluations of the objective function to converge.

Table 6 Average performance of DE on test problems.

	Average number of Evaluations	Average Δv found in km/s	$\Delta v_{avg} - \Delta v_{best}$ in km/s
Jupiter Easy	6675.00	12.967	0.000
Jupiter Hard	9808.00	11.532	2.112
Cassini 1	15489.00	10.105	5.175

6.1.4 GA

We optimized each test problem using the GA algorithm 100 times. The GA implementation used in this paper was provided by the Python library pymoo [31] version 0.6.0. The GA implementation on pymoo requires the population size as a parameter. We used the default value of population size = 100. The initial population is sampled using random uniform distribution from the problem bounds. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization) and best Δv found per run (run meaning we executed the optimization algorithm).

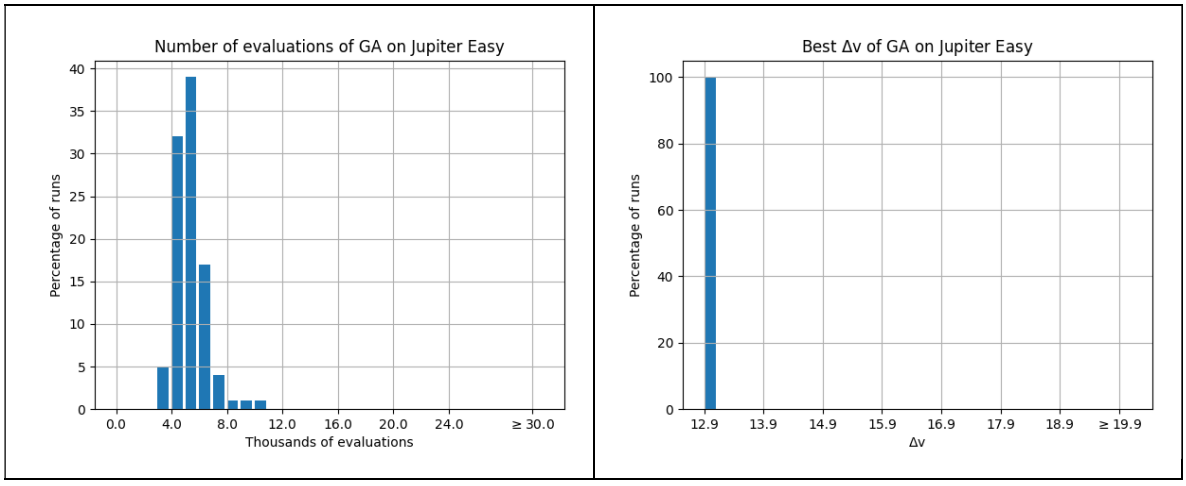

Figure 21 Performance of GA on Jupiter Easy.

The results presented on Figure 21 are similar to that of Figure 18 (DE algorithm on Jupiter Easy).

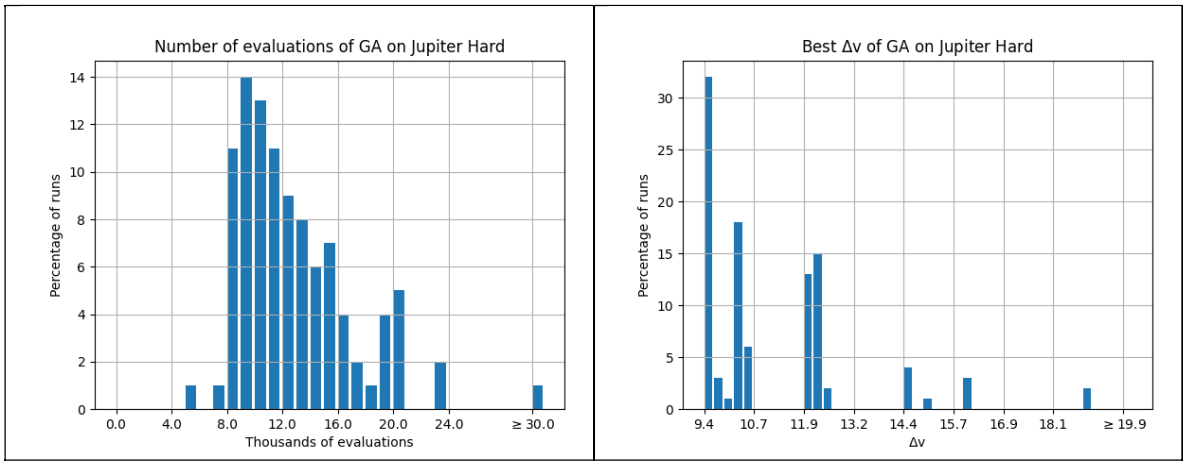

Figure 22 Performance of GA on Jupiter Hard.

Again, GA obtained results similar to that of DE, although GA seems to be converged to the best answer more often with more evaluations of the objective function.

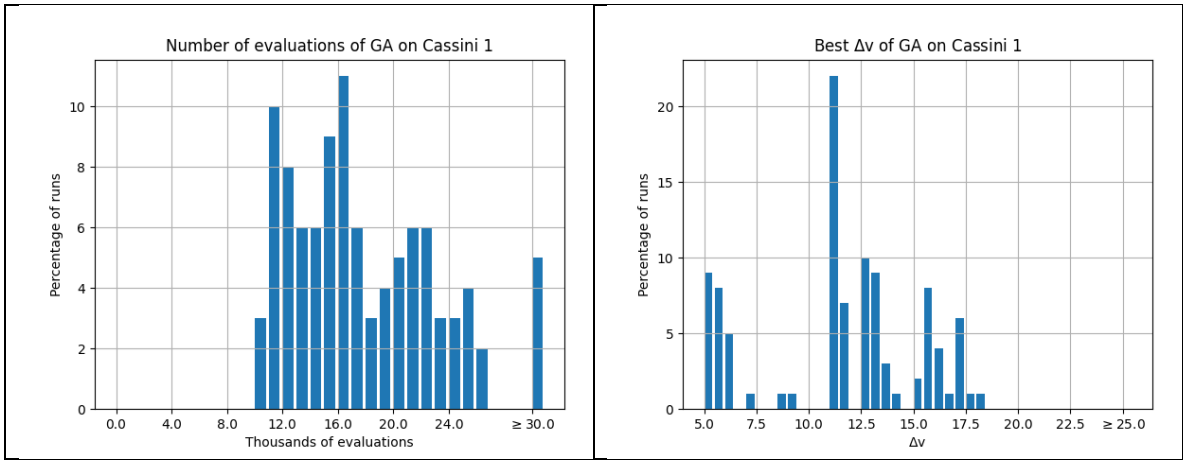

Figure 23 Performance of GA on Cassini 1.

Comparing Figure 23 and Figure 20, DE has converged to the minimum of Cassini 1 more often than GA, even though GA needed more function evaluations. We can see that behavior in the table below:

Table 7 Average performance of GA on test problems

	Average number of Evaluations	Average Δv found in km/s	$\Delta v_{avg} - \Delta v_{best}$ in km/s
Jupiter Easy	5354.00	12.967	0.000
Jupiter Hard	12900.00	11.175	1.755
Cassini 1	18289.00	11.591	6.661

Comparing Table 6 and Table 7 we can see DE performed better on Cassini 1 compared to GA, as it required less evaluations to get a better solution on average. On problem Jupiter Hard DE did less function evaluations and performed a bit worse. We concluded that DE performed better than GA on our test problems.

6.1.5 SA

We optimized each test problem using the SA algorithm 100 times. The SA implementation used in this paper was provided by the Python library pygmo [35] version 2.19.5. The SA implementation on pygmo accepts as parameters T_s, the starting temperature, T_f, the final temperature, $n_{T_{adj}}$, number of temperature adjustments in the annealing schedule, $n_{range_{adj}}$, the number of adjustments of the search range performed at a constant temperature, bin_{size} the number of mutations that are used to compute the acceptance rate and $start\_range$, the starting range for mutating the decision vector. We used all the default values provided by the library: $T_s = 10$, $T_f = 1$, $n_{T_{adj}} = 10$, $n_{range_{adj}} = 10$, $bin_{size} = 10$ and $start_{range} = 1$. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization) and best Δv found per run (run meaning we executed the optimization algorithm).

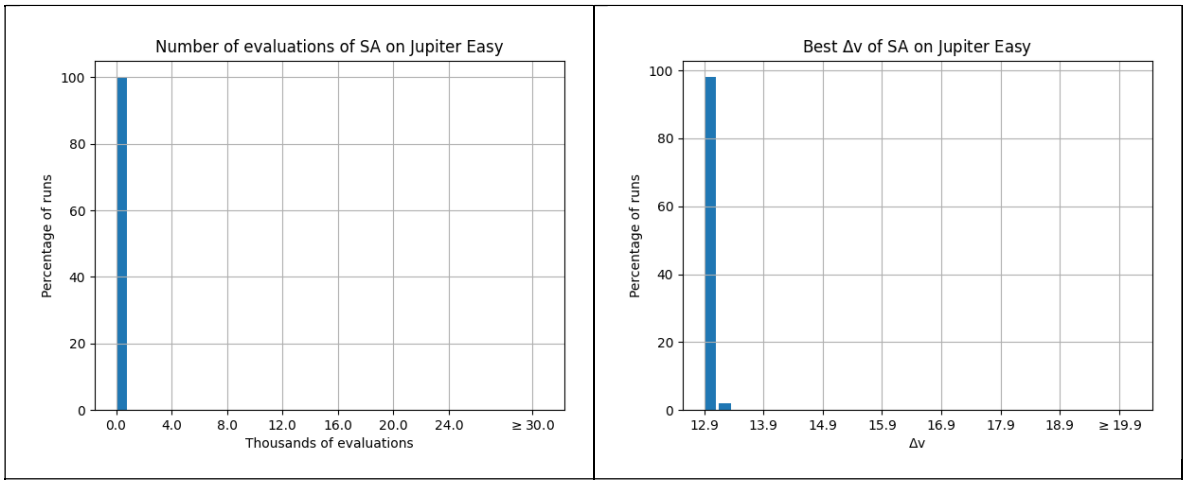

Figure 24 Performance of SA on Jupiter Easy.

Figure 24 shows that on all the 100 times we run the optimizer SA on problem Jupiter Easy it needed less than 1000 evaluations of the objective function to finish the optimization process. It also obtained the best solution (the one with $\Delta v = 12.9\ km/s$) in 98 % of the runs.

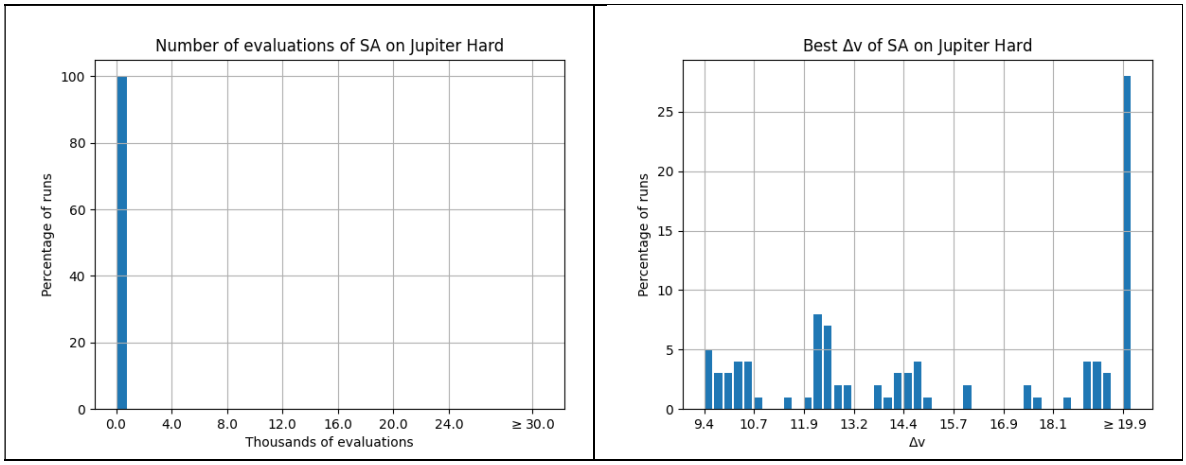

Figure 25 Performance of SA on Jupiter Hard.

Comparing Figure 16 and Figure 25, we can see SA performed better than PSO in the problem Jupiter Hard, although it needed less evaluations of the objective function to converge.

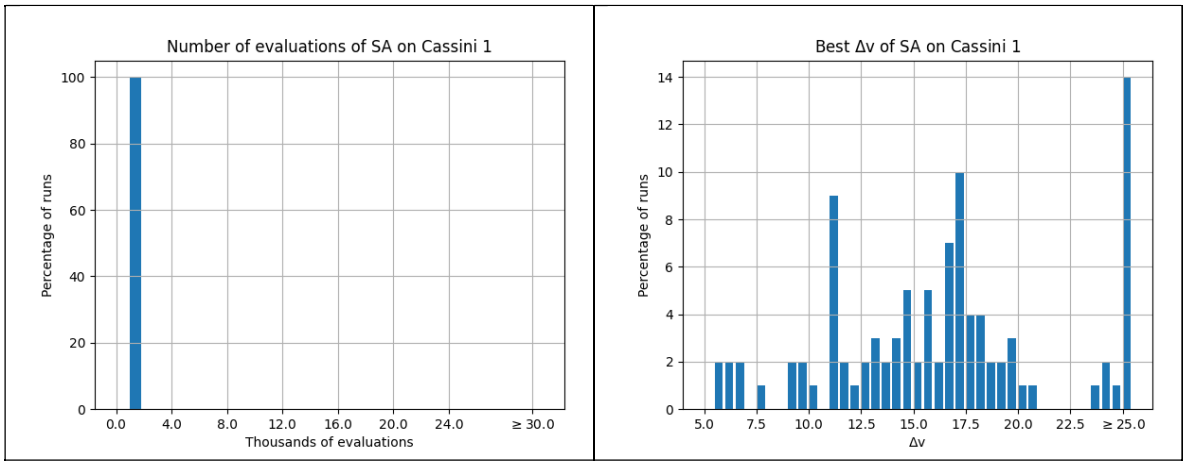

Figure 26 Performance of SA on Cassini 1.

According to Figure 26 SA performed poorly on problem Cassini 1, but it also needed much less evaluations of the objective function than the previously analyzed algorithms to converge: it needed between 1000 and 2000 evaluations in all the runs. The table below summarizes the performance of SA:

Table 8 Average performance of SA on test problems.

	Average number of Evaluations	Average Δv found in km/s	$\Delta v_{avg} - \Delta v_{best}$ in km/s
Jupiter Easy	528.00	12.991	0.021
Jupiter Hard	728.00	16.255	6.835
Cassini 1	1328.00	16.998	12.068

Comparing Table 5 and Table 8 we see that SA and PSO had comparable performance, although SA needed much less evaluations of the objective function to converge.

6.1.6 ABC

We optimized each test problem using the ABC algorithm 100 times. The ABC implementation used in this paper was provided by the Python library pygmo [35] version 2.19.5. The ABC implementation on pygmo accepts as parameters the population size and the number of generations. We set the population size to 128 and the number of generations to 64. Because we fixed the number of generations, the number of evaluations of the objective

function will be fixed. We chose a number of generations that lead to a number of evaluations comparable to the algorithms GA and DE on the problems Jupiter Hard and Cassini 1, so we can compare the algorithms more fairly. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization) and best Δv found per run (run meaning we executed the optimization algorithm).

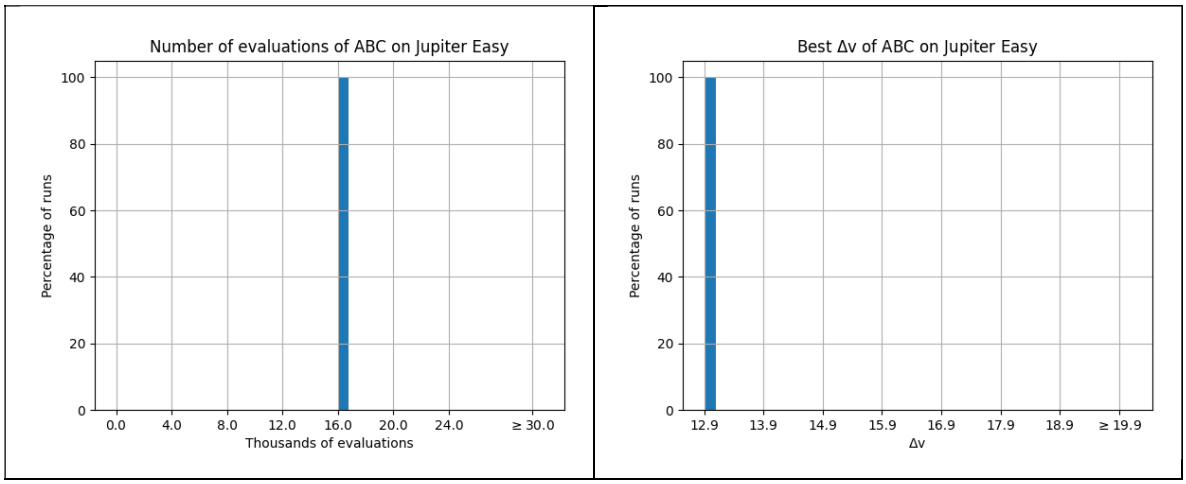

Figure 27 Performance of ABC on Jupiter Easy.

As expected, the algorithm ABC always finds the optimal Δv on problem Jupiter Easy.

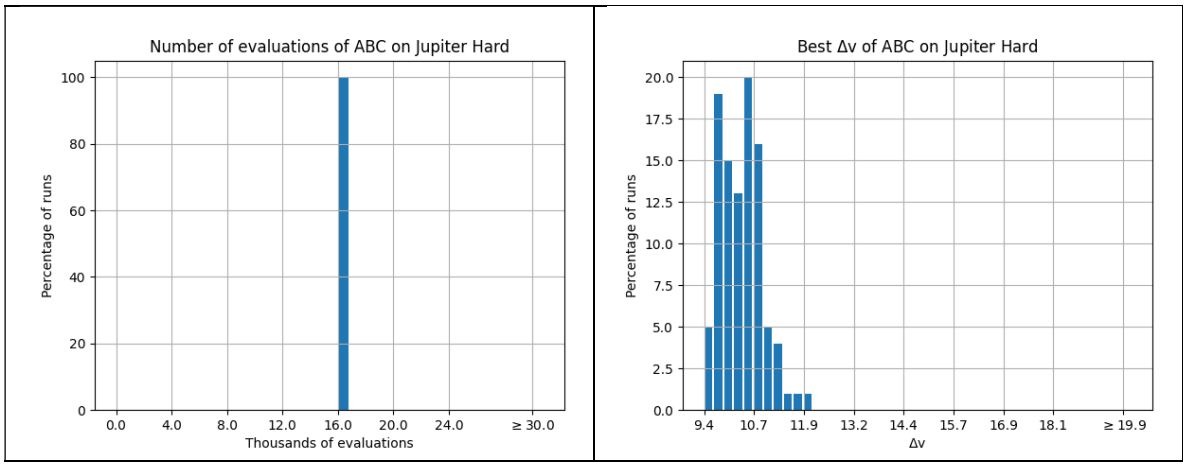

Figure 28 Performance of ABC on Jupiter Hard.

According to Figure 28, ABC performs better than GA and DE on the problem Jupiter Hard, as it never found a value far from the optimal Δv. But it needed about 16000 evaluations on all runs to achieve this performance.

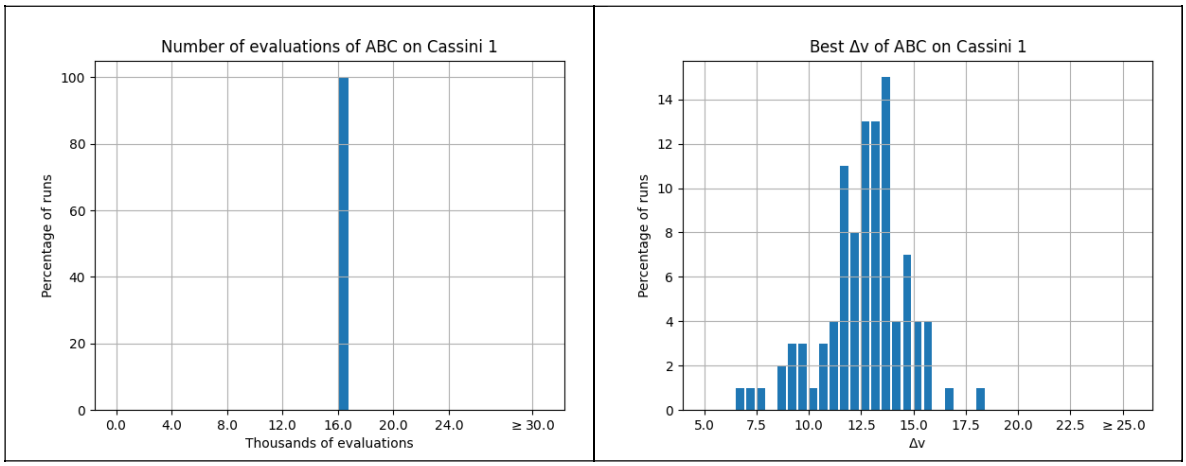

Figure 29 Performance of ABC on Cassini 1.

According to Figure 29, ABC performs worse than GA and DE on the problem Cassini 1, as in 89 % of runs it got a value of Δv of at least 10 km/s. We should also remember that ABC did a fixed amount of evaluation of the objective function. The table below summarizes the performance of the algorithm ABC:

Table 9 Average performance of ABC on test problems.

	Average number of Evaluations	Average Δv found in km/s	$\Delta v_{avg} - \Delta v_{best}$ in km/s
Jupiter Easy	16512	12.967	0.000
Jupiter Hard	16512	10.337	0.917
Cassini 1	16512	12.718	7.788

Comparing Table 6 and Table 9, in particular for the Cassini 1 problem, where the number of evaluations are comparable, we can see that DE performed better than ABC even with less evaluations on average.

6.1.7 NSGA-II single-objetive

We optimized each test problem using the NSGA-II algorithm 100 times. The NSGA-II implementation used in this paper was provided by the Python library pymoo [31] version 0.6.0. The NSGA-II implementation on pymoo requires the population size as a parameter. We used the default value of population size = 100. The initial population is sampled using random uniform distribution from the problem bounds. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization) and best Δv found per run (run meaning we executed the optimization algorithm).

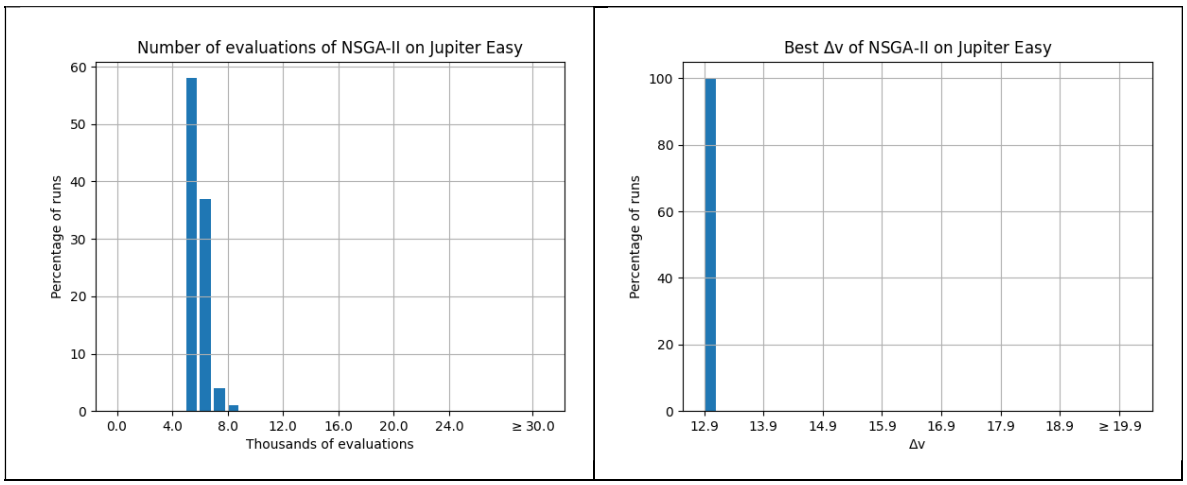

Figure 30 NSGA-II performance on Jupiter Easy single-objective.

The results shown on Figure 30 are similar to that of Figure 21 (GA), as expected.

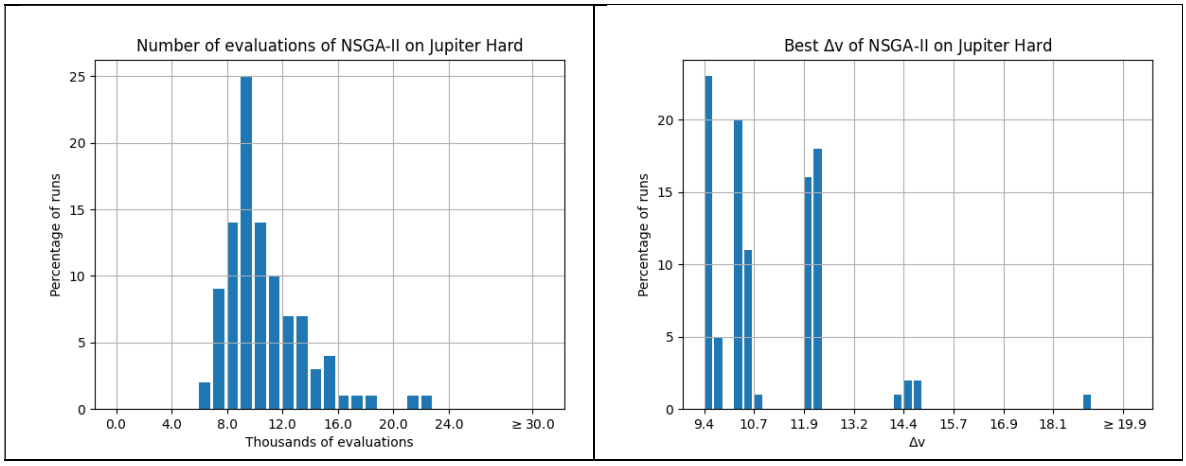

Figure 31 NSGA-II performance on Jupiter Hard single-objective.

Again, we found similar results between GA and NSGA-II comparing Figure 22 and Figure 31.

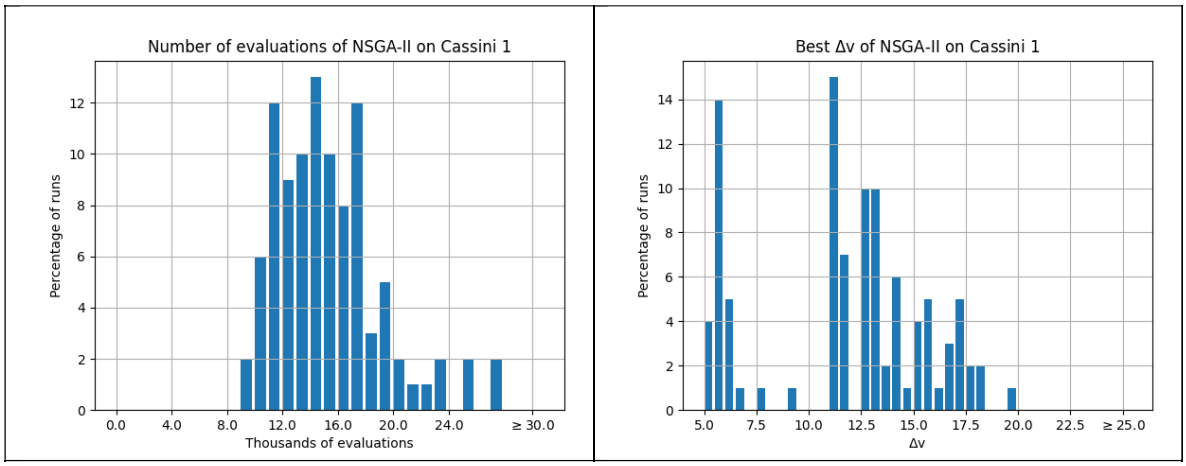

Figure 32 NSGA-II performance on Cassini 1 single-objective.

NSGA-II performance in problem Cassini 1 was really similar to that of GA and DE, making it hard to draw conclusions just by looking at the graphics. We need the average performance of each to provide a fair comparison:

Table 10 Average performance of NSGA-II on test problems.

	Average number of Evaluations	Average Δv found in km/s	$\Delta v_{avg} - \Delta v_{best}$ in km/s
Jupiter Easy	5977.00	12.967	0.000
Jupiter Hard	10791.00	11.066	1.646
Cassini 1	15350.00	11.773	6.843

NSGA-II performed better than DE on Jupiter Hard, but worse on Cassini 1, having a comparable number of evaluations on each. Because Cassini 1 is the harder problem and NSGA-II performed significantly worse than DE, we chose DE as the preferred algorithm between the two. NSGA-II had really similar results to GA and it needed less function evaluations, so we set NSGA-II as the preferred algorithm over GA, although both have really close performance.

6.2 Multi-objective optimizers

To assess the chosen multi-objective optimizers, we compared the number of evaluations (that is, how many times we called the objective function) and the quality of the results of the optimizers. To compare the quality of the results we analyzed two graphics: one that shows the obtained pareto fronts (that is, the pareto optimal outputs) merged after 100 runs and another one that shows the best Δv of each run (just like in the previous sections). At least one of the pareto optimal solutions minimizes the problem Δv, so a good multi-objective minimizer must also find a good solution for the single-objective case, that is why we also provided a graph of the best Δv of each run. The best known solution shown in each pareto front graph was obtained after running.

6.2.1 MOPSO

We optimized each test problem using the MOPSO algorithm 100 times. The MOPSO implementation used in this paper was provided by the Python library pygmo [35] version 2.19.5. The MOPSO implementation on pygmo accepts as parameters the population size, the number of generations, Ω (inertia weight), c_1 (magnitude of the force, applied to the particle's velocity, in the direction of its previous best position), c_2 (magnitude of the force, applied to the particle's velocity, in the direction of its global best position), χ (scaling factor), v_{coef} (velocity coefficient), leader selection range and diversity mechanism. We set the population size to 128 and the number of generations to 128. The remaining parameters were left as default, so we have $\Omega = 0.6$, $c_1 = 0.01$, $c_2 = 0.5$, $\chi = 0.5$, $v_{coef} = 0.5$, leader selection range = 2 and diversity mechanism = 'crowding distance'. Because we fixed the number of generations, the number of evaluations of the objective function will be fixed. We chose a number of generations that lead to a number of evaluations comparable to the algorithm NSGA-II on the problems Jupiter Hard and Cassini 1, so we can compare the algorithms more fairly. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization), best Δv found per run (run meaning we executed the optimization algorithm) and a scatter graph of the pareto front found by the algorithm.

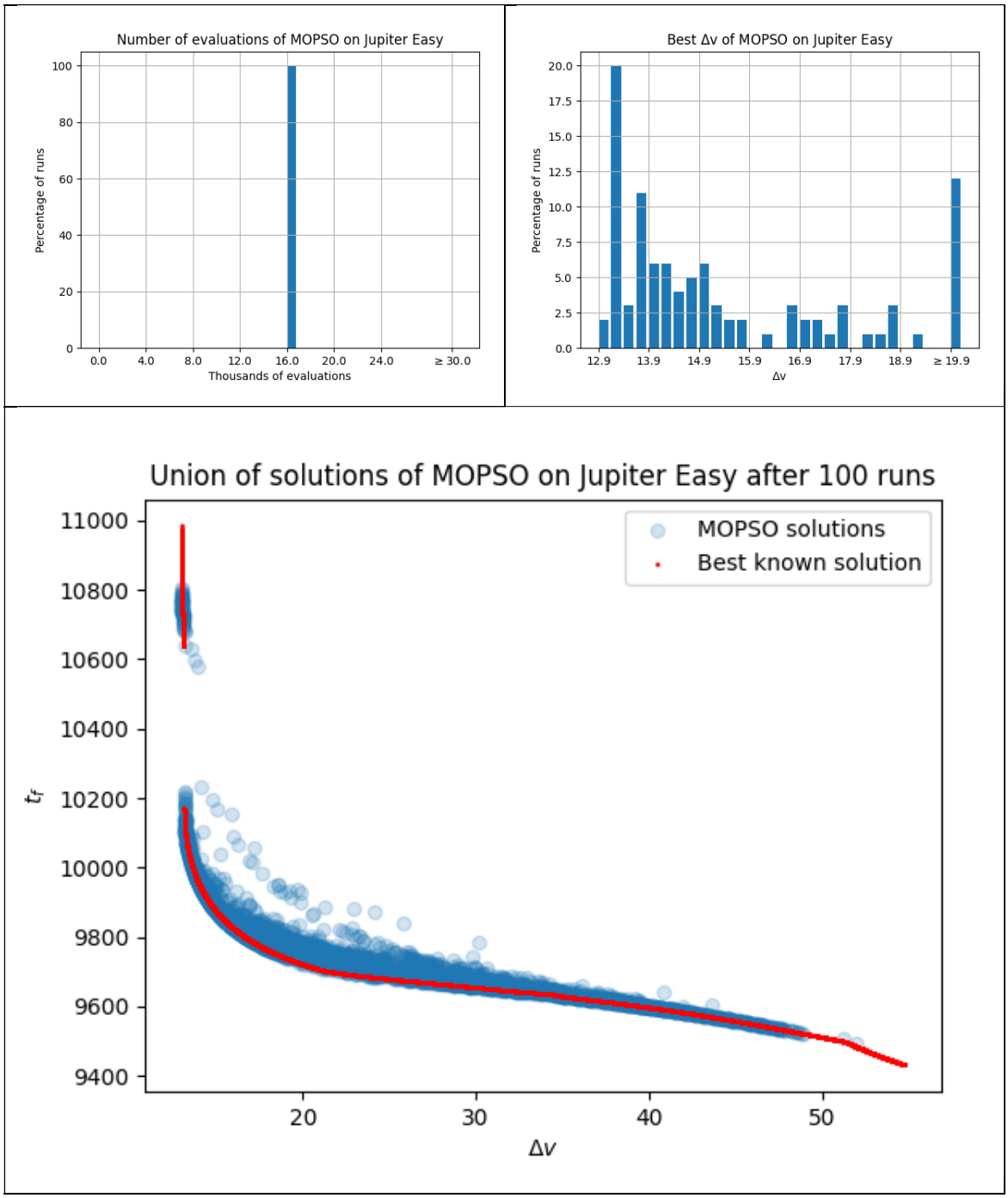

Figure 33 Performance of MOPSO on Jupiter Easy.

According to Figure 33 the optimizer MOPSO was able to find the pareto front in some runs. It had difficulty in finding the values that minimize the problem's Δv in some runs.

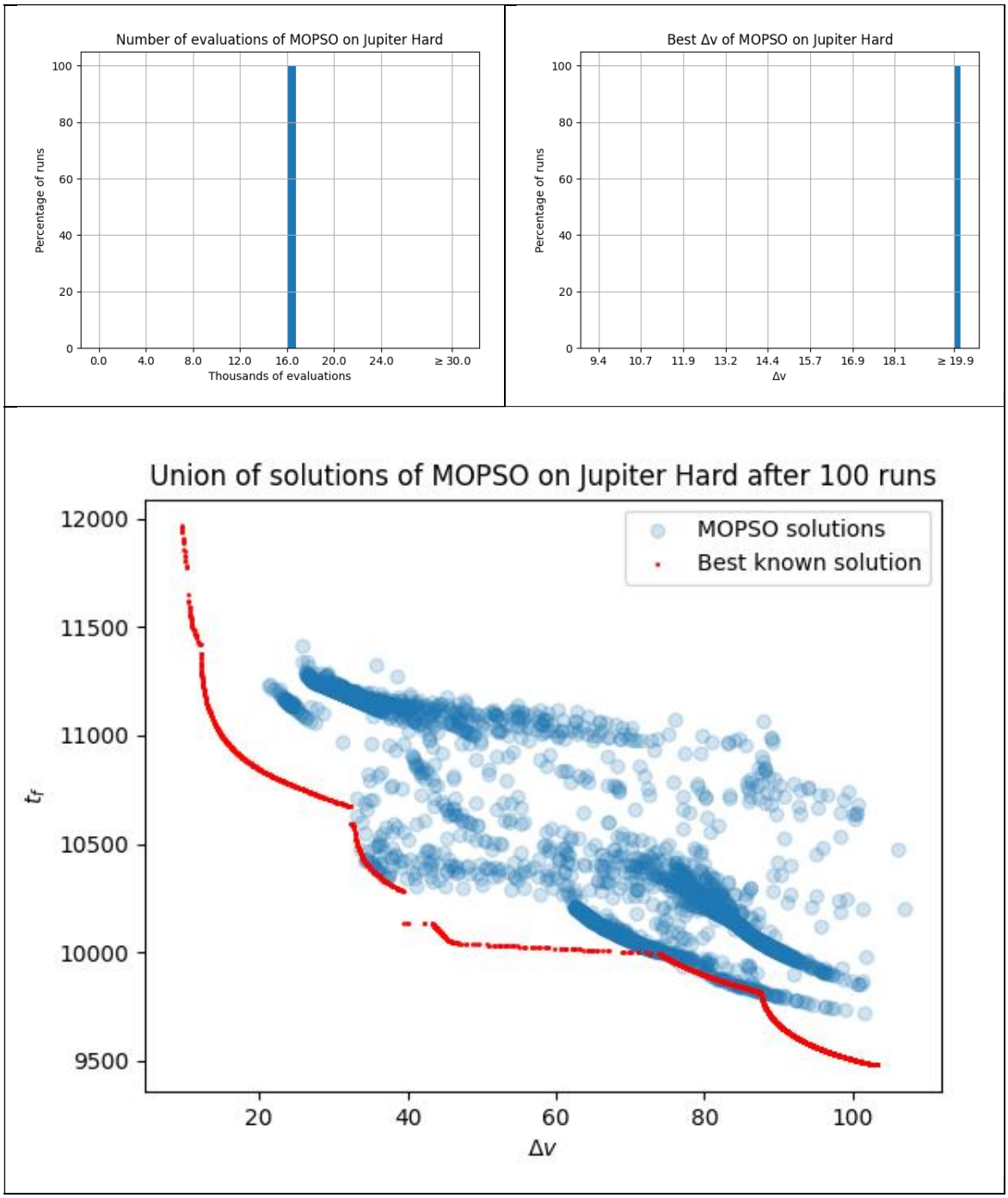

Figure 34 Performance of MOPSO on Jupiter Hard.

According to Figure 34 the optimizer MOPSO was unable to find the pareto front of the problem, especially in points close to the minimum Δv.

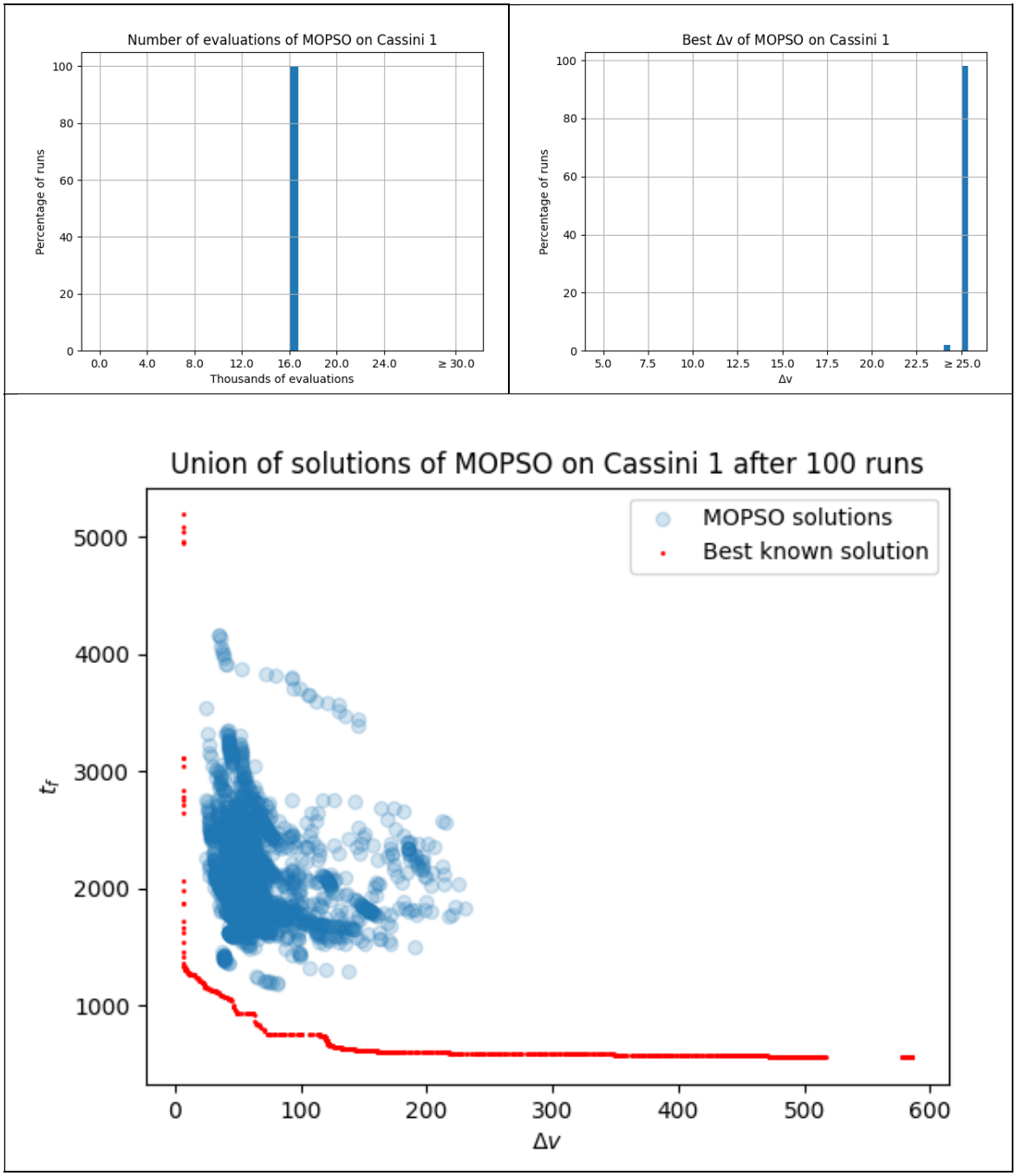

Figure 35 Performance of MOPSO on Cassini 1.

According to Figure 35 the optimizer MOPSO performed very poorly in the problem Cassini 1, even though it had 16512 evaluations of the objective function in each run (that is, each time we executed the optimization process).

6.2.2 NSGA-II

We optimized each test problem using the NSGA-II algorithm 100 times. The NSGA-II implementation used in this paper was provided by the Python library pymoo [31] version 0.6.0. The NSGA-II implementation on pymoo requires the population size as a parameter. We used the default value of population size = 100. The initial population is sampled using random uniform distribution from the problem bounds. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization), best Δv found per run (run meaning we executed the optimization algorithm) and a scatter graph of the pareto front found by the algorithm.

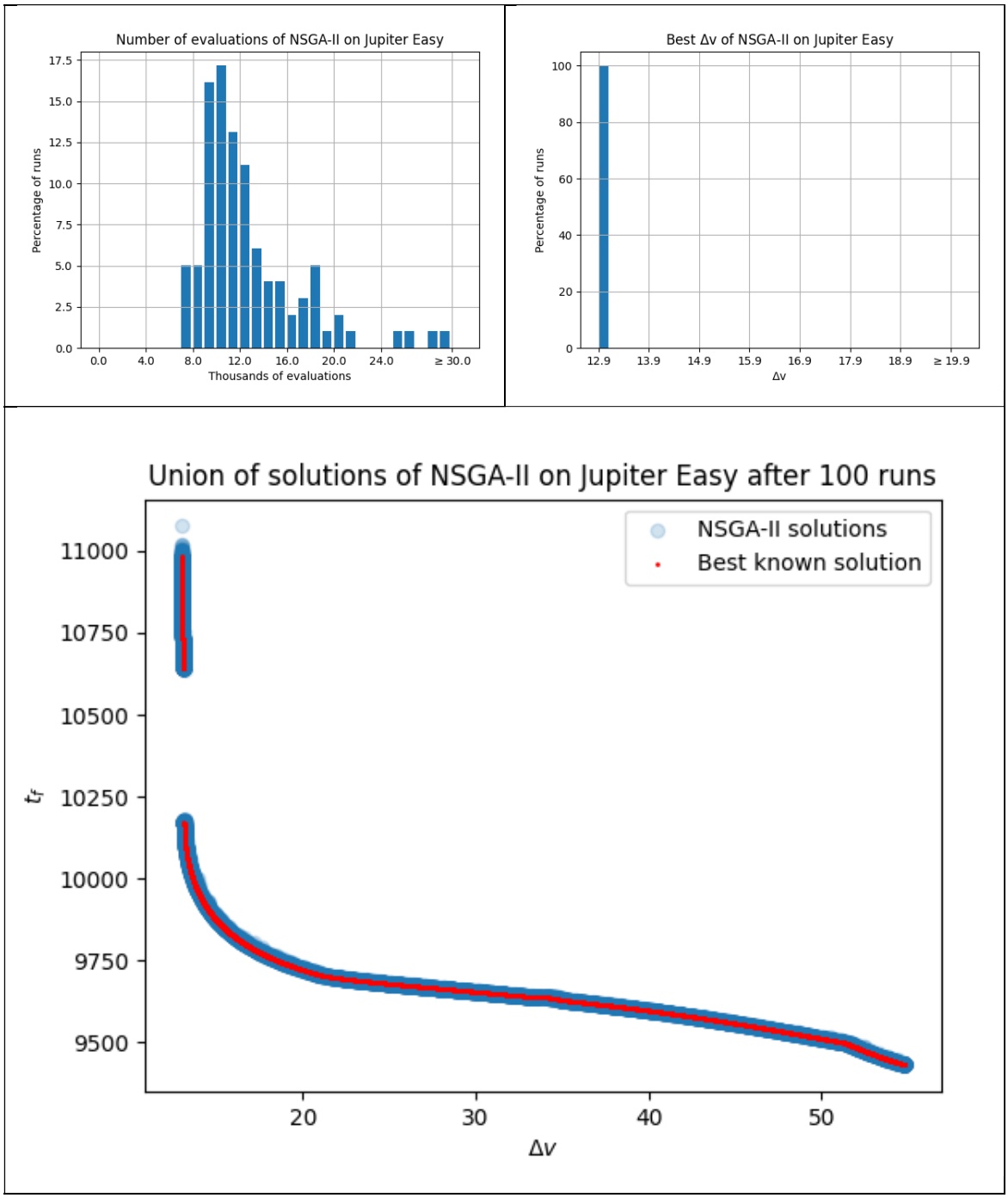

Figure 36 Performance of NSGA-II on Jupiter Easy multi-objective.

According to Figure 36 the optimizer NSGA-II was able to successfully find the pareto front in all runs, even needing less evaluations of the objective function when compared to MOPSO, as can be seen in Figure 33.

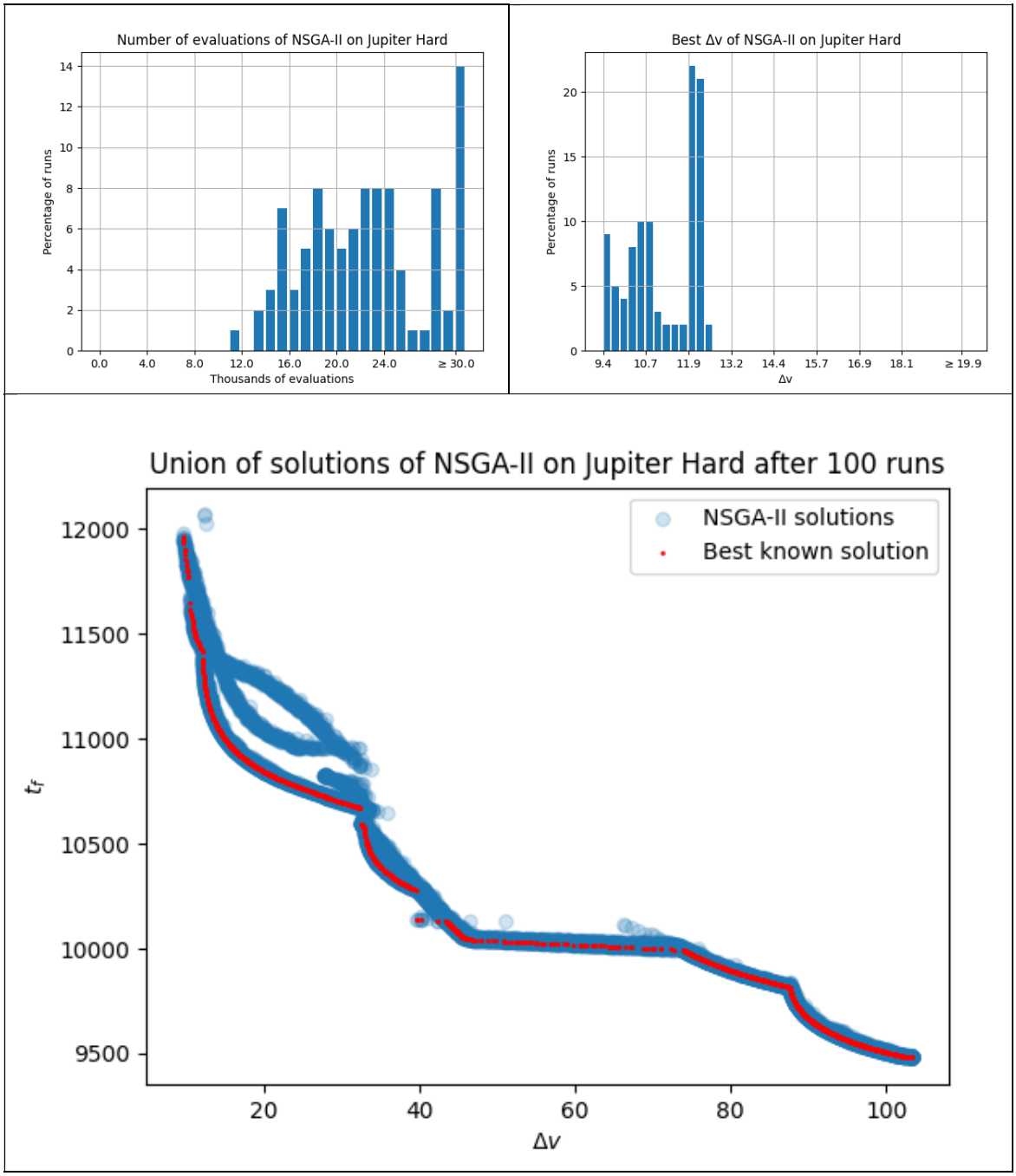

Figure 37 Performance of NSGA-II on Jupiter Hard multi-objective.

On average, we can see the algorithm NSGA-II was able to find the pareto front of the problem Jupiter Hard, unlike the previous analyzed algorithm MOPSO.

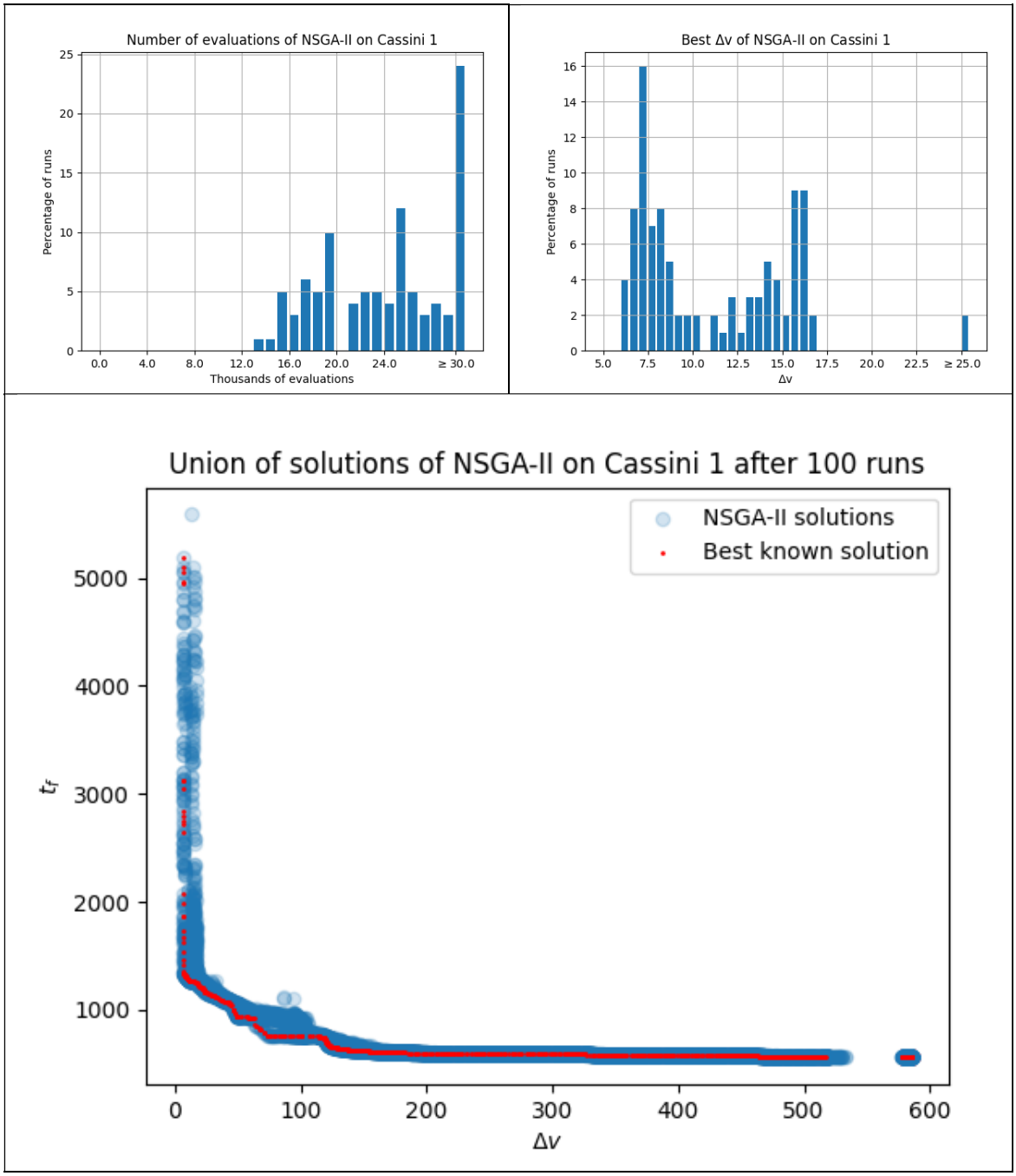

Figure 38 Performance of NSGA-II on Cassini 1 multi-objective.

NSGA-II was able to fit well to the pareto front of the problem Cassini 1, although it had difficulty in finding the minimum Δv for the problem.

6.2.3 MOEA/D

We optimized each test problem using the MOEA/D algorithm 100 times. The MOEA/D implementation used in this paper was provided by the Python library pymoo [31] version 0.6.0. The MOEA/D implementation on pymoo requires reference directions, number of neighbors and probability of mating to instantiate the algorithm. We used 24 reference directions uniformly distributed in a 2D circle, 15 neighbors and probability of mating = 0.7. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization), best Δv found per run (run meaning we executed the optimization algorithm) and a scatter graph of the pareto front found by the algorithm.

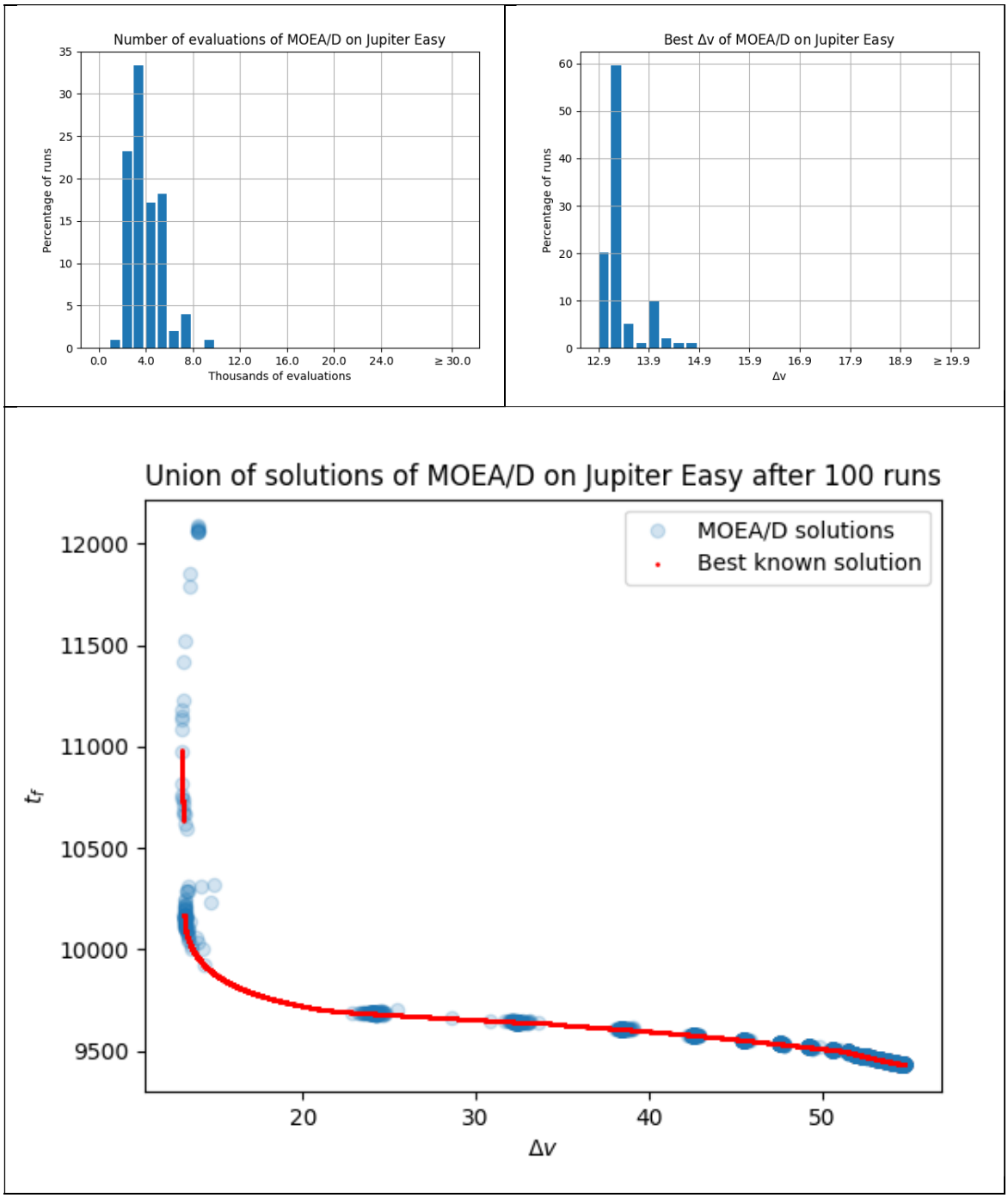

Figure 39 Performance of MOEA/D on Jupiter Easy.

According to Figure 39, MOEA/D found an incomplete pareto front during runs. Notice it also needed less evaluations than the previous algorithms to converge.

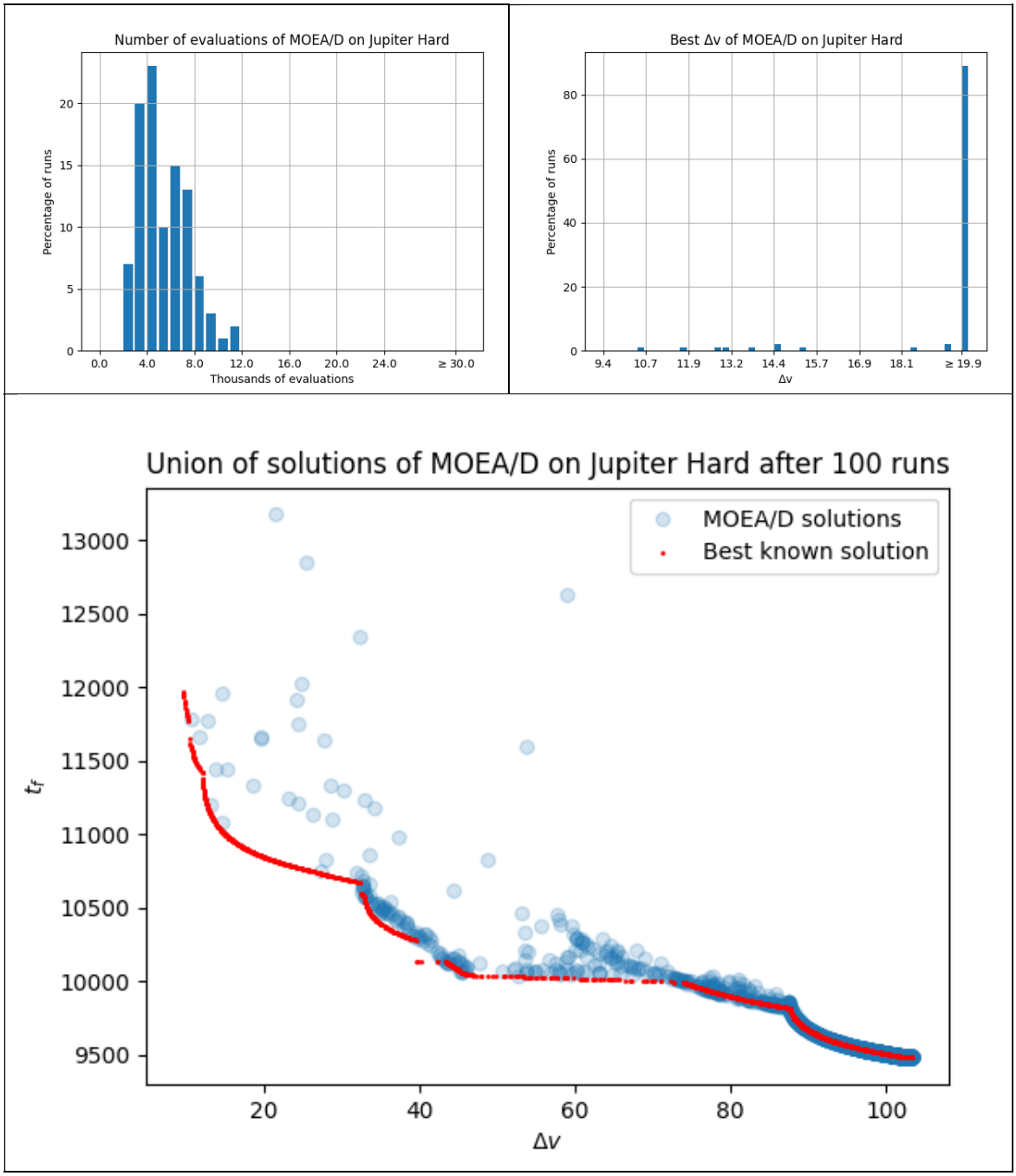

Figure 40 Performance of MOEA/D on Jupiter Hard.

According to Figure 40, MOEA/D had trouble finding the points of the pareto front that minimize the mission Δv. It converged earlier than the previously analyzed algorithms.

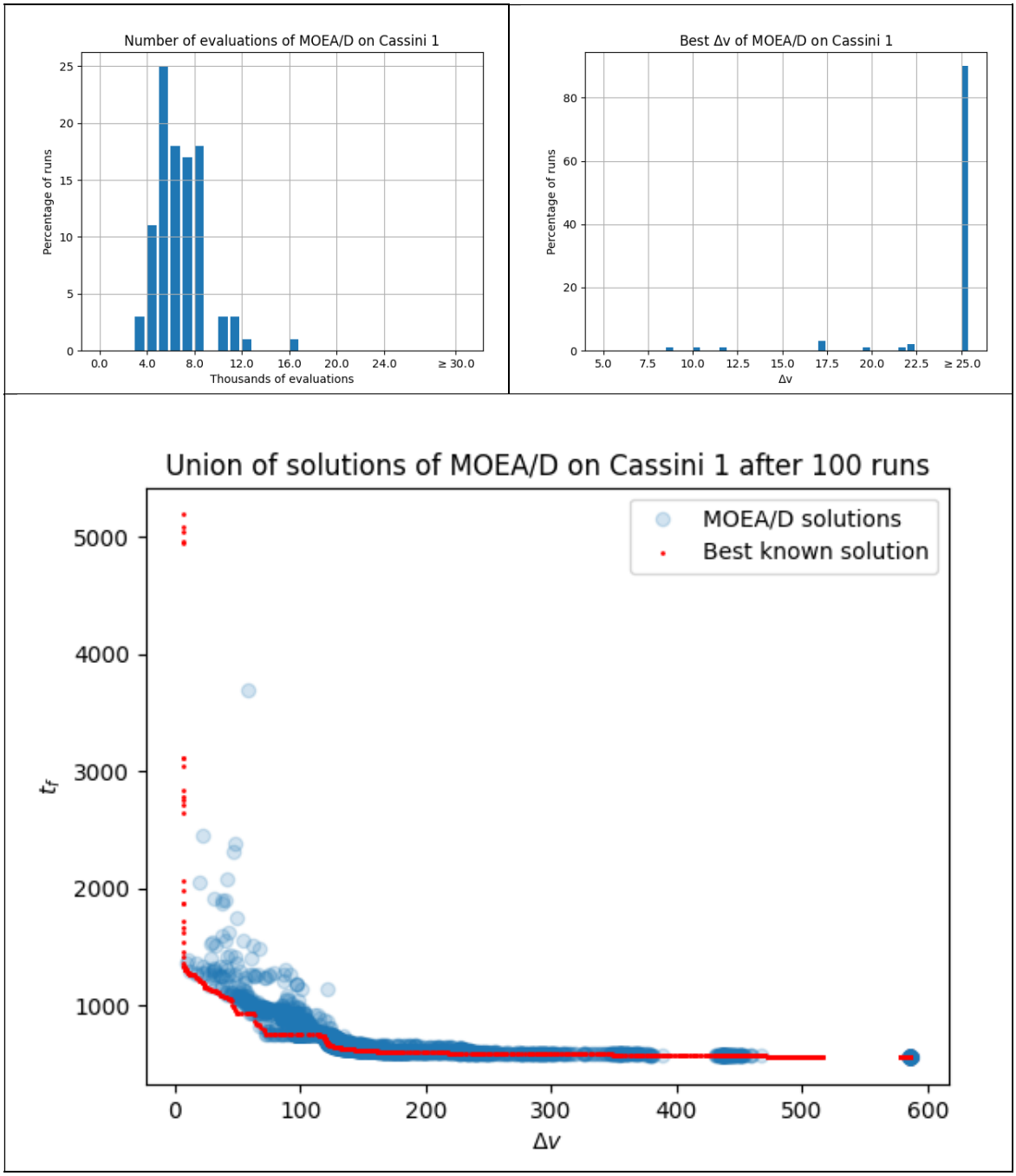

Figure 41 Performance of MOEA/D on Cassini 1.

Again, MOEA/D had trouble finding the points of the pareto front that minimize the mission Δv. It should be noted that MOEA/D took less evaluations to converge than the previous two algorithms on all problems and had results comparable to MOPSO.

6.2.4 MHACO

We optimized each test problem using the MHACO algorithm 100 times. The MHACO implementation used in this paper was provided by the Python library pygmo [35] version 2.19.5. The MHACO implementation on pygmo accepts as parameters the population size, the number of generations, the kernel size, the convergence seed parameter, the threshold paremeter, the standard convergence speed parameter, evaluation stop criteria and focus parameter. We set the population size to 128 and the number of generations to 128. The remaining parameters were left as default, so we have kernel size = 63, convergence speed parameter = 1.0, threshold parameter = 1, standard convergence speed parameter = 7, evaluation stopping criterion = 10’000 and focus parameter = 0. Because we fixed the number of generations, the number of evaluations of the objective function will be fixed. We chose a number of generations that lead to a number of evaluations comparable to the algorithm NSGA-II on the problems Jupiter Hard and Cassini 1, so we can compare the algorithms more fairly. Below are bar graphs for the number of evaluations (that is, how many times the objective function is called during the optimization), best Δv found per run (run meaning we executed the optimization algorithm) and a scatter graph of the pareto front found by the algorithm.

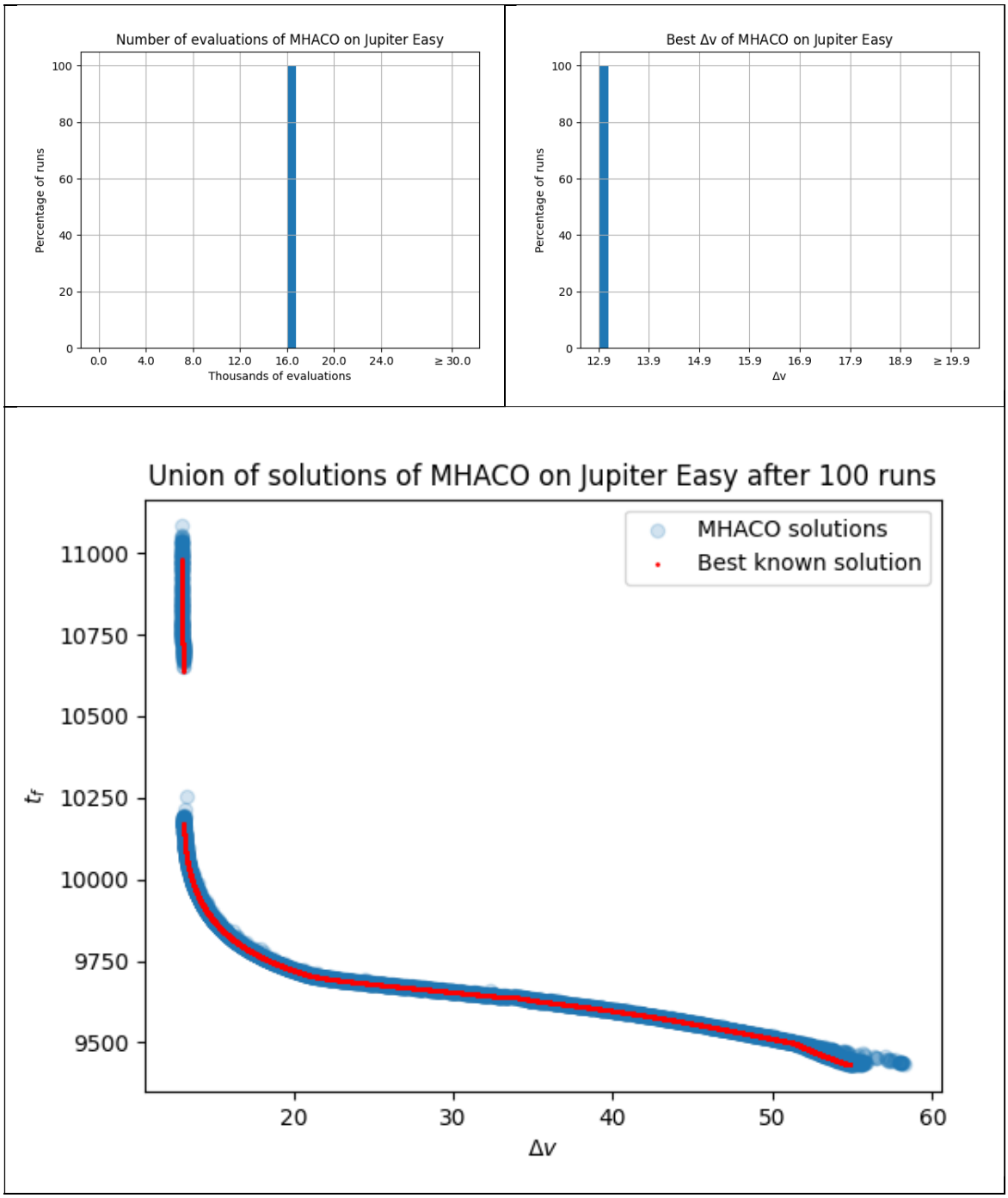

Figure 42 Performance of MHACO on Jupiter Easy.

Comparing Figure 42 and Figure 36, we see the MHACO algorithm performed almost as well as NSGA-II.

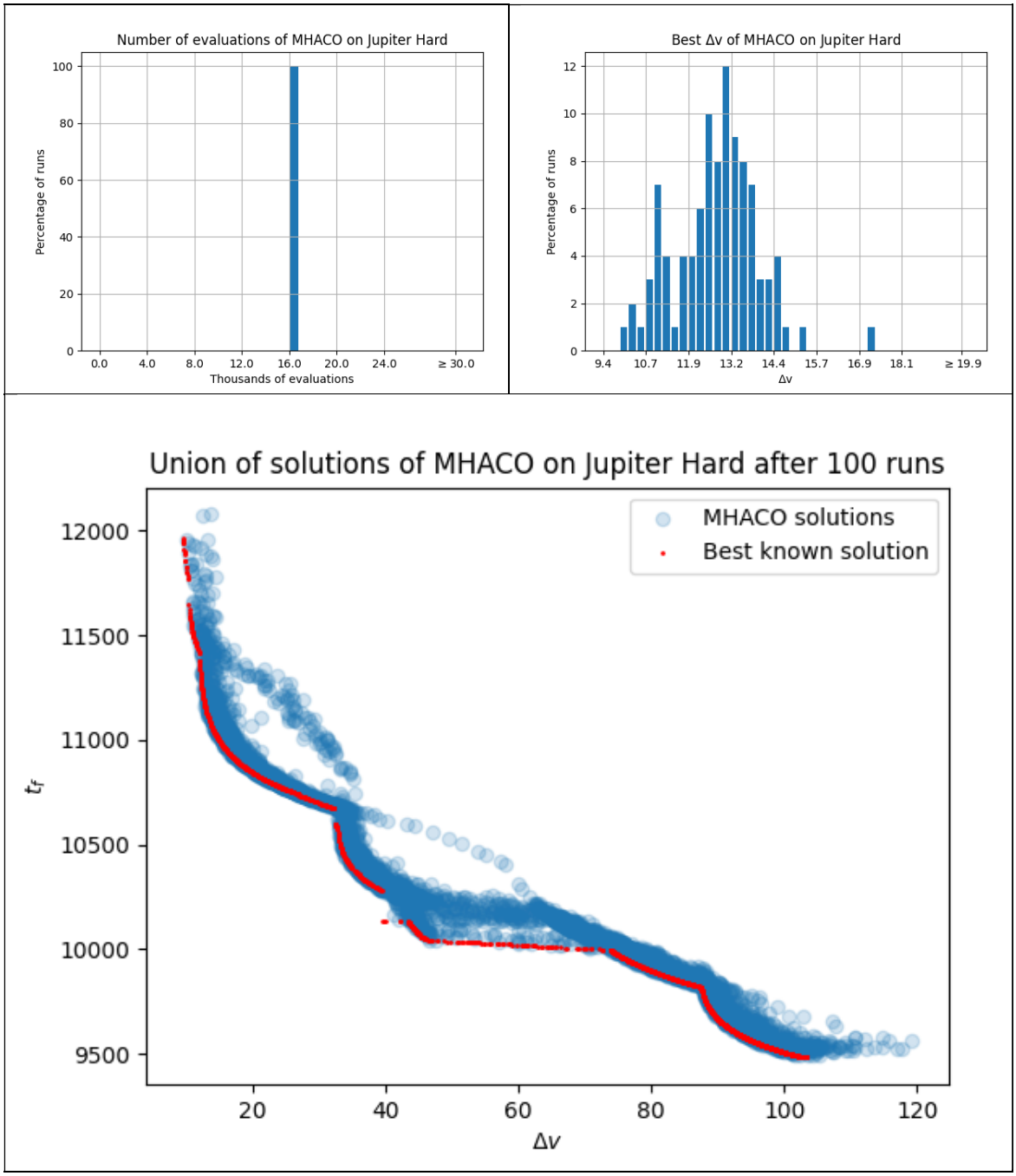

Figure 43 Performance of MHACO on Jupiter Hard.

Comparing Figure 43 and Figure 37 we notice that NSGA-II had considerable better performance on problem Jupiter Hard. It should be noted that NSGA-II had more function evaluations on average.

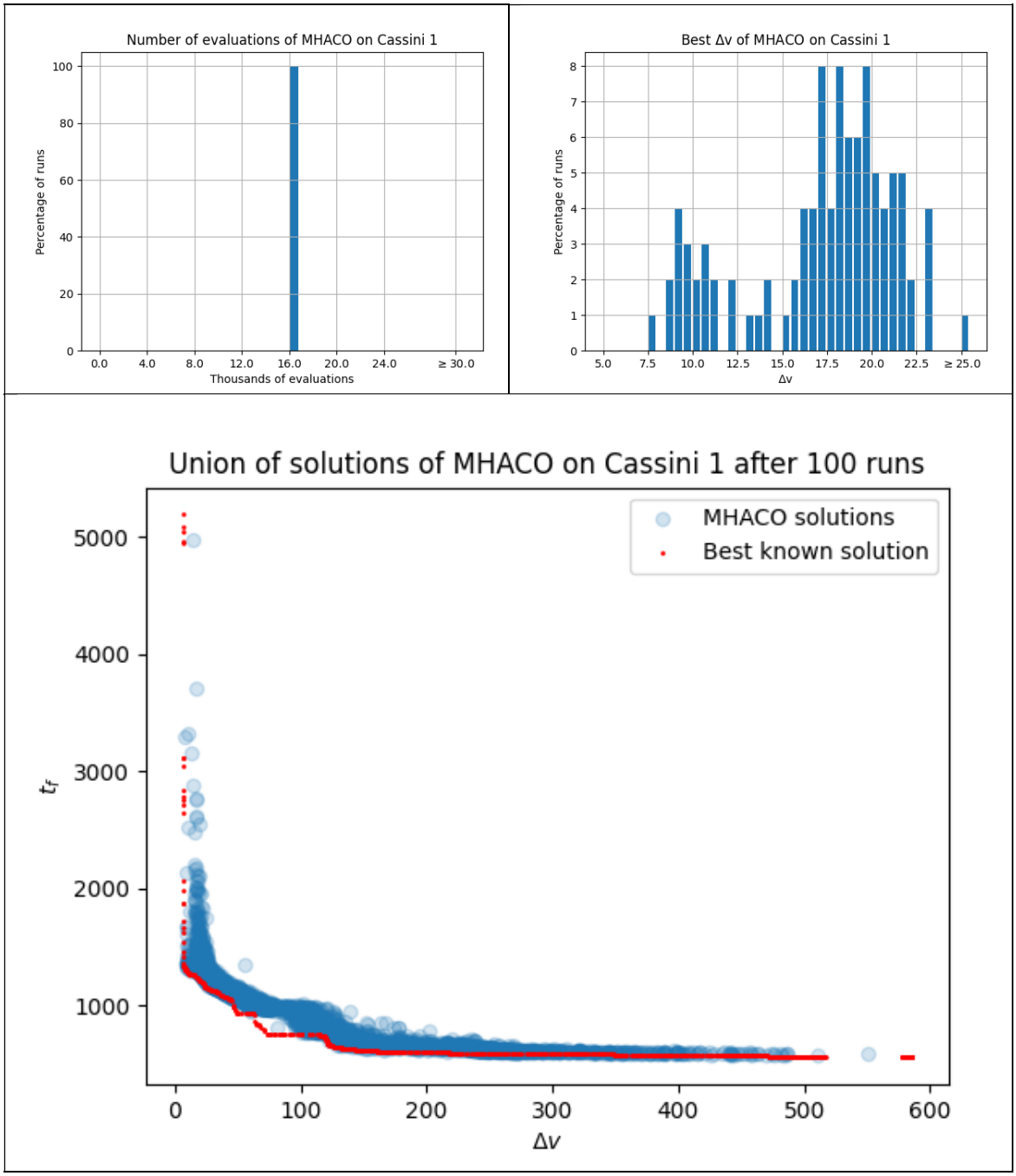

Figure 44 Performance of MHACO on Cassini 1

Comparing Figure 44 and Figure 38 we notice that NSGA-II had considerable better performance on problem Cassini 1, especially for finding the points that minimize the mission Δv. It should be noted that NSGA-II had more function evaluations on average. That said, we conclude that NSGA-II is the best optimizer for multi-objective optimization and MHACO is a close second-place. MOEA/D needs much less evaluations to converge and its performance is worse than those two. Finally, MOPSO needed as much evaluations as MHACO to finish and

had bad performance. We encourage the reader to check the GitHub repository and run your own tests: https://github.com/fbrunodr/TestOptimizersOnSpaceTrajectoryProblems.

7 Conclusion

As highlighted in section 6:

- **Single-objective algorithms:**
  - DE has the best performance when it comes to the quality of solution (minimizing the objective function).
  - NSGA-II has a similar performance to DE, although it performed worse on the harder problem.
  - GA has a performance comparable to that of NSGA-II, although it is placed worse because it needed more function evaluations to converge on average.
  - SA needed much fewer evaluations to converge than the other algorithms, but it is also more susceptible to getting stuck on local minima. This is expected behavior.
  - The remaining single-objective algorithms are Pareto dominated by DE and SA regarding the number of evaluations and quality of solution, so we suggest that you do not use them if you have the option.
- **Multi-objective algorithms:**
  - NSGA-II has the best performance when it comes to quality of solution.
  - MHACO is a close second place compared to NSGA-II.
  - MOEA/D needs significantly less evaluations than NSGA-II and MHACO to converge and it had worse performance finding the pareto front.
  - MOPSO performed poorly on the tests. We suggest that you do not use it.

With that said, we recommend using either DE, NSGA-II or GA for single-objective optimization. If very few evaluations is a must SA can be a great alternative. For multi-objective optimization NSGA-II and MHACO are the suggested algorithms. For future work, we suggest:

- Study how changing the algorithms' parameters may influence their performance.
- Test the optimizers presented in this paper on a more diverse set of test problems, particularly more complex problems.
- Test more optimizers.

8 Bibliography

[1] U.S. government, "What is GPS?," [Online]. Available: https://www.gps.gov/systems/gps/. [Accessed 3 July 2023].

[2] SpaceX, "World's most advanced broadband satellite internet," SpaceX, [Online]. Available: https://www.starlink.com/technology. [Accessed 3 July 2023].

[3] NASA, "Satellite Sensors Would Deliver Global Fire Coverage," NASA, 19 November 2015. [Online]. Available: https://www.nasa.gov/feature/satellite-sensors-would-deliver-global-fire-coverage.

[4] McKinsey&Company, "How will the space economy change the world?," McKinsey&Company, 28 November 2022. [Online]. Available: https://www.mckinsey.com/industries/aerospace-and-defense/our-insights/how-will-the-space-economy-change-the-world.

[5] Reuters, "Apple makes history as first $3 trillion company amid tech stock surge," Reuters, 3 July 2023. [Online]. Available: https://www.reuters.com/technology/global-markets-apple-2023-07-03/.

[6] C. Dreier, "An Improved Cost Analysis of the Apollo Program," *Space Policy, Volume 60,* 2022.

[7] J. Wertz, D. Everett and J. Puschell, Space Mission Engineering - The New Smad, Space Technology Library, Vol. 28, 2011.

[8] A. L. De Lima dos Santos, "Study of multiple-stage trajectories from the SL1 to SL4 points using solar sails," Instituto Tecnológico de Aeronáutica, São José dos Campos, 2022.

[9] IEEE, "IEEE Xplore," IEEE, [Online]. Available: https://ieeexplore.ieee.org/Xplore/home.jsp.

[10] Google, "Google Scholar," [Online]. Available: https://scholar.google.com/.

[11] Springer, "Springer Link," [Online]. Available: https://link.springer.com/.

[12] N. Corporation, "Cuda Zone," 2023. [Online]. Available: https://developer.nvidia.com/cuda-zone.

[13] G. Acciarini, D. Izzo and E. Mooij, "MHACO: a Multi-Objective Hypervolume-Based Ant Colony Optimizer for Space Trajectory Optimization," *2020 IEEE Congress on Evolutionary Computation (CEC),* pp. 1-8, 2020.

[14] T. Vinkó, D. Izzo and C. Bombardelli, "Benchmarking different global optimisation techniques for preliminary space trajectory design," *58th International Astronautical Congress,* September 2007.

[15] P. D. Lizia and G. Radice, "Advanced Global Optimisation Tools for Mission Analysis and Design," Department of Aerospace Engineering, University of Glasgow, 2004.

[16] Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," *IEEE Transactions on Evolutionary Computation,* pp. 712-731, 2007.

[17] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* pp. 182-197, 2002.

[18] Y. Wang, M. Zhu, Y. Wei and Y. Zhang, "Solar sail spacecraft trajectory optimization based on improved imperialist competitive algorithm," in *10th World Congress on Intelligent Control and Automation,* 2012.

[19] A. Shirazi, J. Ceberio and J. A. Lozano, "Evolutionary algorithms to optimize low-thrust trajectory design in spacecraft orbital precession mission," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, 2017.

[20] S. Samsam and R. Chhabra, "Multi-impulse Shape-based Trajectory Optimization for Target Chasing in On-orbit Servicing Missions," in *2021 IEEE Aerospace Conference (50100)*, 2021.

[21] M. Navabi and E. Meshkinfam, "Space low-thrust trajectory optimization utilizing numerical techniques, a comparative study," in *2013 6th International Conference on Recent Advances in Space Technologies (RAST)*, 2013.

[22] R. Chai, A. Savvaris and A. Tsourdos, "Violation Learning Differential Evolution-Based hp-Adaptive Pseudospectral Method for Trajectory Optimization of Space Maneuver Vehicle," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 2031-2044, 2017.

[23] A. Shirazi, "Multi-objective optimization of orbit transfer trajectory using imperialist competitive algorithm," in *2017 IEEE Aerospace Conference*, 2017.

[24] A. Shirazi, "Analysis of a hybrid genetic simulated annealing strategy applied in multi-objective optimization of orbital maneuvers," *IEEE Aerospace and Electronic Systems Magazine*, pp. 6-22, 2017.

[25] A. Shirazi, J. Ceberio and J. A. Lozano, "Spacecraft trajectory optimization: A review of models, objectives,," *Progress in Aerospace Sciences*, pp. 76-98, 2018.

[26] R. Chai, A. Savvaris, A. Tsourdos, S. Chai and Y. Xia, "A review of optimization techniques in spacecraft flight trajectory design," *Progress in Aerospace Sciences*, 2019.

[27] R. Chai, A. Savvaris and A. Tsourdos, "Solving Multi-objective Aeroassisted Spacecraft Trajectory Optimization Problems Using Extended NSGA-II," *AIAA SPACE and Astronautics Forum and Exposition*, September 2017.

[28] I. Hare, "Infographics: Operation Costs in CPU Clock Cycles," 12 September 2016. [Online]. Available: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/. [Accessed 9 October 2023].

[29] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, 1995.

[30] Ephramac, "Particle swarm optimization," [Online]. Available: https://en.wikipedia.org/wiki/Particle_swarm_optimization. [Accessed 14 11 2023].

[31] J. Blank and K. Deb, "pymoo: Multi-Objective Optimization in Python," *IEEE Access*, vol. 8, pp. 89497-89509, 2020.

[32] R. Storn and K. Price, "A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization*, vol. 11, p. 341–359, 1997.

[33] R. Matsunaga, "Genetic Algorithm Walkers," redhunt.org, [Online]. Available: https://rednuht.org/genetic_walkers/. [Accessed 14 November 2023].

[34] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671 - 680, 1983.

[35] F. Biscani and D. Izzo, "A parallel global multiobjective framework for optimization: pagmo," *Journal of Open Source Software*, vol. 5, p. 2338, 2020.

[36] Karaboğa, "Artificial bee colony algorithm," *Scholarpedia*, vol. 5, p. 6915, 2010.

[37] C. Coello Coello and M. and Lechuga, "MOPSO: a proposal for multiple objective particle swarm optimization," in *Proceedings of the 2002 Congress on Evolutionary Computation*, 2002.

[38] E. A. C. Team, "Cassini 1," 30 April 2013. [Online]. Available: https://www.esa.int/gsp/ACT/projects/gtop/cassini1/. [Accessed 13 November 2023].

[39] S. Man, "Why is the bend angle of a hyperbolic trajectory giving different results?," 31 March 2020. [Online]. Available: https://space.stackexchange.com/questions/43165/why-is-the-bend-angle-of-a-hyperbolic-trajectory-giving-different-results.

[40] Tosic, "Calculating object velocity at perihelion," 22 November 2019. [Online]. Available: https://astronomy.stackexchange.com/questions/34041/calculating-object-velocity-at-perihelion.

[41] D. D. R. Williams, "Jupiter Fact Sheet," NASA Goddard Space Flight Center, 22 May 2023. [Online]. Available: https://nssdc.gsfc.nasa.gov/planetary/factsheet/jupiterfact.html. [Accessed 13 11 2023].

[42] D. D. R. Williams, "Sun Fact Sheet," NASA Goddard Space Flight Center, 17 November 2022. [Online]. Available: https://nssdc.gsfc.nasa.gov/planetary/factsheet/sunfact.html. [Accessed 13 November 2023].

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. REGISTRO N°	4. N° DE PÁGINAS
TC	23 de novembro de 2023	DCTA/ITA/TC-128/2023	83

5. TÍTULO E SUBTÍTULO:

Efficient algorithms for solving optimization problems in aerospace trajectories

6. AUTOR(ES):

Francisco Bruno Dias Ribeiro da Silva

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

1. Optimization. 2. Optimizer. 3. Aerospace. 4. Trajectory.

9.PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Otimização; Algoritmos; Trajetórias; Problemas de dois corpos; Mecânica celeste; Engenharia aeroespacial.

10. APRESENTAÇÃO: **(X) Nacional ( ) Internacional**

ITA, São José dos Campos. Curso de Graduação em Engenharia Aeroespacial. Orientadora: Profa. Dra. Maísa de Oliveira Terra. Publicado em 2023.

11. RESUMO:

The technological demands of the aerospace field make it one of the costliest industries in the world. It is then of uttermost importance to optimize every aspect of an aerospace mission, from spaceship hardware design to the mission trajectory. Optimizing the trajectory usually requires setting a finite number of parameters. For some cases, such as the 2-body problem, we can get a closed form solution for the spaceship trajectories. Optimizing the parameters is then a matter of solving some closed-form equations. In other cases, however, an optimizer is needed to find the optimal parameters of a space flight. This work analyses commonly used optimizers in literature for solving problems related to aerospace trajectories. It then applies those optimizers to some test problems and reviews the performance of each one of them by comparing the solution found and the number of evaluations to get such solution. The aim of this work is to set a standard of what are the best optimizers to solve trajectory problems in the aerospace field and the trade-offs between each of them. We found that, considering the investigated algorithms and the implementations available in the researched libraries, DE, NSGA-II, and GA are the best optimizers for single-objective problems, while NSGA-II and MHACO are the best optimizers for multi-objective problems.

12. GRAU DE SIGILO:

(X) OSTENSIVO ( ) RESERVADO ( ) SECRETO