|  |
| --- |
| Lab Report |
| Mars Lander |
| A guide to the modifications to the original source code to complete core tasks of developing an autopilot for landing and algorithms for numeric integration. In addition, notes and theory to complete some of the additional exercises. |

|  |
| --- |
| Oliver Jones  Christ’s College Lab Group 115  01.08.2017 |

# Introduction

The purpose of this task is to introduce the C++ programming language as well as an introduction to basic control theory. The importance of the choice of integration algorithm is key as it the instabilities introduced can cause unpredictable behaviour. This is initially tested in a simple 1-dimensional spring system, before being implemented in 3D.

# Spring Simulation

## Integral Types

The first task is to complete a Verlet integration algorithm for the provided Spring source code. The initial code demonstrates Euler integration and it can be shown that a smaller time-step will reduce the instability in the system, however it cannot be removed completely.

|  |  |  |
| --- | --- | --- |
|  |  | [ 1 ] |

The most important part of dealing with Verlet integrals is the initial calculation. As shown in [ 1 ], the current position and the previous position is required to calculate the next time-step. There are two possible ways of doing this:

* Apply the first time-step as a complete Euler integral and saving this for the
* Initialise the variable as

Both options produce similar results though for the purposes of simpler code, it is faster to initialise as an appropriate value then use Euler, as no acceleration or new velocity is required to be calculated.

## Conversion to C++

Processing the simulation through C++ has the obvious advantages of speed. With compiler optimisation, the code can run up to 20x faster than its Python equivalent. In an effort to further understand data manipulation in C++ additions have also been made that rather than writing the code out to a .txt file, each array is saved as its own binary file which can be read into python in a similar manner. Such changes do have a marginal effect on the performance as no ASCII conversion is required.

Similar additions are the use of command line arguments to choose the integrator type, the functions for which are kept in a separate file to better understand headers and forward declaration. If no such arguments are supplied the application requests a choice from the user.

Finally, the source code was modified to automatically call the python function and generate the necessary plots to show the evolution of the system over time.

# Orbital Simulations

Extending the previous task to 3D is fairly trivial when using 3 element arrays. In python, setting up some simple vector functions is straight forward particularly with numpy arrays. Setting the origin as the planetary centre, a circular orbit can be established by setting the velocity of the lander at 90 degrees to the position vector with magnitude as follows:

|  |  |  |
| --- | --- | --- |
|  |  | [ 2 ] |

Using Newton’s law of gravity, the velocity can then be updated as necessary to predict the path of the orbit. Multiplying by a scalar can allow simple conic paths to be shown

[MORE INFO ON RELATION TO DIFFERENT CONICS]

# Simulator Core Tasks

## Numerical Dynamics

The original source code, when compiled, merely shows a stationary lander that does not move in any of the scenarios, as the code does not have any dynamics calculations in it. Unlike the Orbits calculations done in Python, drag is also now a factor, as well as the thrust from the rockets of the lander. Rocket thrust is already provided, and just requires addition into the dynamics but all other calculations must be created.

To keep the code tidy a ‘Dynamics’ file was created. Initially this contained just the additional functions that calculate forces on the lander, but in later extension exercises more functions are added in to keep act as a single location to keep track of all additional functions not in the original source code.

The original functions in ‘Dynamics’ is the calculation of gravity based on the lander’s position, and separate lander and parachute drag calculations using the following equation:

|  |  |  |
| --- | --- | --- |
|  |  | [ 3 ] |

Atmospheric density is also provided in a function, which follows an exponential decay as the absolute position increases and cuts off completely at the edge of the Exosphere (altitude of 200km). The total area of the lander is 10x the lander radius, and for the purposes of simplicity all drag calculations assume the lander is moving with the base pointing in the direction of travel.

To deal with the integrators, a similar method is described as above in section 2. However, as a static variable is declared for ‘old\_position’, the value cannot be initialised straight away, as when the scenario is changed, it is not reset. Instead, a control statement sets its value when the simulation time is 0 (which it is reset to when the scenario changes).

## Autopilot

Once the numerical dynamics have been sorted correctly, the lander can be controlled manually. The difficulty in landing by hand in an of the scenarios, even with parachute assistance, is difficult, and an autopilot is required to consistently land safely. To land safely, the lander must not be moving more than 1m/s in any direction.

The autopilot will be based on a proportional control system, and the descent rate should reduce linearly as the surface approaches:

|  |  |  |
| --- | --- | --- |
|  |  | [ 4 ] |

Where the ideal value of is the descent rate we want to be at when the lander touches the ground, and is a positive constant. An error term can be defined as the difference between the two sides of the equation:

|  |  |  |
| --- | --- | --- |
|  |  | [ 5 ] |

This gives us our Proportional output:

|  |  |  |
| --- | --- | --- |
|  |  | [ 5 ] |

Again is a positive constant. At our ideal velocity, there should be no acceleration so the thrust from the lander should balance its weight:

|  |  |  |
| --- | --- | --- |
|  |  | [ 7 ] |

Where is the maximum thrust capable from the rockets. With these definitions, we can define the throttle regime:

|  |  |  |
| --- | --- | --- |
|  |  | [ 8 ] |

Tuning the values of and allows you to produce an autopilot regime that can either minimise fuel usage or force on the lander. The least fuel possible would mean the throttle firing as late as possible and staying on maximum until the lander touches the ground, as this makes the most of the breaking force provided by the atmosphere. Conversely the minimum force on the lander would require as early a firing as possible, and the fuel just running out as the lander touches down. Table 1 shows tuned values for when there is no parachute available and is kept constant.

Table 1 Values of at of 1 that provide best fuel efficiency and softest landing

|  |  |  |
| --- | --- | --- |
| Scenario | Most Fuel Efficient | Softest Landing |
| 1 | 0.04125 | 0.0137 |
| 3 | 0.01812 | 0.01625 |
| 5 | 0.01898 | 0.01675 |

### Determining Parachute Release

The simulator will cause the parachute to be lost if the force on I exceeds a certain force (causing tethers to snap) or if the lander is travelling too fast (causing is to vaporise). Therefore, it is important to have a logical system that will decide an appropriate time to release the parachute to make the most use of its breaking abilities while not causing it to be lost.

The source code provides an indicator for when either of these two conditions are met, and initial systems simply checked whether these values are true or false. This works well for scenario 1 as the parachute can be released instantly with no issues, however in scenario 5, as the lander is at the edge of the exosphere, there is not sufficient atmosphere to stop the lander accelerating, causing the parachute to vaporise.

It is safe to assume that when the thrusters fire the lander will not be exceeding the maximum safe conditions once it returns from that regime. Therefore, the system was modified with a second Boolean to confirm whether the initial firing of the rockets had begun, if so and safe to deploy parachute it will. This provides a much more reliable way of deploying the parachute, though has issues in scenario 1 in that it can be released much earlier than when the thrusters are first fired.

Hence, to cover all possible scenarios, a prediction function was created that will return true if it is *useful* for the parachute to be released now. This uses a series of virtual parameters, and predicts how the velocity of the lander will evolve over time if the parachute is released now[[1]](#footnote-1). The function will return false if it predicts evolution into an unsafe regime or if the lander virtually crashes, but returns true if the velocity reduces before any of these happen. This provides an adaptive parachute launcher which, when combined with the autopilot, allows the lander to touch down successfully in all the scenarios that don’t have stable orbits.

1. To save computing power, the function uses Euler integrals with a slightly larger time step than the actual simulation as only a rough estimate is required for reliable results. [↑](#footnote-ref-1)