Mars Lander- Sahil Sindhi

The area of the Mars Lander project which I was most keen to explore further was the control systems in particular designing autopilots for the lander to execute specific tasks while also optimising some aspects of the task. So, I decided to tackle some of the extension tasks related to this.

-I set up an aero stationary orbit by working out the relevant orbital radius and velocity by some circular motion mechanics which can be viewed by running scenario 6.

-I have also added a key to control the attitude of the lander. In order to rotate the lander by 18 degrees clockwise press the ‘r’ or ‘R’ key.

Autopilot

The autopilot has been updated to make use of the parachute, initiate orbital re-entry and a separate autopilot designed for orbital injection given an infinite fuel supply. Several powered decent autopilots have also been designed to minimise decent time, fuel consumption and peak acceleration. The proportional autopilot has also been tuned iteratively to be able to land with engine lag of up to 3 seconds and engine delay of 1 second on top of that.

-In order to activate the landing autopilot, use the ‘a’ or ‘A’ key as usual and make sure the fuel rate is set to 0.5.

-In order to activate the orbital injection autopilot, use the ‘z’ or ‘Z’ key and make sure to set the fuel rate to 0.0 and engine lag and delay are also set to 0.0.

Orbital injection autopilot:

-The orbital injection autopilot works by orienting the lander at an angle and applying the thrusters, so the lander can gain height and ground speed necessary for the orbit until the projected periapsis is above the atmosphere. I have created a function that calculates the projected periapsis and apoapsis of the orbit using the current position and velocity vectors.

-The orbital injection autopilot is capable or injecting the lander into orbit from scenarios 1 and 5 and 3 if the autopilot Is activated near the max altitude of the polar launch

Landing autopilots:

There are a range of autopilots for landing optimised for time, fuel and acceleration using machine learning techniques a brief explanation of which can be found below. In order to choose which of these to employ set the variable AUTOPILOT which can be found at the top of the lander.cpp file to the appropriate number according to the comment above the variable prior to compiling and running the project. The autopilot optimised for decent time and fuel for the 10km decent achieved a decent time of 128.8s compared to 308.4s for the proportional controller and fuel left of 69.1l compared to 22.9l which in fact meant the fuel optimised autopilot had more fuel left after landing successfully from the aero stationary orbit! The peak acceleration optimised autopilot for 10km achieved 2.5 ms^{-2} compared to 3.7 ms^{-2}.

Deorbiting-

This was achieved by rotating the lander to have the thrusters fire in the opposite direction to the current velocity to slow down the lander until the projected periapsis was below the atmosphere and aerobraking handles the rest. In order to minimise fuel usage, the thruster firing was done at the most optimal stage in the orbit considering the Oberth effect.

Powered decent using Neural Networks -

I have recently been exploring the exciting world of machine learning and to this end I gravitated towards (pun not intended) a machine learning approach towards optimising decent time, fuel consumption and peak acceleration. I spent an extensive time learning and researching algorithms and ways to implement these. One of these was the python TensorFlow library.

My first approach was implementing a reinforcement learning algorithm so the lander would learn to land on its own while optimising features. My first attempt was using Deep Q learning. Deep Q learning uses a neural network to take some inputs and output a score for each discrete action that can be taken. The highest scoring action is the one that should be taken, or a random action is taken in order to allow the model to explore. Then the rewards due to that action and future rewards are used to update the neural network to make better predictions on what action to suggest given a current state. Firstly, to implement this I had to discretize the throttle to 11 throttle settings and replicate the powered decent portion of the landing in python in an object orientated manner to run the algorithm on the model using velocity and altitude as inputs. I spent a significant amount of time researching ways of improving the process and implementing these such as using replay memory and target networks and playing around with the range of parameters such as learning rates of the network the exploration probabilities of the network, network topology, reward function etc. Unfortunately, I was not able to get this approach to work(I think it may have been that the reward of landing or crashing was too far into the future) and when it did start to get close to landing it soon got a lot worse but the code has been included for you to look at in the Deep Q learning file along with a plot of the moving average reward. I plan to revisit this when I have a greater understanding of the effects of the parameters.

I looked for an approach that would require less parameters to tune and could provide me with a continuous throttle value output. This is when I discovered evolutionary algorithms and NEAT was particularly appealing for several reasons. So, I learned how to implement this and used it on my python model of the lander, and this also came with parameters to configure in terms of mutations to the network and breeding etc. I set these to sensible values and started to tweak the reward function in order to teach it too land while optimising something. At First a negative reward was given at the end of the episode for crashing, this did not work well as it would always crash and never learn how not to crash. After some experimenting I set the reward to be the negative of the velocity at the time of landing and so it learnt to minimise this which in turn would achieve landing and then if it landed it would be awarded a positive reward on top of this based on how well it optimised a certain objective. This approach worked well as the first few generations would build and evolve the landers capable of landing and later generations would breed landers that also optimised the best. The code for running this evolution process is also included In the NEAT folder along with some velocity profiles of optimised landers.

The NEAT algorithm proved to be quite powerful in terms of the speed of training and results obtained. Given more time I would have liked to have gone on to develop landers that learn to inject themselves into orbit and land from orbit too!