Designing a model of the solar system and simulating a mission to Titan

Group 12

Mikolaj Gawrys
Weronika Gorak
Leo Ebeling
Martin Popov
Yuri Manna
Alejandro Ibarrola
Emre Pelzer

Department of Advanced Computing Sciences
Maastricht University
Netherlands
26th of June 2023

Contents

Abstract

This report presents a Java program designed to create a simulation of a space mission to Titan, Saturn's largest moon. The program incorporates a graphical user interface (GUI) for interaction and visualization of the solar system, as well as a physics model that accurately computes gravitational forces and simulates the trajectories of celestial bodies and the rocket. The research questions addressed in this study focus on accurately modeling the solar system, choosing most efficient differential equation solvers, and enhancing the simulation with additional features. The implemented Java program is an accurate representation of the solar system and it enables a successful space mission. Different differential equation solvers, including the Euler solver, classic Runge-Kutta 4th order solver, Adams-Bashforth 2nd order solver, and the predictor-corrector method are evaluated, and the trade-offs between accuracy and computational complexity are highlighted. The program also includes rocket launching and flight control functionalities, incorporating optimization algorithms such as the hill-climbing algorithm to calculate the velocities required for a successful mission and to minimize the fuel consumption of the engine. Experiments are conducted to demonstrate the influence of step size on the precision of the solvers and to provide insights into the complex structure and interdependent forces within the solar system. The findings contribute to the understanding of space travel and the impact of mathematical models and solutions in simulations.

1 Introduction

The solar system, being an incredibly complex structure, poses quite a challenge to accurately predict, model and implement. Thus, understanding and exposing the underlying mathematical equations and models embedded within is vital. We want to successfully simulate a space mission to Titan, a moon of Saturn - enter its orbit, land on the surface, then come back into Earth's orbit in a time span of two years. This implies robust mathematical models that are able to reliably simulate the solar system and the motions of celestial bodies and the physical interactions between them as well as ways to compute the correct trajectories and velocities for the space shuttle (Hao-ran & Wei-dong, 2023). Those mathematical models and formulas exist already, are well understood and used widely in tackling space travel. Orbital mechanics and Lambert's problem more specifically have important applications

in areas of rendezvous, targeting, guidance and preliminary orbit determination (Blanchard & Lancaster, 1968). Newton's laws of motion and the law of universal gravity are used to define the relationships between celestial bodies (Heggie, 2006). Differential equation solvers, such as Runge–Kutta methods, enable us to numerically approximate the state of the solar system through time (Faires & Burden, 2012).Designing a model of the solar system and simulating a space mission to Titan raises several research questions:

1. How can a Java program accurately represent the solar system, including gravitational forces acting on celestial bodies and the spacecraft?

2. What implications arise from choosing one differential equation solver over another, and how can these implications be considered when simulating the mission?

3. How can the simulation be enhanced with additional features such as rocket stages and fuel consumption?

4. What approaches can be used to design a user-friendly GUI for interacting with and visualising the solar system and the space mission?

Understanding those implications can help space travel in general, but more specifically, it can aid in measuring the impact of using arbitrary mathematical models and solutions over others in performing those simulations.

We base our research off existing methods to approximate numerical solutions to differential equations already mentioned. Additionally, we build on optimization algorithms such as hill climbing and gradient descent to find the initial conditions needed to launch the space shuttle (Johnson & Jacobson, 2002). The main idea behind those approaches is as follows - we first want to find those initial conditions for the shuttle to reach Titan using the optimization algorithms, which would take into account all the complex forces involved, then accurately simulate the behaviour of the entire system by approximating the solutions to the differential equations and forces that govern the system. Finally, we repeat this process and safely return to Earth.

Detailed explanations of the differential equation solvers as well as the description of the physics model can be found in section 3. We touch on optimization algorithms in subsection 3.4. The findings are presented in the Results section. We then analyse and discuss those results in section 5. Closing remarks and what follows from our research can be found in the

Conclusion section. The UML diagram and screenshots of the user interface are in the Appendix.

2 Physics model

2.1 Solar System

The solar system is represented as a three dimensional coordinate system with the sun in the centre. Every object in the space has a 3d vector of its position, a 3d vector of its velocity and a double that represents its mass in kilograms. The position coordinates are represented in kilometres and the velocities in kilometres per second (Murray & Dermott, 2000).

To calculate the gravitational impact of all the objects in the system on each other, the method "gravitationalForce(...)" is called for each object during the solving process. It takes the current position and mass of the object it is calculating the force for and a list of all the objects in the system as input parameters. The gravitational force that pulls on object o is calculated using Newton's law of universal gravity.

$$F_i^G = -\sum G m_i m_j \frac{x_i - x_j}{||x_i - x_j||^3}$$

The formula calculates the force of each object individually and sums them up to return a 3d vector of force that is pulling on the object. G is the gravitational constant, m_i and m_j are the masses of the object and the current object in the equation and x_i and x_j are the positions of the object o and the current object in the equation. The result of these calculations are used to calculate the velocity of a given object at the next step in time during the solving process of the differential equations, which is described in more detail in section 2.4.

2.2 Titan Landing

For the landing simulation the model of the physical forces acting in the system is much simpler. In the simulation exists only one object, the landing module, for which forces are calculated. The module moves in a two dimensional space and has, additionally to the two coordinates x and y also another parameter θ, that represents the current rotational angle. The acceleration

in change of these parameters is modeled by the following equation.

$$\ddot{x} = u\,sin(\theta),$$
$$\ddot{y} = u\,cos(\theta) - g,$$
$$\ddot{\theta} = v,$$

where u is the thrust of the main engine of the module, v the thrust of side-boosters to turn the module, and g the gravitational force of Titan pulling on the module ($g \approx 1.352 * 10^{-3}\, km/s^2$).

2.3 Wind and Atmosphere

Since the wind and atmosphere plays an important role in the landing process it could not have been overlooked. To ensure a realistic and cohesive approach, we took into account all the forces at play during the descent into dense atmosphere. Both temperature and pressure play a big role in evaluating the air resistance. To get an idea of the structure of Titan's atmosphere we looked at data from NASA (https://attic.gsfc.nasa.gov/huygensgcms/Titan_atmos.gif). We then used polynomial interpolation to evaluate both pressure and temperature functions based on a given altitude. The former being:

$$\begin{aligned} f(x) = &-0.000000000079339x^5 \\ &+0.000000049373809x^4 \\ &-0.000011714823097x^3 \\ &+0.001320377017440x^2 \\ &-0.071004201806174x \\ &+1.494389829591593 \end{aligned}$$

And the latter:

$$\begin{aligned} f(x) = &-0.000000000282784x^6 \\ &+0.000000160592161x^5 \\ &-0.000033550883549x^4 \\ &+0.003020265171911x^3 \\ &-0.095206120316128x^2 \\ &+0.219198689819765x \\ &+88.631154566255688 \end{aligned}$$

Those functions enabled us to reasonably predict the air resistance throughout the landing process.

The air density ρ was calculated with the following formula:

$$\rho = \frac{pm}{rt} * 1.5,$$

where p is the pressure at a given altitude represented in pascals, m is the molar mass of air on Earth, r is the ideal gas constant equal to 8.314 J/(mol·K) and t is the temperature at a given altitude represented in Kelvins. Because the atmosphere of Titan is 50% denser than Earth's atmosphere, and we used the molar mass of air on Earth, we simply multiply it by 1.5 to obtain the air density on Titan.

The air resistance (drag) was calculated using the following formula:

$$D = C_d \frac{rV^2}{2} A,$$

where C_d is the drag coefficient, which equals 0.47 for a sphere, r is the ideal gas constant equal to 8.314 J/(mol·K), V is the velocity of the object, and A is the area of the object facing the air.

To calculate the wind velocity vector we calculate two values: the wind speed and the wind angle. They both use the stochastic function:

$$f(step, shift) = c_1 * cos(4.667 * step/3600) + c_2 * cos(12.22 * step/3600) + shift,$$

where the c coefficients are randomized (within boundaries), $step$ is the time at which we want to calculate the velocity, and $shift$ varies depending on the wind type.

The speed of the wind (first entry in the wind velocity vector) of the module is the value of the function multiplied by the direction of the wind (it can be either positive or negative) and divided by a strength factor of a specified wind (for example hurricane, light wind, etc.), the higher it is the weaker the wind.

Similarly, the angle of the wind is calculated using simple trigonometry ($arcsin$). This allows us to obtain the angle immediately from the length of the vector and its y-coordinate entry. The last entry of the velocity vector is zero, as we are working in a two-dimensional coordinate system for the landing module.

After calculating the wind velocity vector, we subtract the velocity of the module from the wind velocity, and then we calculate the air resistance using this difference. Finally, we apply the force (air resistance) to the module, making the wind blow.

2.4 Differential Equation

In order to approximate the positions of all celestial objects during the simulation, numerical solvers for differential equations had to be implemented. Said solvers are used to approximate the solutions of ODE's (Ordinary Differential Equations). The differential equation used in the program to calculate the derivative of the state of a celestial object at a step t is given by

$$\dot{y}_i(t) = \frac{d}{dt}\begin{pmatrix} x_i(t) \\ y_i(t) \end{pmatrix} = \begin{pmatrix} v_i(t) \\ \frac{F_i}{m_i} \end{pmatrix},$$

where t is the current step, $x(t)$ and $v(t)$ are the current position and velocity of the celestial object at step t, m is the mass of the celestial object, and F is the gravitational force that pulls on the celestial object. The calculation of F is explained in more detail in section 3.1. For our calculations we used a state 2D vector for each celestial object, where the first entry is a 3x1 vector with the x,y and z position values at a given step t, and the second entry is a 3x1 vector with the velocities of x,y and z at a given step t.

3 ODE solvers

3.1 Euler Solver

The Euler solver is a first-order numerical method for solving ordinary differential equations (ODE's). It uses the first derivative of a function times the step-size at a specific step t to approximate the value of the function at the $t + h$ where h is the step size. This formula is given by

$$w_{i+1} = w_i + h_i f(t_i, w_i),$$

where t_i is the step taken during the i-th iteration, $f(t_i, w_i)$ is the value of the derivative at the specified step t_i and w_i is the state of an object during the i-th iteration. It is a very easy to implement solver, which does not require much computations, however, compared to other methods it is not very accurate.

3.2 Classical Runge–Kutta 4th order solver

Another solver used was the classical Runge-Kutta 4-stage method. It is an iterative method and it uses four different estimates of the slope at each

iteration to get the value of the function. Those estimates are given by

$$\begin{aligned} k_{i,1} &= h_i f(t_i, w_i), \\ k_{i,2} &= h_i f(t_i + \tfrac{1}{2}h_i, w_i + \tfrac{1}{2}k_{i,1}), \\ k_{i,3} &= h_i f(t_i + \tfrac{1}{2}h_i, w_i + \tfrac{1}{2}k_{i,2}), \\ k_{i,4} &= h_i f(t_i + h_i, w_i + k_{i,3}), \\ w_{i+1} &= w_i + \tfrac{1}{6}(k_{i,1} + 2k_{i,2} + 2k_{i,3} + k_{i,4}), \end{aligned}$$

where t_i is the step during the i-th iteration, $f(t_i, w_i)$ is the value of the derivative at the specified step t_i, w_i is the state of an object during the i-th iteration and $k_{i,n}$ represent the slopes of the solution for a given step. This method is slightly more complex to implement than the Euler method, however it is much more accurate.

3.3 Adams–Bashforth 2nd order solver

This method is a numerical method which requires two initial values to predict the next value of the function. Because there is only one initial state given, it is necessary to bootstrap the second initial value. It is a common practice to use a Runge–Kutta method of the same or higher order. Since the 4th order Runge–Kutta method has already been implemented, it is used to approximate the second initial value. The model is given by

$$w_{i+1} = w_i + \tfrac{1}{2}h(3f(t_i, w_i) - f(t_{i-1}, w_{i-1})),$$

where t_i is the step during the i-th iteration, $f(t_i, w_i)$ is the value of the derivative at the specified step t_i, w_i is the state of an object during the i-th iteration. This method does not require much computations, therefore is fast, but it is not more accurate than the Runge–Kutta method. However, it does have a higher accuracy than the Euler solver.

3.4 Predictor–corrector method

This method is also known under the name of Adams–Bashforth–Moulton two stage method. It differs from the other methods, because it has two components: predictor and corrector. The predictor step simply calculates the estimated value of the function at the next step using the Adams–Bashforth 2nd order solver. Afterwards, the corrector step improves that approximation using an implicit method and makes the approximation more accurate

by using the derivatives of the current and previous values. The implemented predictor-corrector uses the Adams-Moulton implicit method as a corrector (Collatz, 1960). This method is given by

$$w_{i+1} = w_i + (h/12)(5f(t_{i+1}, w_{i+1}) + 8f(t_i, w_i) - f(t_{i-1}, w_{i-1})),$$

where t_i is the step during the i-th iteration, $f(t_i, w_i)$ is the value of the derivative at the specified step t_i, w_i is the state of an object during the i-th iteration. It is important to note that the $f(t_{i+1}, w_{i+1})$ is the predictor step, which uses the Adams–Bashforth 2nd order to get the approximation.

4 Launch and flight control

In the simulation, the rocket is treated as any other celestial object in the system. It is controlled by the engine and calculates the resulting velocity of the rocket and the consumed fuel during execution. The velocity is calculated by

$$v(t + \delta t) = v(t) + \frac{I}{m},$$

where $v(t)$ is the original velocity of the rocket and the added velocity is an impulse, divided by the mass of the rocket. The impulse is calculated by taking a vector of force in Newton and multiplying it with the current step size.

The implementation of the rocket allows us to alter its position and velocity during runtime. In our case it initially fires the engine of the rocket twice to leave earth and reach Titan. After a year and if the distance to Titan is low enough to enter a orbit, the controls class is firing the rocket engine again to slow it down and then to enter the orbit of titan at the orbital speed, calculated by the formula

$$v_0 \approx \sqrt{\frac{GM}{r}},$$

where G is the gravitational constant, M is the mass of the object in the centre of the orbit and r the radius of the orbit (Lissauer & Pater, 2019).

To return to Earth the same process is applied as well as for entering the orbit, although for slowing down when approaching Earth, three engine fires

are used because the rocket has a very high velocity, hence a third engine fire gives the program more control and time to slow it down and calculate the correct orbital speed.

The vectors to reach Titan and return back to Earth were calculated using the hill-climbing algorithm explained in section 6.1. All the other calculations are happening during the runtime of the simulation.

5 Landing Control

In order to land successfully on Titan at given coordinates, the landing controls are modifying the thrust of both the main and the side engines. The target coordinates are $x = 0, y = 0$ and the target angle is $\theta = 0$. The initial position of the landing module is the orbital height as x and the initial velocity is $\dot{x} = 0, \dot{y} = \mathit{orbital\ speed}$.

The algorithm to land the module is divided into two main stages. First it aligns the vector of velocity of the module with the vector between the module and the target. During this phase the path of the module is corrected at each step in order to reach an angle between the module and target that is as close to zero as possible. It achieves this by dividing the route into multiple parts and targeting always a new point in space that reduces the angle between the module and the final target.

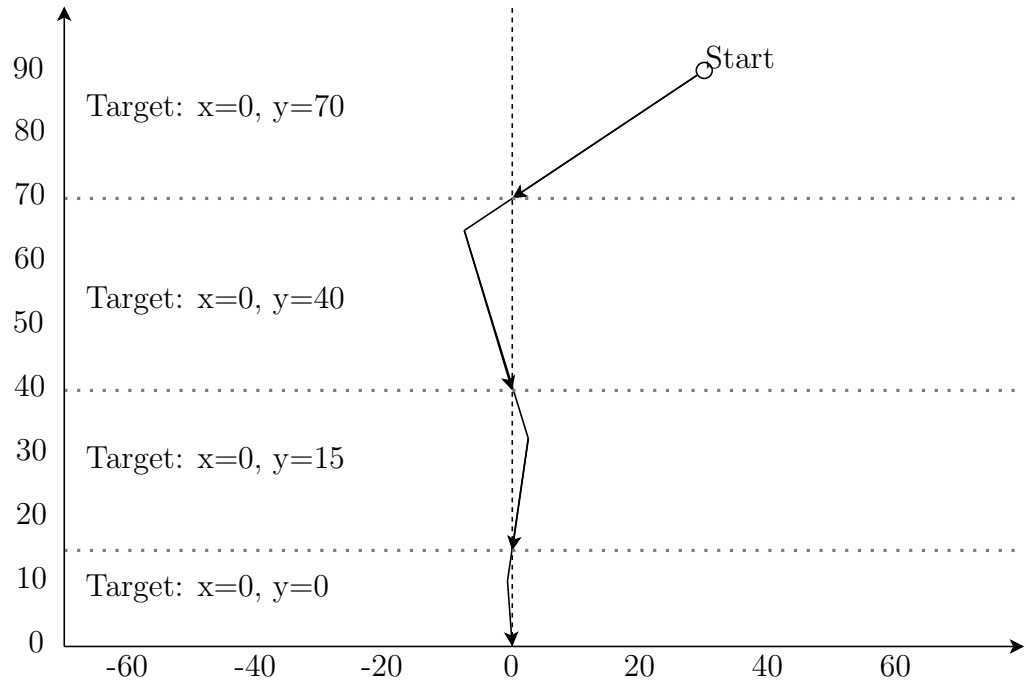

Figure 1: A representation of the path of the landing module. It overshoots the current target and gets closer at every stage

Figure 1 represents this conceptual idea. The actual targets used in the application are adjusted to the real environment of the module. As shown in the figure, the landing module targets a different target on the line at x=0 each time it reaches the next zone. Because of the high velocity, the module overshoots the target each time, but the angle between the module and the new target gets smaller and smaller.

When the module reaches a position very close to the ground, the second phase of the landing process begins, where the main objective is, to slow the module down to a velocity that ensures a safe landing. Because this will require the main engine to fire with a lot of force, it is crucial for the module to be as perfectly aligned with the target as possible. Otherwise the module will not precisely land on the final target, since the engine will accelerate it in the x direction if the module has a large rotational angle.

6 Optimization

6.1 Hill Climbing

We developed a hill-climbing algorithm to calculate the velocities for the engine of the rocket to reach Titan and go back to Earth. In both cases the engine fires two times to reach its travelling velocity. Since the engine takes a vector with x, y and z values as a parameter, this vector is the input criteria the algorithm optimises, while the total distance to the target object after a certain time span is the error the algorithm minimises.

The algorithm starts with a vector with initial values. These initial values are not important for the outcome of the hill-climbing algorithm, but if they are already somewhat in the right direction, it will speed up the process a lot. We approximate this value by running the simulation once and subtracting the positions vector of Titan and Earth, then divide it by the number of seconds in a year which yields a somewhat close estimation of the initial velocity. In each iteration of the climbing process, the algorithm takes steps in each direction, x, y, z, $-x$, $-y$, and $-z$. If the error is lower than the previous best error after one of these steps, the input vector of that step is the new vector the algorithm takes as a basis to do its steps. If the steps the algorithm does during this iteration are not decreasing the error anymore, the step size is halved and the process starts over again.

After a predefined number of iterations, the algorithm terminates and outputs the vector that performed best during the process. In our case the algorithm calculated a vector to reach Titan after less than 30 iterations. Since it starts with a step size of 10, the resulting precision of the vector is $10/2^{29}$.

7 Experiments and Discussion

7.1 Measuring the performance of the solvers

The solvers were tested with the initial value problem $y' = t - y^2$, with initial $y(0) = 1$ and final $y(1.5) = 1.028601835296908$ as experiments. The program ran with 20 different step sizes (h): $h = 1.5/j^2$, where j is the iteration, starting from 0. The results given by all the four solvers were then used to calculate the absolute error from the exact value $y(1.5)$ to derive their accuracy. This experiment provided an opportunity to assert the differences

between the different methods and their behaviour with increasing step sizes, other than confirming our expectations derived from the mathematical formulas of the errors. The results of the experiments on the ODE solvers with increasing step size are shown the figure below.

Table 1: Solver errors when approximating $y' = t - y^2$ with the exact solution $y(1.5) = 1.0286(4dp.)$

Step-size	**Euler's method**	**Predictor–corrector**	**Adams-bashforth**	**Runge–Kutta**
1.0	1.5286	0.7694	0.7694	0.7694
0.5	0.2629	0.2893	0.7527	7.57E-3
0.25	0.0647	0.0391	0.0967	2.78E-4
0.125	0.0289	9.04E-3	0.0172	1.21E-5
0.0625	0.0139	2.29E-3	3.77E-3	6.26E-7
0.03125	6.81E-3	5.73E-4	8.90E-4	3.55E-8
1.56E-2	3.37E-3	1.43E-4	2.17E-4	2.12E-9
7.81E-3	1.69E-3	3.57E-5	5.34E-5	1.29E-10
3.91E-3	8.39E-4	8.93E-6	1.33E-5	8.00E-12
1.95E-3	4.19E-4	2.23E-6	3.31E-6	5.20E-13
9.77E-4	2.09E-4	5.58E-7	8.26E-7	5.6E-14
4.88E-4	1.05E-4	1.40E-7	2.06E-7	2.6E-14
2.44E-4	5.23E-5	3.49E-8	5.16E-8	2.6E-14
1.22E-4	2.62E-5	8.72E-9	1.29E-8	2.3E-14
6.10E-5	1.31E-5	2.18E-9	3.22E-9	2.2E-14
3.05E-5	6.54E-6	5.45E-10	8.05E-10	2.0E-14
1.53E-5	3.27E-6	1.36E-10	2.01E-10	2.9E-14
7.63E-6	1.63E-6	3.41E-11	5.03E-11	2.6E-14
3.81E-6	8.17E-7	8.51E-12	1.26E-11	3.5E-14
1.91E-6	4.09E-7	2.15E-12	3.12E-12	2.2E-14

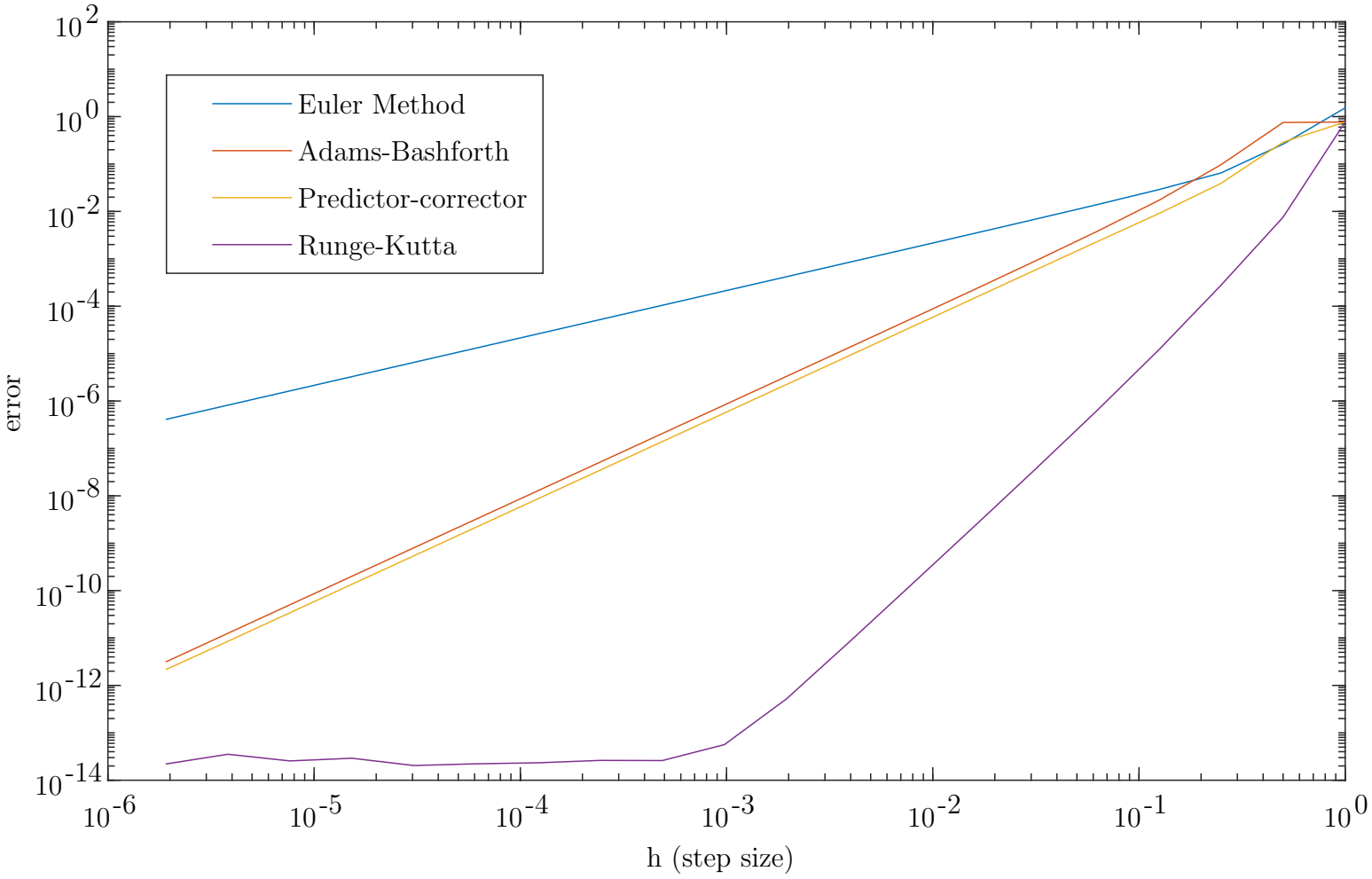

Figure 2: Log-log plot of the errors in solvers approximations. It clearly shows the advantage of higher order methods throughout the entire step size range.

Table 2: Runtime of solvers running the simulation for a year with a step size of 60 seconds. There seems to be a correlation between the order and the runtime of the solvers. Predictor-corrector being the most complex implementation-wise is the slowest.

Step size (seconds)	**Euler Solver**	**Adams–Bashforth**	**Predictor–corrector**	**Runge–Kutta**
60	4.0	8.0	18.0	15.0

Looking at these results we can see that the Euler Solver is the fastest method, followed by the Adam Bashforth, the Runge–Kutta and the Predictor Corrector that takes the most time. When we keep this in mind when looking at our results in 1, it is reasonable that the accuracy of the different solvers is nearly mirroring these results, since the Euler Solver has the biggest error in the approximation, which is followed again by the Adam Bashforth method. Now on the other hand, the Runge–Kutta method actually performs better than the Predictor Corrector method althogh the runtime is faster.

The reason that the Runge–Kutta algorithm performs the best considering accuracy is because, being a fourth-order method, it is our highest order method. These results show us that the Runge–Kutta method performs with such a high accuracy that the other solvers would have to run with a much smaller step size in order to achieve similar results. They most probably will still not be able to outperform the Runge–Kutta solver, as the runtime of the Runge–Kutta method is not increased by a similar factor. From this we can derive that it is the best choice for the solver we will use throughout the simulations in the application.

7.2 Impact of wind on the success of the landing procedure

We designed an experiment that measures the impact of different kind of wind on the success of the landing procedure. The experiment measures the distance to the target, the velocity when hitting the ground and the success of the landing. We defined two scenarios of a successful landing: if the velocity is below 5km/h and if the velocity is below 20km/h. The simulation is then executed 100 times for each type of wind in the project. The results are plotted in the following figures.

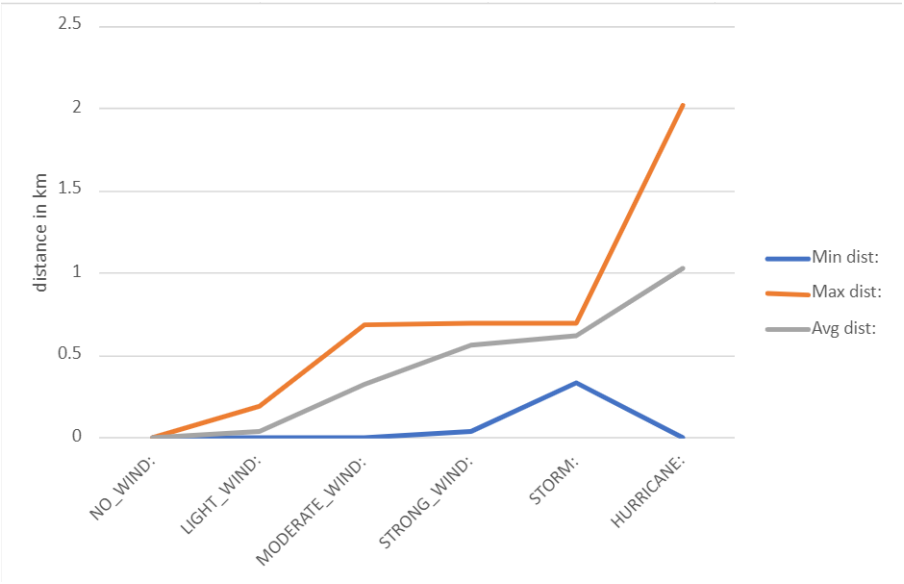

Figure 3: The minimum, maximum and average distance to the target after 100 iterations

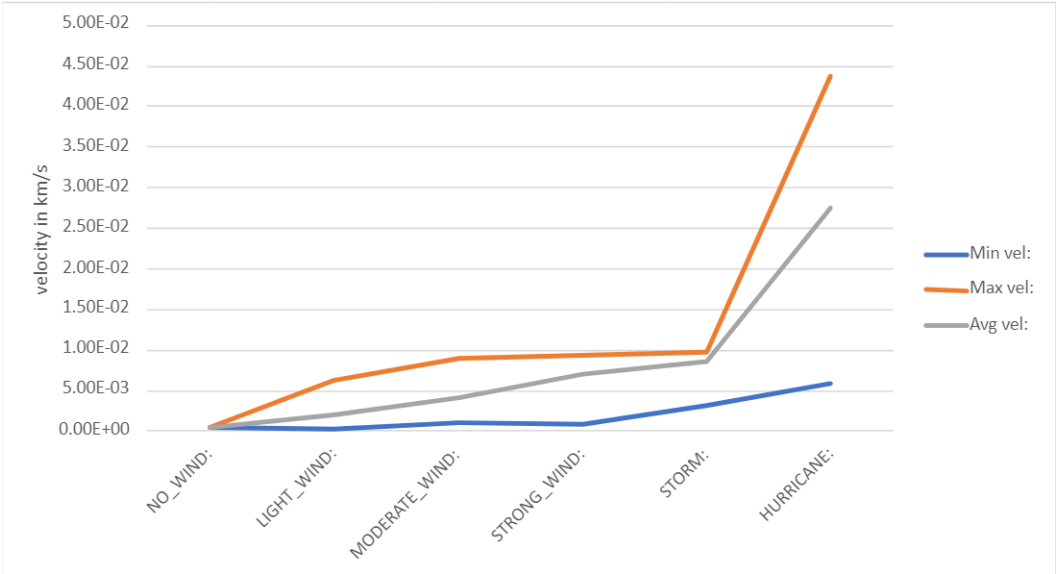

Figure 4: The minimum, maximum and average velocity when hitting the ground after 100 iterations

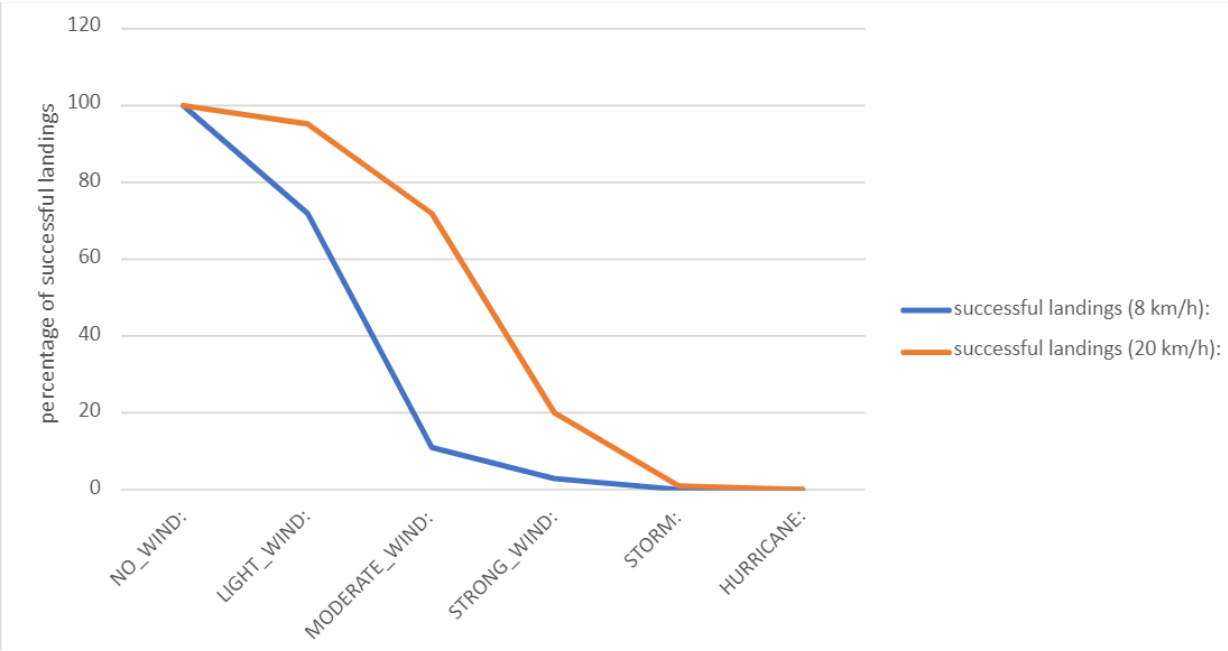

Figure 5: The percentage of successful landings after 100 iterations for an upper bound of velocity of either 8km/h or 20km/h

The results of this experiments show that with increasing strength of wind, the accuracy as well as the success of the landing is decreasing. While the landing is still successful to a certain degree in moderate wind, especially if we look at the landing condition of 20km/h, the success is decreasing rapidly when testing stronger types of wind.

This is a reasonable result and was expected when the experiment was set up. With a model of the atmosphere as complex as we have implemented it, it was already quite challenging to land the module successfully within the given bounds when no wind is acting in the system. The final position as well as the velocity spreads as the strength of the wind is increasing, and, depending on what the upper bound of velocity is considered to be an successful landing, the mission of landing the module on Titan is not likely to be successful if a moderate or strong wind is acting in the system.

8 Conclusion

In this report, we present a Java program simulating a space mission to Saturn's moon Titan. The program incorporates a GUI for interaction and visualisation of the solar system, as well as a physics model that accurately computes the gravitational forces and simulates the trajectories of celestial bodies: planets, moons, and the spaceship. The goal was to address several research questions related to accurately modelling the solar system, and implementing its physics, choosing differential equation solvers, and enhancing the simulation. Through using Java, we successfully represented the solar system and incorporated gravitational forces acting on celestial bodies into the simulation. This allowed for an accurate representation of the space mission to Titan. Different differential equation solvers were implemented and evaluated, including the Euler solver, classic Runge–Kutta 4th order solver, Adams–Bashforth 2nd order solver, and the predictor-corrector method. These solvers provided varying levels of accuracy and computational complexity. The results of the experiments demonstrated the influence of step size on the precision of the solvers.

Additionally, the program incorporated rocket launching and flight control functionalities. The rocket's engine firing and velocity calculations enabled the simulation of the space mission's launch, orbital entry and exit, landing on Titan and returning to Earth. Optimization algorithms, specifically the hill-climbing algorithm, were used to calculate the velocities required for the rocket's engine to reach Titan and return to Earth successfully. The experiments we ran showcased the accuracy of the differential equation solvers with increasing step sizes. The results demonstrated the trade-off between computational complexity and precision, emphasising the importance of selecting an appropriate solver for specific simulation requirements. Additionally the experiments in section 7.2 show the increasing difficulty to land successfully with increasing wind strength.

By accurately modelling the solar system, we gain insights into the complex structure and interdependent forces at play. This understanding can have broad implications for space travel and help measure the impact of different mathematical models and solutions in simulations.

References

Blanchard, R., & Lancaster, E. R. (1968). A unified form of lambert's theorem. https://ntrs.nasa.gov/citations/19690027552.

Collatz, L. (1960). *The numerical treatment of differential equations*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-05500-7_2

Faires, J. D., & Burden, R. (2012). *Numerical methods* (International 4th Edition) [ISBN: 978-0-495-38569-1]. Brooks/Cole.

Hao-ran, S., & Wei-dong, H. (2023). The long-term error estimation method for the numerical integrations of celestial orbits. *Chinese Astronomy and Astrophysics*, *47*(1), 177–203. https://doi.org/https://doi.org/10.1016/j.chinastron.2023.03.005

Heggie, D. (2006). Gravitational n-body problem (classical). In J.-P. Françoise, G. L. Naber, & T. S. Tsun (Eds.), *Encyclopedia of mathematical physics* (pp. 575–582). Academic Press. https://doi.org/https://doi.org/10.1016/B0-12-512666-2/00003-1

Johnson, A. W., & Jacobson, S. H. (2002). A class of convergent generalized hill climbing algorithms. *Applied Mathematics and Computation*, *125*(2), 359–373. https://doi.org/https://doi.org/10.1016/S0096-3003(00)00137-5

Lissauer, J., & Pater, I. (2019). *Fundamental planetary science: Physics, chemistry and habitability*. https://doi.org/10.1017/9781108304061

Murray, C. D., & Dermott, S. F. (2000). *Solar system dynamics*. Cambridge University Press. https://doi.org/10.1017/CBO9781139174817

Figure 6: Getting to Titan

Appendix

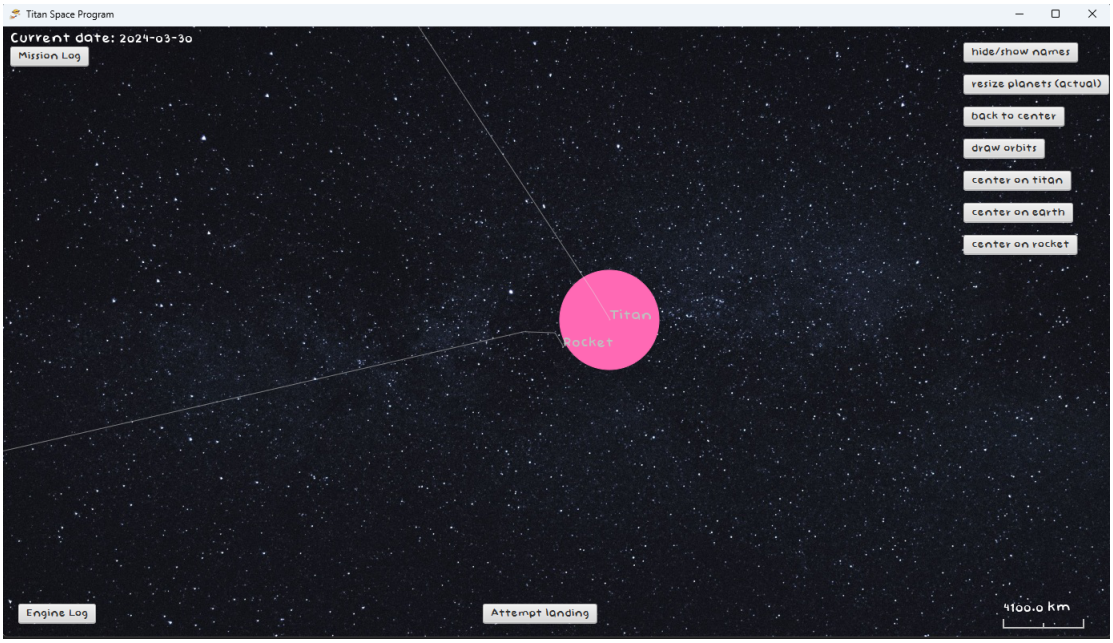

Figure 7: Entered Titan's orbit

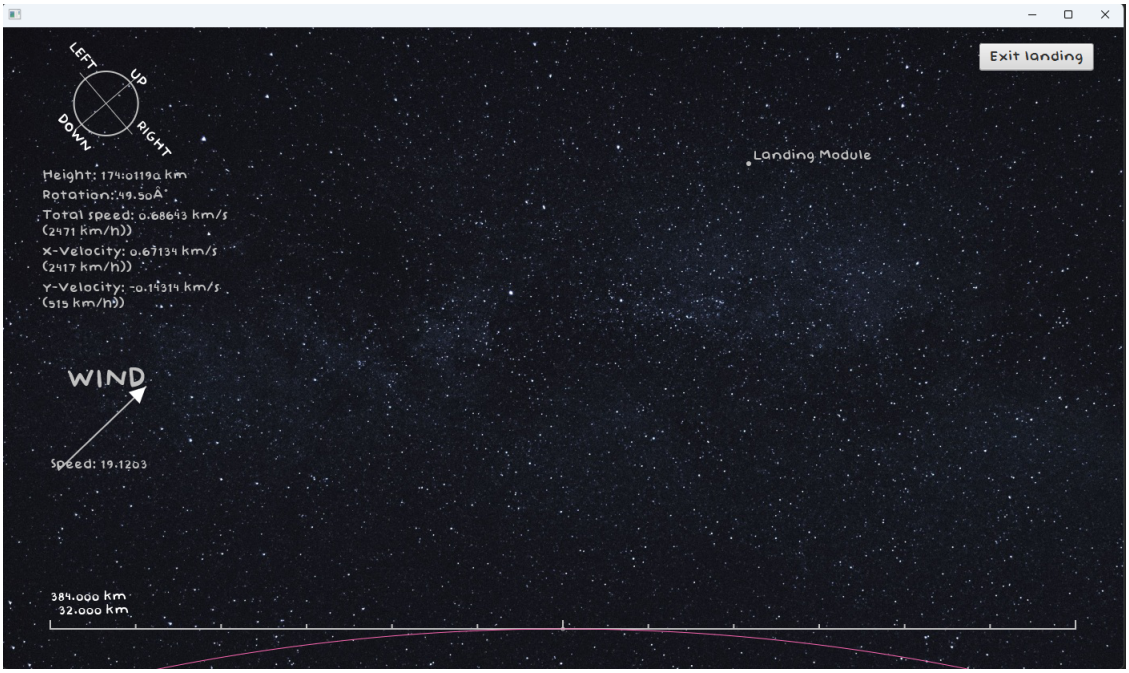

Figure 8: Landing module attempting a landing procedure

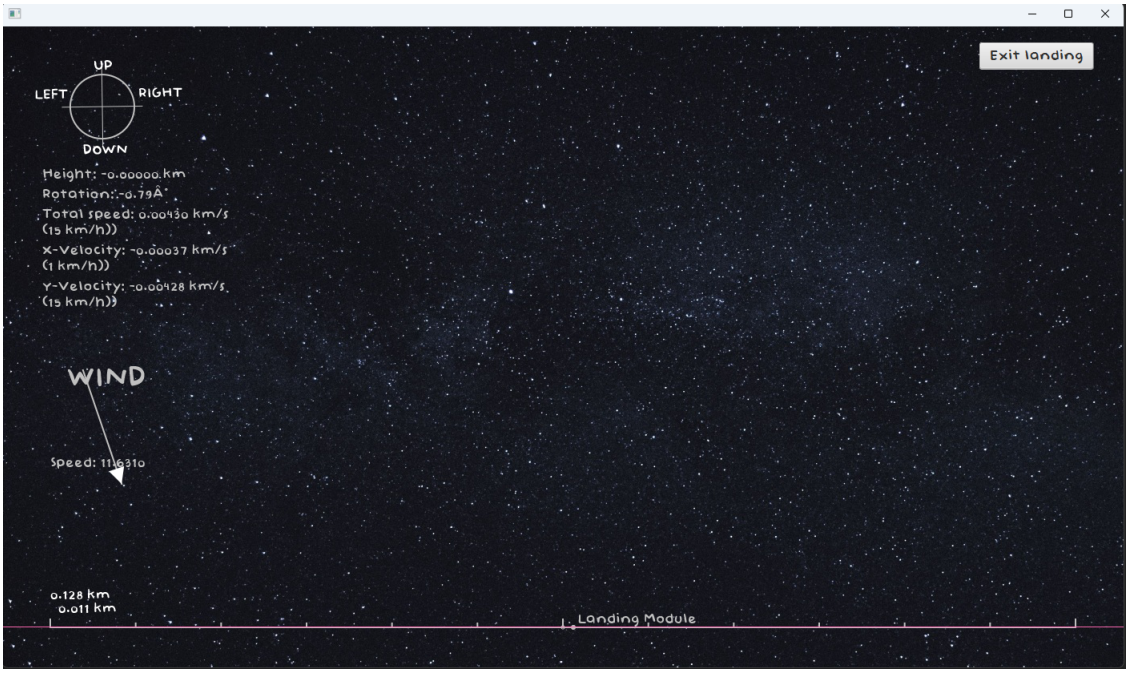

Figure 9: Successfully landed at Titan's surface

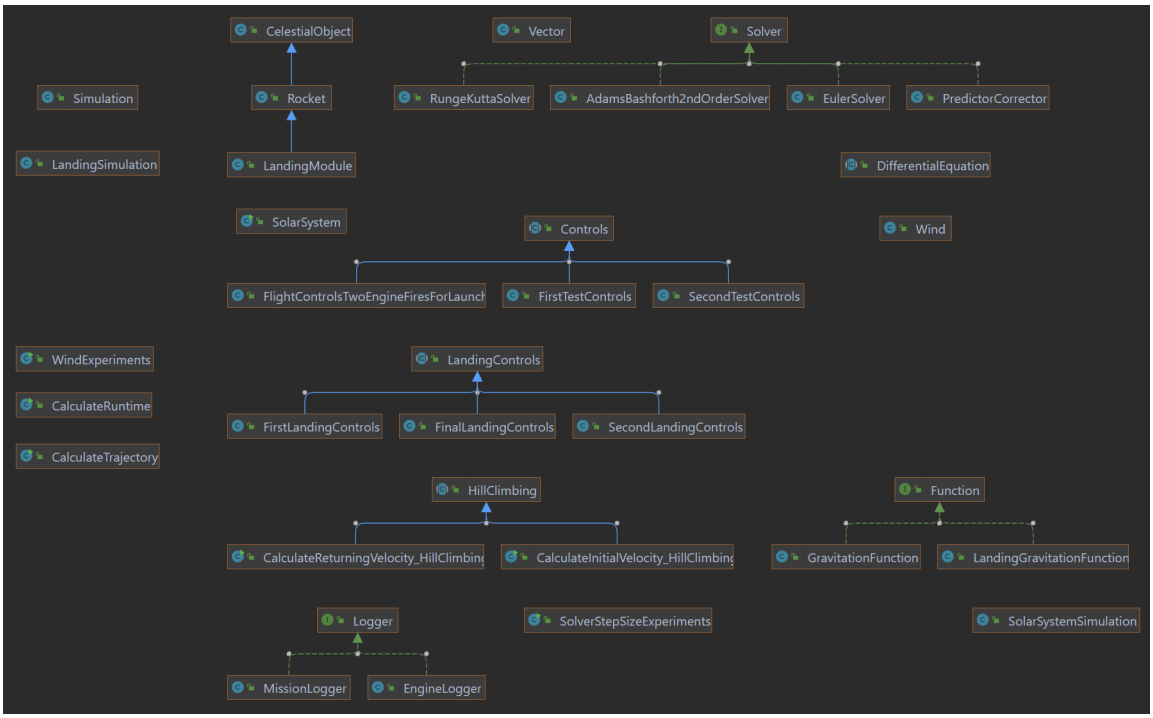

Figure 10: Java Class UML Diagram