Three-Body Problem

App documentation

2025 | Konrad Flis | MIT Licence

updated on February 15th, 2025

Table of contents

1. Introduction

1.1 Documentation

This documentation aims to explain all app components and applications in the most comprehensive and practical way. It includes the context app was created in, applied swarm algorithms, parameters' selection and app usage.

1.2 Context

Initial Orbit Determination (IOD) is a process of establishing a body's orbit properties with a limited number of measurements in time. An object, like a satellite, can be monitored and controlled based on the sparse observations of its state (positions and/or velocities). With just a few measurements and assumptions on the gravitational forces, one can estimate the initial state and further propagate it in time to get complete information about the body's predicted movement.

There are many different approaches to the problem, and their usage depends on the available data. In my model, the expected orbits' positions and velocities in time are known, as they come from NASA database[1]. In such a case, there is no need to include an additional measurement mechanism or to change the units from metric to angular - states and velocities are given in [km] and [km/s] respectively. In a real-life scenario, it would be necessary to include the additional radar or tracking satellite, to estimate the angles and measurement errors. The gravitational forces can be included if assumptions on the bodies affecting the satellite are made. This simulation uses the Circular Restricted Three-Body Problem (CR3BP) movement equations. Both Moon and Earth determine the motion of the satellite. center of mass is the initial reference point.

This app uses three **swarm optimization algorithms** to find the initial states:

- Particle Swarm Optimization,
- Two-stage Particle Swarm Optimization (my own approach, using the original algorithm),
- Artificial Bee Colony Algorithm.

The choice was based on the model's and objective function's properties. This six-dimensional space is full of local minimums and reacts abruptly to even minor changes in the initial conditions. The algorithms are expected to both explore the space and exploit the most promising initial states, to be able to determine the initial state and the initial orbit itself.

[1] https://ssd.jpl.nasa.gov/tools/periodic_orbits.html

1.3 Dictionary

The list below can be a helpful tool to better understand the problem, applied algorithms and their parameters.

Particle Swarm Optimisation (PSO) - swarm intelligence algorithm inspired by nature. Particles are a representation of animals like birds, fish or even humans, interacting with each other to explore the space. The algorithm uses particle as a point in the solution space. It moves based on its own best state and the global best state of all particles in the swarm. This behaviour combines exploration of the solution space with exploitation of the best found state(s). Typical parameters are : **inertia** (impact of particle's current velocity on its movement in the next iteration, taking into account the change caused by global best state and its own best state), **cognitive coefficient** (random factor weighting the impact of particle's best state on its movement) and **social coefficient** (random factor weighting the impact of global best state on each particle's movement). You can find more details in the article[2].

Artificial Bee Colony (ABC) – swarm intelligence algorithm inspired by bees behaviour. Bees tend to organise their swarm by dividing into groups. **Employee bees** explore the food sources (equivalent of points in solution space). **Onlooker bees** overlook the employee bees and their sources, indicating the most promising ones (best solutions). If a food source is no longer profitable (exploitation doesn't bring any important progress with regard to objective function values), it is dropped, and an employee bee becomes a **scout bee**, looking for a new source to exploit. You can find more details in the article[3].

Objective function – a function measuring the quality of solution with regard to predefined criteria. Solutions are compared based on the objective function values.

Neighbourhood – a space around a solution (point in *n*-dimensional space) within predefined limits. Neighbourhood can be defined in many ways. In the presented model, there are limits with regards to position (that is : a neighbouring solution cannot differ more than *n* in terms of its position) and velocity. An example : for a given point (10, 10, 20, 0.1, 0.2, -0.1) with position limit of 2 and velocity limit of 0.1, a neighbour can take states such as (11, 10, 18, 0, 0.15, -0.03) or (10, 10.5, 18.5, 0.15, 0.2, 0).

[2] Kennedy, J.; Eberhart, R. (1995). "Particle Swarm Optimization". *Proceedings of IEEE International Conference on Neural Networks*. Vol. IV. pp. 1942–1948.
[3] Karaboğa, Derviş (2005). "An Idea Based on Honey Bee Swarm For Numerical Optimization". S2CID 8215393

2. Application

2.1 Project structure

```
Three-body-problem/
|— sources/
        # Algorithms - logic, useful data structures, and data processing
|   ├— abc_alg.py
        # Artificial Bee Colony implementation
|   ├— common_elements.py
        # Structures, classes, etc. common for all the algorithms - scalable
|   ├— data_load.py
        # Orbit data preprocessing, unit converters
|   ├— data_structures.py
        # Useful data structures to handle algorithms' inputs and outputs
|   ├— plot_functions.py
        #  Functions  to  handle  plotting  logic  (data  preparation,  axis,
        legends, etc.)
|   └— pso.py
        #  Particle  Swarm  Optimization  implementation  -  used  by  both  basic
        and modified approaches
|— gui_files/
        # GUI - files used to define the GUI properties, handle translations
        and visualisation
|   ├— TBP_visualisation.py
        # Python file with GUI generated by QtDesigner
|   ├— TBP_visualisation.ui
        # .ui file with GUI generated by QtDesigner
|   ├— en_translation.qm
        # Translations - English version definition (raw)
|   ├— en_translation.es
        # Translations - TypeScript - English version definition
|   ├— gui.py
        --> #  [Run me!] Main GUI file, should be run to start the application
|   ├— user_inputs.py
        # GUI file - input elements definitions
|   └— visualisation.py
        # GUI file - visualisation elements - plots and tables
|— tests/
        # Unit tests
|   └— test_algorithms.py
        # pytest file with unit tests for algorithms
|— docs/
        # project documentation
|   └— assets/
        # images and UML code used in documentation
|— orbits/
        # Text files with raw NASA orbits - processed by sources/data_load.py
|— LICENCES/
        # Licences for external sources used in this project
|— Combinear.qss
        # .qss template with pre-prepared GUI style
└— requirements.txt
        # .txt file with libraries required to run this project locally
```

2.2 Main components structure

Mathematical model is the essential part of the project. Moon-Earth system's definition and original orbit's properties need to be accessible for all algorithms. This is why *Swarm* and *PropagatedElement* classes uses the model (*ModelProperties*) as their attribute. Both *Swarm* and *PropagatedElement* classes are extended by the specific algorithms implementations. This structure guarantees scalability.

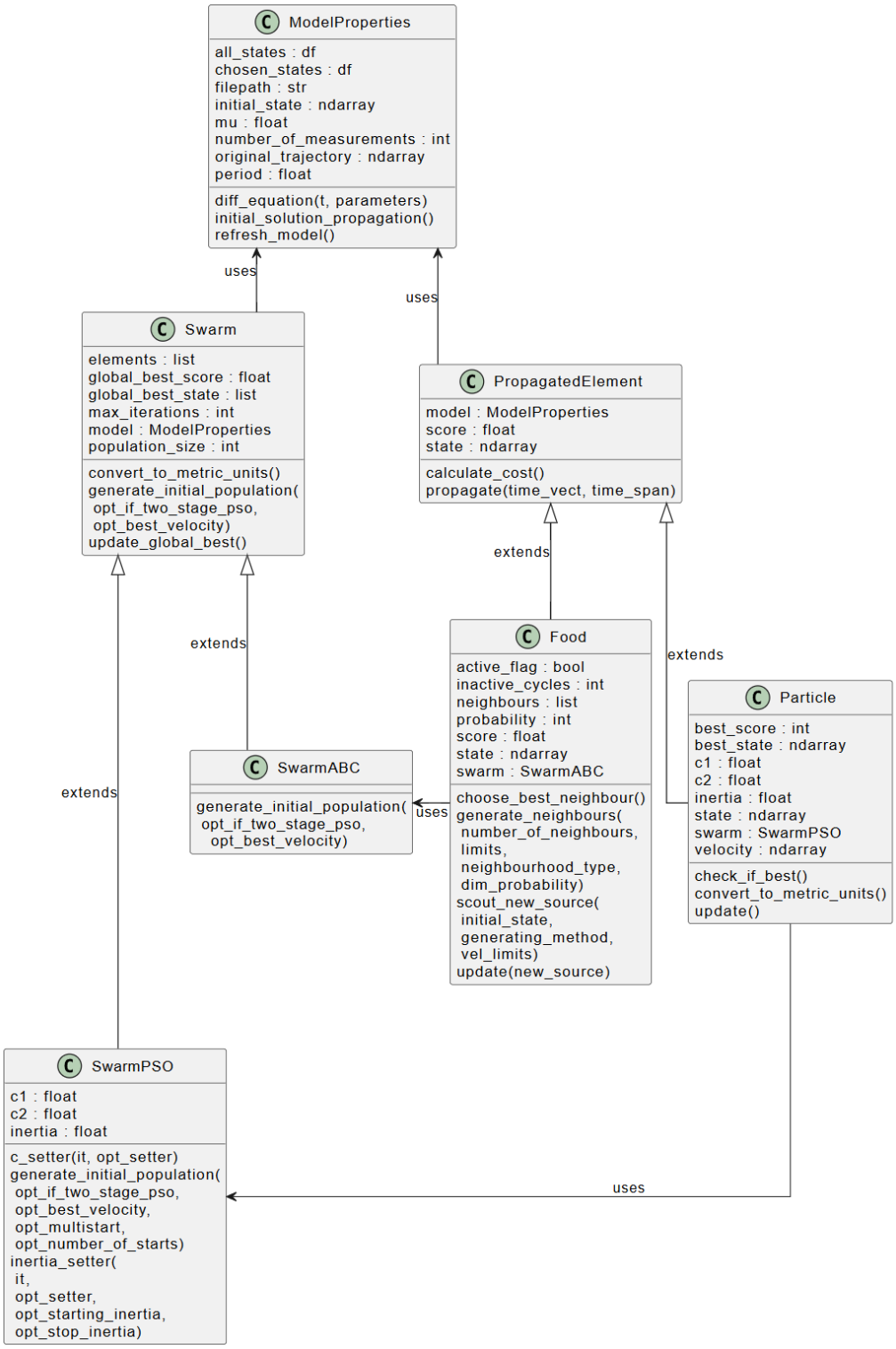

Figure 1 - UML diagram for algorithms logic

2.3 Data structures

Algorithms inputs and outputs are handled using data structures that help organise interfacing information between source files and PyQt6 GUI. They are defined in *sources/data_structures.py.*

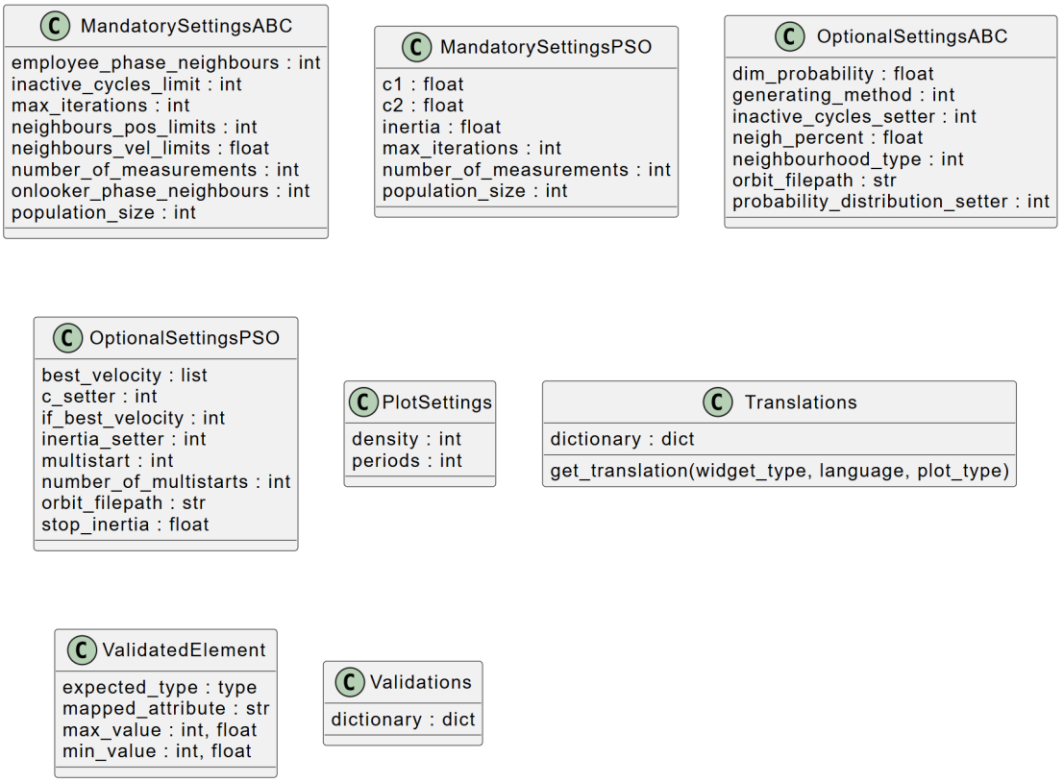

Figure 2 - UML for data structures

The mandatory and optional inputs are handled in *[Mandatory/Optional]Settings[PSO/ABC]* classes. Default values are defined there as well.

```
class MandatorySettingsPSO:
    def __init__(self,
                 max_iterations=10,
                 population_size=10,
                 number_of_measurements=35,
                 inertia=0.9,
                 c1=1.49,
                 c2=1.49,
                 ):
        self.max_iterations = max_iterations
        self.population_size = population_size
        self.number_of_measurements = number_of_measurements
        self.inertia = inertia
        self.c1 = c1
        self.c2 = c2
```

The universal convention to pass the parameters between algorithms and GUI is to use mandatory or optional objects initialised with requested values.

```
  def pso(mandatory, optional=None)
```

Validations are done using a *ValidatedElement* class. All GUI elements with boundaries have their own associated object of this class.

```
class ValidatedElement:
  def __init__(self,
               expected_type,
               min_value,
               max_value,
               mapped_attribute):
      self.expected_type = expected_type
      self.min_value = min_value
      self.max_value = max_value
      self.mapped_attribute = mapped_attribute
```

You can check the limits associated with validations later in this documentation (see section *3. Parameters*) or in *Validation* structure.

2.4 Loading .txt orbit data

All original orbits from NASA database should be saved into *sources/* directory in .txt format. They are transferred into pandas data frames with *sources/data_load.py*. Based on the orbit file path and number of measurements, the raw data is processed to return only the required number of the original orbit points that should be equally spread in time. The function returns the new, sparser set of measurements, original measurements, initial state and original orbit period.

2.5 Plots

All plots definitions can be found in *sources/plot_functions.py*. The inputs are vectors of data returned by algorithms. Plots are generated with *matplotlib.pyplot* library. The functions are called by GUI to visualise the results, but they can also be used as independent functions while implementing and testing new algorithms.

2.6 GUI

App's GUI was created with PyQt6 and QtDesigner. The class structure is based on main *App* class, inheriting from *UiMainWindow* (class automatically generated while created GUI elements in QtDesigner). It depends on *UserInputs* class to receive all parameters and *Visualisation* to present the results. The simplified UML diagram shows the dependencies (see Figure 3).

GUI is available in two language versions – Polish and English. User can switch between them at any time. New languages can be added to the app. To make this possible, QTranslator needs to be imported (as it currently is). To generate a .ts file with all GUI labels to be translated, run pylupdate6 gui.py -ts [file_name].ts in the terminal. Edit the generated .ts file by adding your target language labels. Finally, run lrelease [file_name].ts -qm [file_name].qm to create a binary file that can be processed by PyQt6. To make sure changes are visible in the app, review *combobox_language_selected()* method of gui_files/*gui.py/App* and *sources/data_structures.py/Translations*.

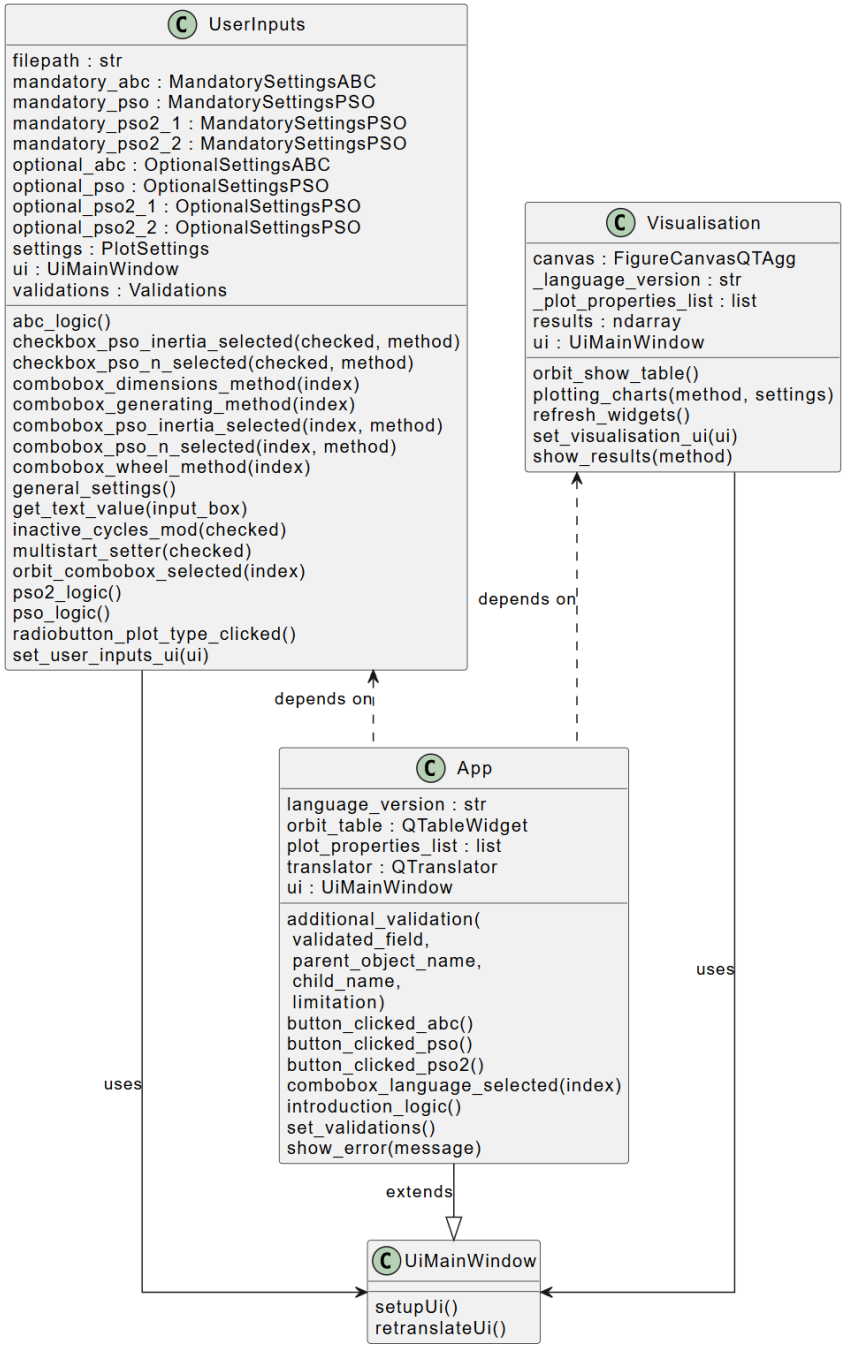

Figure 3 - UML diagram for GUI implementation

2.7 Tests

Unit tests can be found in *tests/* directory. For now, only general unit tests for algorithms have been implemented. To run them, make sure you use *pytest*.

3. Parameters

3.1 Overview

The table below describes all user inputs that impact the algorithms.

Parameter : input label from the interface.
Algorithm : informs if given algorithm uses the parameter.
Type : M – mandatory input, O – optional input (will not be used if not selected).
Range : expected type and value.
Description : parameter application.

Table 1 : Parameter overview

#	Parameter	Algorithm	Type	Range	Description
1	Max iterations	PSO, PSO2, ABC	M	Int : [1, 1000]	Upper limit of iterations.
2	Population size	PSO, PSO2, ABC	M	Int : [1, 1000]	Swarm population size limit. For ABC, it refers to number of employee bees.
3	Measurements per orbit	PSO, PSO2, ABC	M	Int : [1, 250]	Number of measurement points where orbits are compared.
4	Inertia	PSO, PSO2	M	Float : [0, 2]	PSO control parameter. Bigger values promote space exploration, while smaller – solution exploitation.
5	Cognitive coeff.	PSO, PSO2	M	Float : [0, 4]	PSO control parameter. Bigger values promote particle's best position impact on its movement.
6	Social coeff.	PSO, PSO2	M	Float : [0, 4]	PSO control parameter. Bigger values promote global best position impact on each particle's movement.
7	Inertia$_{[mod]}$	PSO, PSO2	O	Dropdown list #1	Modification : inertia changes dynamically based on current iteration number.
8	Stop inertia$_{[mod]}$	PSO, PSO2	O	Float : [0, 2]	Modification : inertia changes in a linear manner between the start value (#4) and stop value.
9	Coefficients$_{[mod]}$	PSO, PSO2	O	Dropdown list #2	Modification : cognitive and social coeffs change dynamically based on current iteration number.

10	Multistart[mod]	PSO2	O	Int : [1, 25]	Modification : number of early starts, where 1/(number of multistarts) best particles are taken to form a final swarm.
11	Neigh. bees (stage 1/2)	ABC	M	Int : [1, 20]	Number of neighbours of each source visited in employee/onlooker phases respectively
12	Neighbourhood limits – pos	ABC	M	Float : [0, 250]	Limit [km] a new neighbour cannot exceed with respect to the current source.
13	Neighbourhood limits – vel	ABC	M	Float : [0, 0.1]	Limit [km/s] a new neighbour cannot exceed with respect to the current source.
14	Inactive cycles	ABC	M	Int : [1, 100]	Acceptable number of iterations with no changes in source state. After exceeding, the source will be dropped.
15	New solutions generation[mod]	ABC	O	Dropdown list #3	Modification : a way new solutions (scout bees) are generated – it can be random or with respect to the global best source at a given moment.
16	% of vel. [mod]	ABC	O	Float : [0, 1]	Modification : related to (#15) – if new solution is generated with respect to the global best score, this parameters defines how much the new velocity can differ from the current best.
17	Number of mod. neigh. dims. [mod]	ABC	O	Dropdown list #4	Modification : defines which dimension should be modified when generating neighbours – either all or just some of them.
18	Prob. [mod]	ABC	O	Float : [0, 1]	Modification : related to (#17) – defines the probability a dimension will be changed when generating a neighbour.
18	Roulette wheel selection[mod]	ABC	O	Dropdown list #5	Modification : defines the way roulette wheel works when assigning onlooker bees to sources. It can be more or less proportional to the source score.

3.2 Dropdown lists

Dropdown lists are explained in the table below.

Table 2 : dropdown lists - parameters

#	Parameter	Option	Description
1	Inertia	Linear	Inertia changes in a linear manner between start and stop values.
		Dynamic – variant 1	Inertia changes in a waterfall manner : $\omega = \begin{cases} 0.95, when\ iter.in\ [1,20) \\ 0.9, when\ iter.in\ [20,50) \\ 0.85, when\ iter.in\ [50,100) \\ 0.8, when\ iter.in\ [100,150) \\ 0.75, when\ iter.in\ [150,1000) \end{cases}$.
		Dynamic – variant 2	Inertia changes in a waterfall manner : $\omega = \begin{cases} 0.95, when\ iter.in\ [1,20) \\ 0.9, when\ iter.in\ [10,50) \\ 0.825, when\ iter.in\ [50,100) \\ 0.75, when\ iter.in\ [100,150) \\ 0.65, when\ iter.in\ [150,1000) \end{cases}$.
2	Coefficients	Dynamic – variant 1	Cognitive and social coefficients change in a waterfall manner : $c1, c2 = \begin{cases} [1;\ 1], when\ iter.in\ [1,20) \\ [1.2; 0.8], when\ iter.in\ [10,50) \\ [1.4; 0.6], when\ iter.in\ [50,100) \\ [1.6; 0.4], when\ iter.in\ [100,150) \\ [1.75; 0.25], when\ iter.in\ [150,1000) \end{cases}$.
		Dynamic – variant 2	Cognitive and social coefficients change in a waterfall manner : $c1, c2 = \begin{cases} [1;\ 1], when\ iter.in\ [1,20) \\ [0.8;\ 1.2], when\ iter.in\ [10,50) \\ [0.6; 1.4], when\ iter.in\ [50,100) \\ [0.4;\ 1.6], when\ iter.in\ [100,150) \\ [0.25;\ 1.75], when\ iter.in\ [150,1000) \end{cases}$.
3	New solutions generation	Default	Solution generated randomly within 250km and 0.5 km/s with regards to initial state (both positions and velocities).
		Close to global best solution	Solution generated randomly within 125km and global best velocity +- (#16)% with regards to global best state.
		Velocity of global best solution	Solution generated randomly within 250km and global best velocity +- (#16)% with regards to initial state (positions).
4	Number of mod. neigh. dims.	All	All dimensions are modified when looking for neighbouring food sources.
		Random	Dimensions are randomly modified (within limits) with probability (#18)
5	Roulette wheel selection	Linear	The best and worst global scores impact on the probability of choosing each source. Closer to best score, better the result, with linear drop toward the worst one.
		Inversely proportional	The bigger 1/(source score) value, the bigger probability is assigned.

3.3 Other modifications

The parameters below were not classified into any of previous tables.

Table 3 : Other parameters

#	Parameter	Type	Description
1	Inactive cycles	Checkbox	If active, inactive cycles limit changes dynamically in a waterfall manner : $n = \begin{cases} 1, when\ iter.in\ [1, 50) \\ 3, when\ iter.in\ [50, 100) \\ 5, when\ iter.in\ [100, 150) \\ 7, when\ iter.in\ [150, 1000) \end{cases}$.
2	Trajectory generation	Radio-button	Generates a plot of densely spaced points or just the measurement points.
3	Number of periods	Input	Propagates the found solution with n * (original period) time vector.

4. Requirements

Table 4 : Requirements overview

Library/Tool	Version	Notes
Python	3.11+	Algorithms and GUI were created in Python
PyQt6	6.7.1	GUI
matplotlib	3.9.2	Results visualisation
numpy	1.26.4	Calculations
scipy	1.13.1	Differential equations
pandas	2.2.2	Loading and converting .csv data
pytest	7.4.4	Unit tests
pytest-cov	6.0.0	Unit tests - coverage
pytest-mock	3.14.0	Unit tests - mocking
pytest-qt	4.2.0	Unit tests - PyQt elements

5. Usage

To run the project locally, all the required libraries should be installed. The file structure should not be modified without further verification of dependencies, as many imports are used in each directory. You can start GUI by running `gui.py`.

When app starts, you will find 5 tabs you can interact with:

- **Introduction:** instructions and tips, original orbit selection,
- **Settings:** visualisation settings (to be extended),
- **PSO algorithm:** parameters selection and results visualisation,
- **Two-stage PSO algorithm:** parameters selection and results visualisation,
- **ABC algorithm:** parameters selection and results visualisation.

All algorithmic tabs share the same layout patterns (see Figure 4).

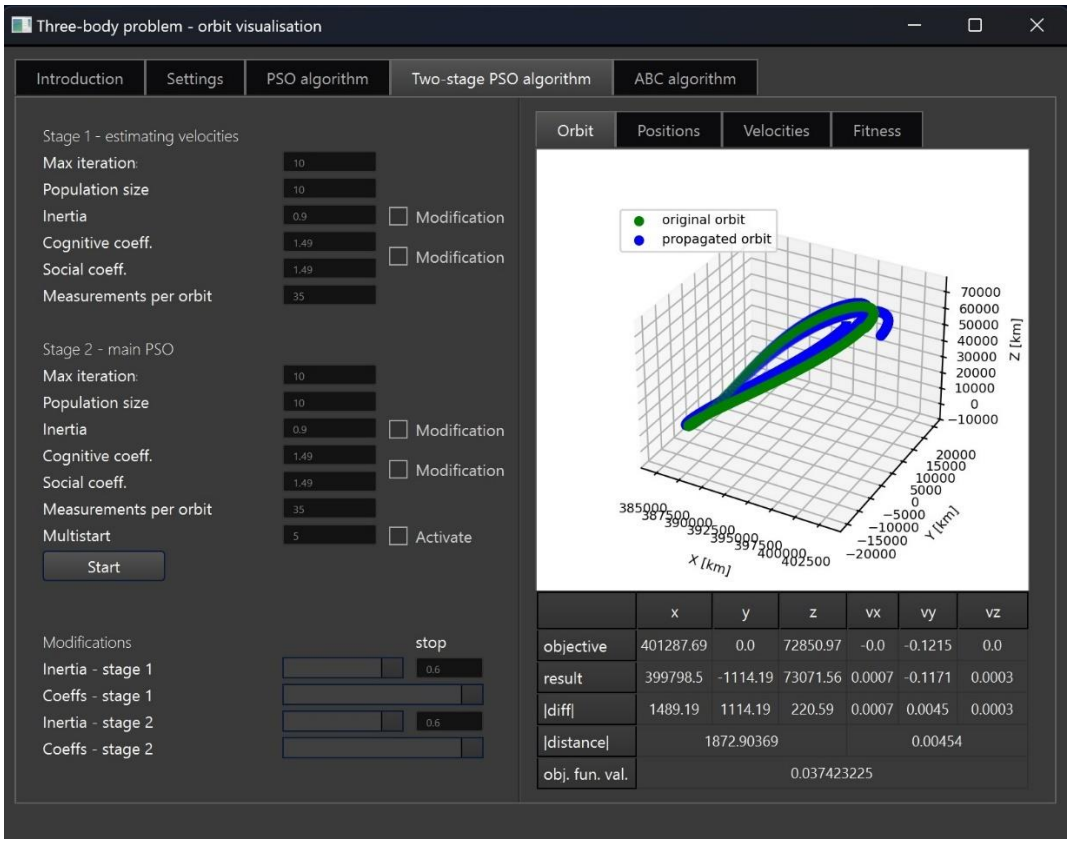

Figure 4 - Two-stage PSO - GUI

You can set your preferred parameters in the control panel on the left. If your choice exceeds the limits, you will notice an error message, and the field will turn red for a few seconds.

Figure 5 - Error message

The checkboxes neighbouring the parameters are used to activate the modifications (bottom-left) - additional parameters extending the basic functionalities. You will find more details in section *3. Parameters.*

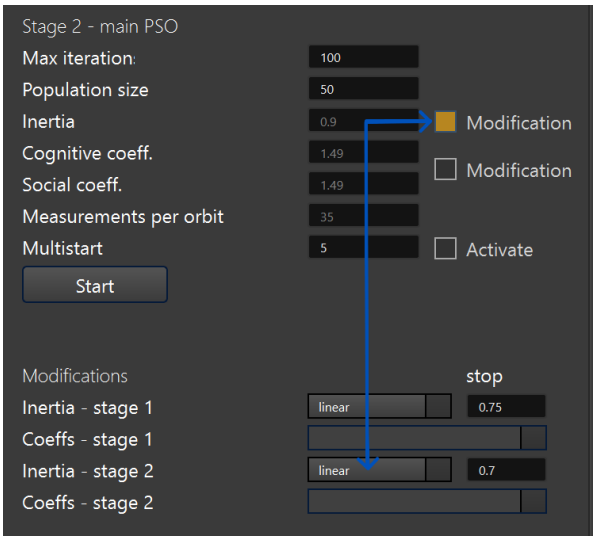

Figure 6 - Modification checkbox

You can run the algorithm by clicking on 'Start'. Depending on your parameters, it can take a few moments to generate the results. The set of plots includes:

- **Orbit:** comparison of original and propagated orbit,
- **Positions:** initial and final swarms limited to the first three dimensions,
- **Velocities:** initial and final swarms limited to the last three dimensions,
- **Fitness:** objective function values in each iteration.

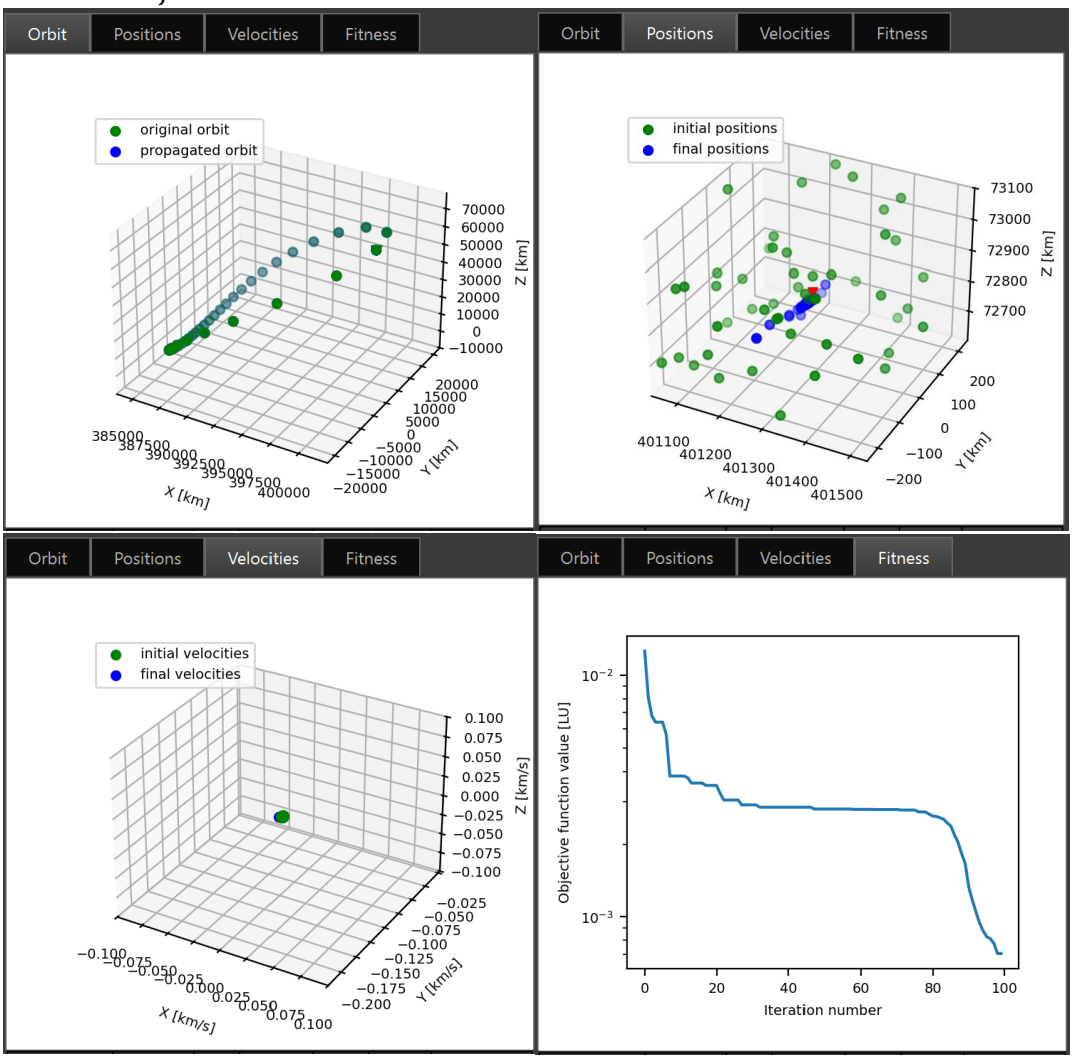

Figure 7 - Four plots for Two-stage PSO

You can compare the orbits using the score table. It contains the initial state of original and propagated orbit, the difference between them and final objective function value.

	x	y	z	vx	vy	vz
objective	401287.69	0.0	72850.97	-0.0	-0.1215	0.0
result	401273.88	3.02	72808.74	0.0001	-0.1215	0.0002
|diff|	13.81	3.02	42.23	0.0001	0.0	0.0002
|distance|	44.53456			0.00026		
obj. fun. val.	0.0007047341					

Figure 8 - Result table