|  |
| --- |
| 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  13.08.2017 |

# 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:

|  |  |  |
| --- | --- | --- |
|  |  | [ 6 ] |

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 ] |

For information on selecting the values of refer to section 5.4.

### 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-2). 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.

# Extension Tasks

## Aerostationary Orbit

To produce an aerosationary orbit, the lander must be moving such that its orbital period is equal to that of Mars. From Kepler’s law of planetary motion:

|  |  |  |
| --- | --- | --- |
|  |  | [ 9 ] |

Setting as appropriate allows is to calculate the appropriate radius. Then from simple circular motion. Plugging this into [ 2 ] means that we can calculate the appropriate speed to maintain that circular motion.

## Any Angle Attitude Control

Currently the attitude stabilisation set the orientation of the lander such that its base is facing the centre of Mars. To allow for planetary rotation and wind, any angle attitude control must be possible. Fortunately, we already have a set of vectors aligned with the centre of the lander as well. By crossing one of these vectors with the position, we get an axis that is perpendicular to the plane of the screen which we can rotate about.

To form a rotation about an arbitrary axis we can use the following rotation matrix:

|  |  |  |
| --- | --- | --- |
|  |  | [ 10 ] |

Where is the normalised axis of rotation and is the angle of rotation about that axis. Applying this matrix does allow for some rotation control, however due to a phenomenon called gimble lock, getting into a certain orientation then removes one degree of freedom. Furthermore, calculating the elements of the matrix is computationally expensive and can slow down the programme, particularly at higher speeds.

An alternative to the rotation matrix is using a technique called quaternions. This is a set of complex numbers extended to a 4-dimensional space (3 imaginary and one real). Many computer games use this technique over matrices as they do not have the issues described above.

|  |  |  |
| --- | --- | --- |
|  |  | [ 11 ] |

Applying the rotation quaternion as defined above on the ‘up’ vector (converted to a quaternion with a real part of 0), and will rotate the given ‘up’ vector in the attitude stabilisation function to the desired orientation, causing the lander to turn. Fortunately, many quaternion functions had already been defined in the original source code so this could be implemented in a few lines. An additional statement was also added that if the stabilised attitude angle was approximately 0, then set up to the normalised position vector to save the computation.

Additional controls were then also added to allow manual changing of the angle by incrementing the stabilised attitude angle.

## Planetary Rotation Mechanics and Wind

### Rotation

Currently, while the orbital view visuals show the rotation of the planet, the mechanics themselves do not take this into account. To correct this, a new relative velocity variable needs to be defined. This is calculated by using similar methods as described in section 5.1, by saying ‘if the lander was on the surface now, how fast would it need to be moving to stay on the same spot on the surface?’ By adding this additional speed onto ground speed and drag calculations, the rotation of the atmosphere will accelerate the lander in the appropriate direction as with 0 velocity is will now have a significant speed in relation to the surface.

### Wind

Adding wind mechanics to this was straightforward. A new Dynamics function was created to create a normal distribution about a mean wind value and return a wind speed to the drag calculations. Using a normal distribution allows for slight variations in the wind, and increasing the standard deviation of the distribution increases how gusty and varying the wind can be.

A new switch was added to the ‘w’ key to allow the wind to be toggled on and off.

### Modifying Autopilot to cope

With planetary rotation, there is now a lateral motion that needs to be considered. Using the parachute, no change needs to be made as the drag provides the necessary force to reduce the ground speed to 0. However, without the parachute, thrusters now need to fire at an angle to slow both descent and ground speed.

Initial solutions were to set the stabilised attitude angle such that the lander pointed against the direction of the velocity. However, this caused and odd bug where just as the lander was about to touch down the lander would flip on its side causing the ground speed to suddenly increase again. The second solution was to set the ‘up’ vector in the stabilised attitude function to the direction of velocity, but again this caused similar bugs. Interestingly combining the two appears to stabilise the autopilot and remove the bug altogether so worked as an appropriate work around, though why it works is unclear.

When the wind speed increases there is the danger that the parachutes drag will work against the thrusters so the autopilot will eject the parachute when the ground speed of the lander approaches the wind speed and it is near the ground, so that the wind does not drag the lander into a crash.

## Tuning Autopilot

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 firing as early as possible, and the fuel just running out as the lander touches down.

To tune these values to their optimum, an interval bisection algorithm was set up that calculates the value of for the preferred optimisation. The function itself creates a copy of the lander object at the instance the function is called (by using the class explained in section 6.1), and running the simulation repeatedly with different values of based on the interval bisection algorithm. Importantly a timeout clause is added to stop the function if the simulation time in the function exceeds a certain amount, as in scenarios such as stable orbits a crash will never be registered.

This value is recalculated if a change in circumstance is forced such as removing the parachute or disabling then re-enabling the autopilot. A toggle key of ‘m’ was added to allow the user to select the preferred landing optimisation.

Table 1 shows tuned values for when there is no parachute available and is kept constant. The values for this table were determined after the introduction of planetary rotation.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Scenario | Without Parachute | | With Parachute | |
| Most Fuel Efficient | Softest Landing | Most Fuel Efficient | Softest Landing |
| 1 | 0.03931 | 0.01374 | 0.15273 | 0.01030 |
| 3 | 0.02001 | 0.01735 | 0.05588 | 0.01382 |
| 4 | 0.03486 | 0.01637 | 0.15160 | 0.01128 |
| 5 | 0.01909 | 0.01670 | 0.05305 | 0.01440 |

## Autopilot Modes

The main autopilot mode can be considered Orbital Descent. There are two additional things the autopilot can do:

1. Orbital Re-Entry: bring the lander from any stable orbit into a ballistic trajectory that leads to Orbital Descent, allowing the lander to touchdown safely.
2. Orbital Injection: Going from a ballistic trajectory into a stable orbit i.e. the reverse of Re-entry.

# Coding Practice

As well as being an exercise in control theory, first and foremost this task is about developing an understanding of the C++ coding language. The following section details changes made to the original source-code that, while not adding extra functionality, improves readability of the code and its structure.

## Orbiter and Lander Classes

Being an object-oriented programming language, classes allow groups of functions and variables to be kept in a single location, as well as control what can and cannot change those variables. All the lander variables and corresponding physics functions (including thrust and attitude stabilisation) were merged into a class that contains all the basic properties of the lander. It was then possible to remove repeated calculations of this like altitude and drag and put them in one location, making debugging more straightforward as only 1 location needs to be changed, as well as minor performance improvements by not recalculating dependant values multiple times in a loop.

All of this is put in the ‘Orbiter Class.h’ file, though the functions where applicable were kept in their original location to maintain the code structure. Geometric arguments such as position and velocity are kept as protected so that only the numerical dynamics function and resetting the scenario can explicitly change it to reduce the likelihood of accidentally changing values somewhere unknown.

The number of files declared in the header files could be slashed, as now only one declaration of a lander variable is required, and it will be initialised to contain all the appropriate variables.

## Header Files

Header files allow functions and global variables to be forward declared so that the order they are defined does not need to be considered, and that functions from different files can be accessed. Best practice suggests that every .cpp file should have its own .h file with the same name so that it is clear where declarations are kept, and do simplify any includes. The ‘lander.h’ file was therefore split into 2 separate header files for the corresponding original .cpp files.

Now with multiple files and being included at the top of each file, repeated declarations will cause errors in compilation. Head guards using the pre-processor #ifndef create conditional inclusion in the file if it has not been already (from being included within a header file previously defined). It should be noted #pragma once can also be used but not all compilers have this pre-processor directive.

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-2)