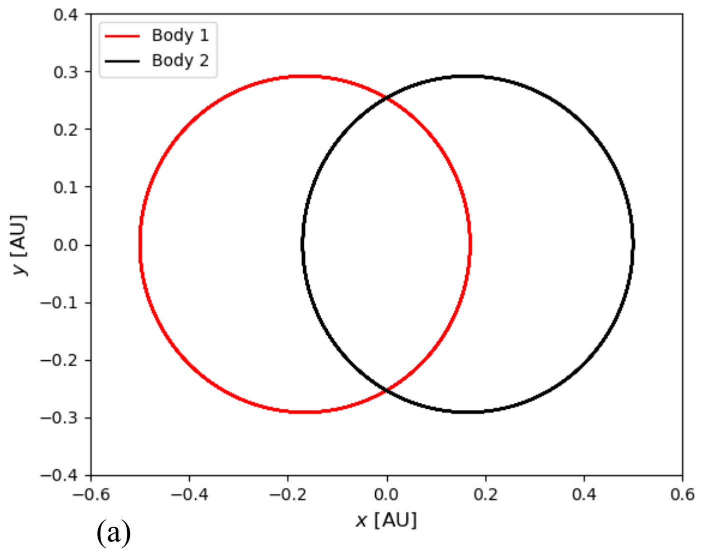
Scientific Computing Project - Gravitational N-body Simulation

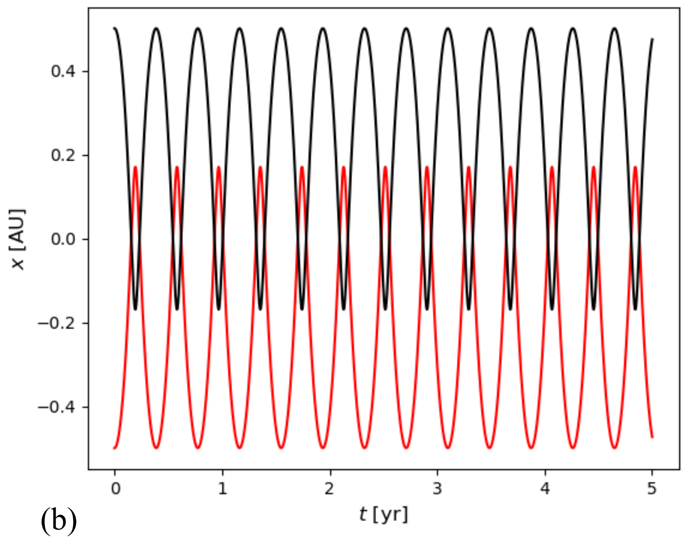
<u>About the model:</u>
This project uses as direct summation approach to calculate the gravitational acceleration experienced by each body, *i*, in the simulation, applying Newtonian gravity and solving the equation:

$$\frac{d^2\mathbf{r}_i}{dt^2} = \sum_{j\neq i}^{N} \frac{Gm_j\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3}$$

Where $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$. This approach is slower than many of the methods used by computational cosmologists, but it is simple to implement and produces very accurate solutions. The summation is not computed explicitly, one j index at a time, but rather using the numpy sum function along specific axes, and by slicing arrays up in units of N and D (number of bodies and number of dimensions) to generalise the algorithm so it may solve problems with any number of bodies or spatial dimensions.

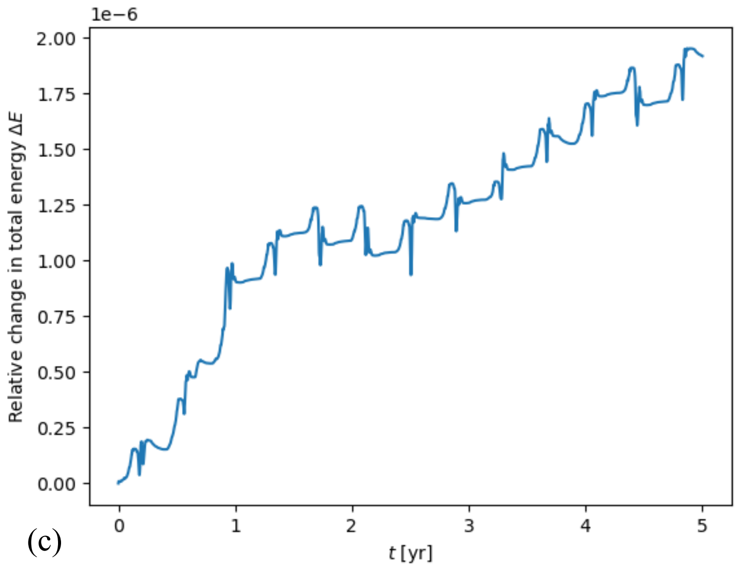


(a)



(b)

Figure 1: Validation test. These images were generated using test_case.py, with Figure 1a showing the orbital paths of the two bodies, Figure 1b showing the variation in the x coordinate of each of the bodies with time, and Figure 1c showing the relative change in total energy of the system. Figure 1c shows the code conserves energy well, as the difference in the total energy compared to the initial value is on the order of 10^{-6}. test_case.py uses a non-generalised algorithm to solve the differential equations, but figures equivalent to 1a and 1c can be generated using a generalised version of the algorithm if you run n_body.py simulation number 1.



(c)

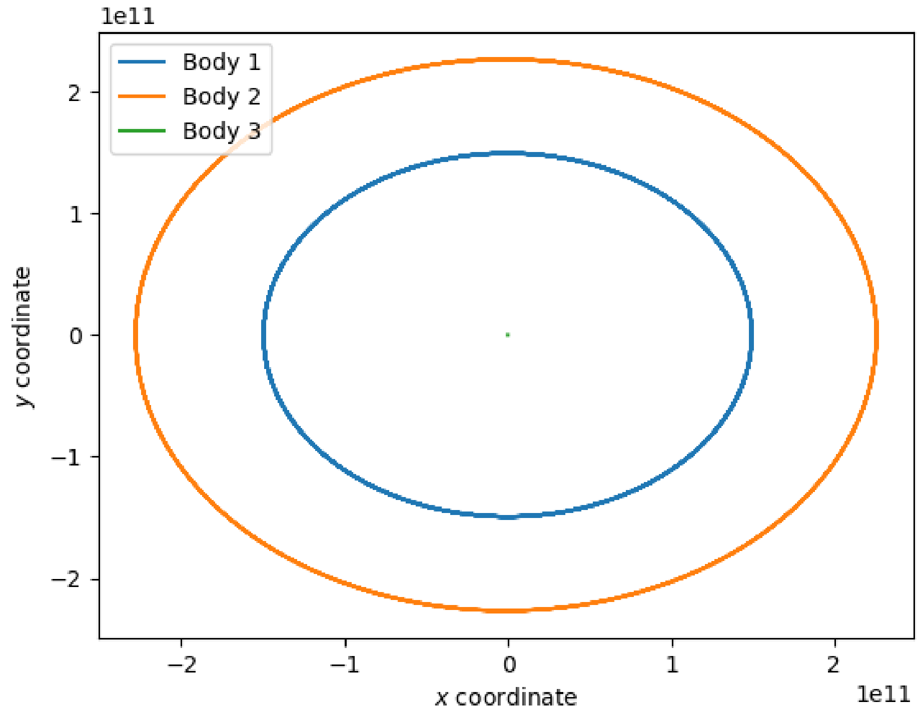


Figure 2: This image was generated by running simulation number 2 in n_body.py, and shows the orbital paths of two light bodies orbiting a heavy body (Mars, Earth and the Sun). It can be seen from the values on the axes that the orbits are circular and stable. Producing this figure was the first step in generalising the code in test_case.py, as it brought the number of bodies from two to three. At this stage in the development process the differential equations were solved in vector form for the first time.

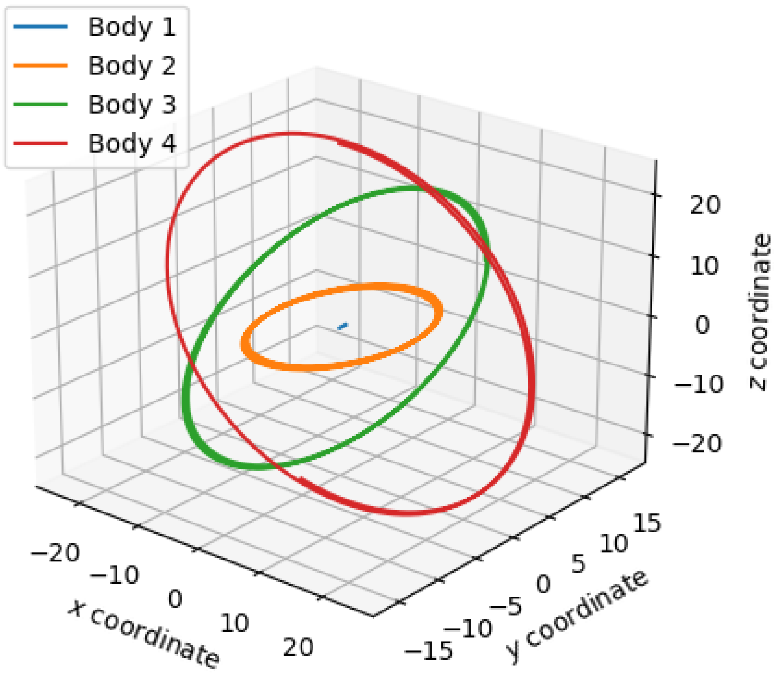
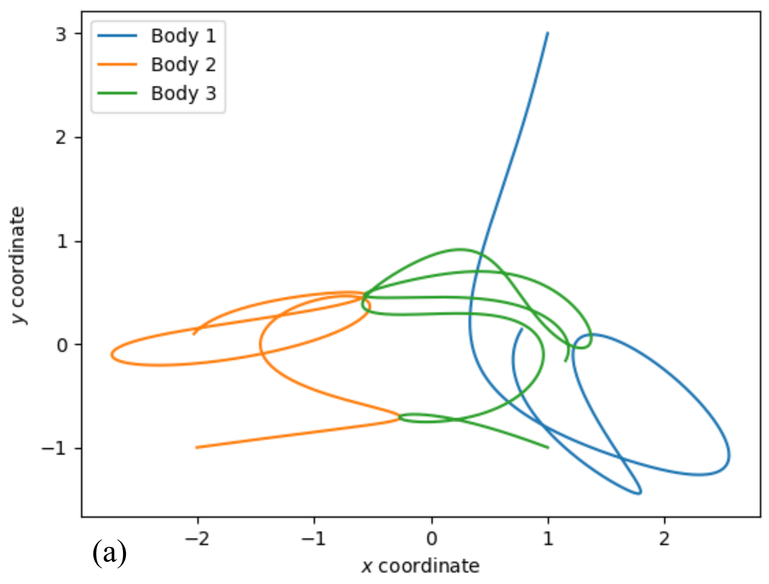


Figure 3: This image was generated using n_body.py, by running simulation number 3. It shows a 4 body system where a large massive body is orbited by three smaller bodies in three dimensions – with orbital planes in the x=0, y=0 and z=0 planes respectively. The lines in the plot are quite thick, as they do not trace over the same path between each orbit. This is because body 1 is moving. Though initially stationary, it picked up some velocity due to its attraction to the other bodies, and since the others are gravitationally bound to it they get dragged along too. Producing this first 3D plot marked an important step for the project, as it necessitated further refinement of the differential equation solving algorithm, which by this point could technically handle any number of bodies and any number of spatial dimensions. More work would need to be done however before it could do so in a timely fashion.



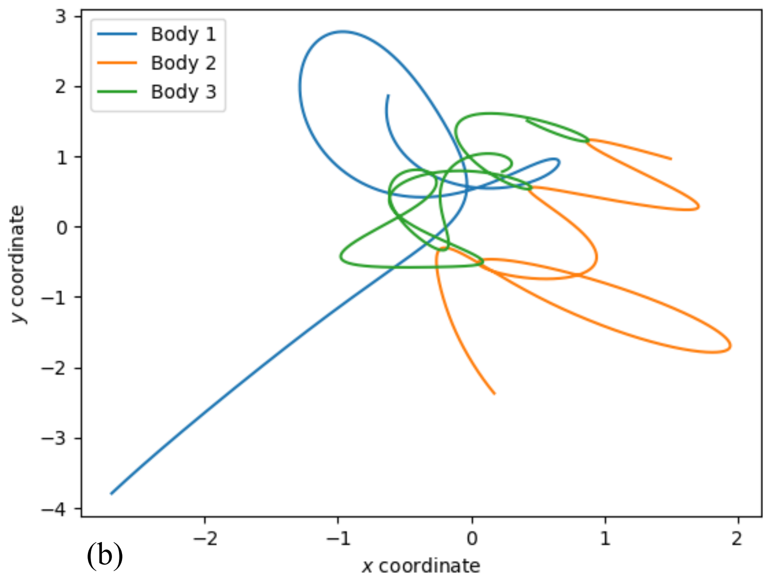


Figure 4: Burrau's problem (with the full details of the problem given in the paper "Complete Solution of a General Problem of Three Bodies" by Szebehely and Peters, DOI: 10.1086/110355).
These figures were produced by running simulation number 4 in n_body.py, with the plot limits t_llim and t_ulim set to plot the path of the bodies between specific times (see lines 124 and 125). In the case of Figure 4a, this is from t = 0 to t = 10, and for Figure 4b it's t = 40 to t = 50. t_tot can be set to any value so long as it is greater than t_ulim. The paths shown in these plots look identical to those shown in the aforementioned paper, indicating that my simulation is highly accurate up to t = 50. After this point however, as we shall see in subsequent figures, the match is not as close.
It was at this stage of the development process that it became necessary to optimise the speed of my solving algorithm, as Burrau's problem requires tight integration tolerances and long simulation times, which both strongly affect how long the calculations take to perform.

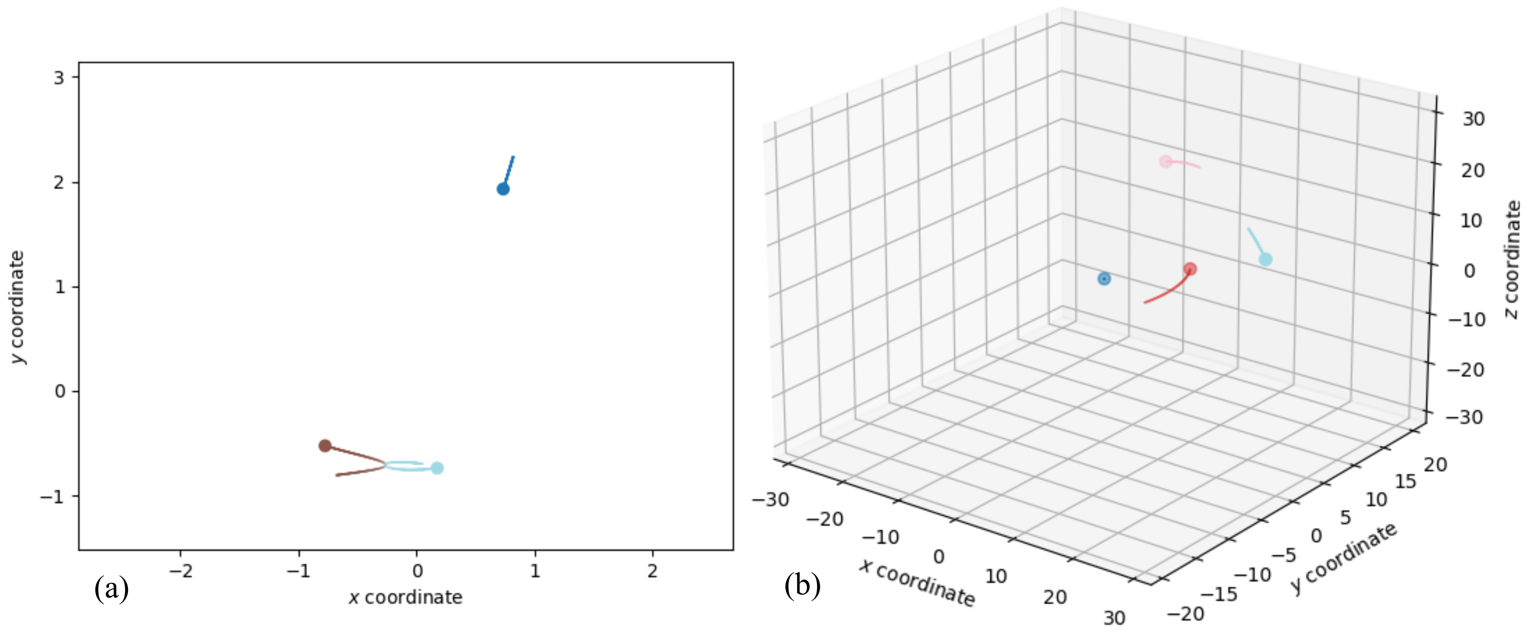


Figure 5: Animations. Figure 5a and 5b show frames from animations generated for simulations 4 and 3 respectively in n_body.py. An animation can be generated for any of the simulations in n_body.py, as well as in solar_system.py, and works by plotting a scatter plot of specific points from the solutions array (generated by the ODE solving function). The sizes of the scatter points are set such that each body has a large "head" representing the position of the body, and a thin "tail" a fixed number of points long, such that the apparent physical length of the tail grows the faster the body moves. This gives a good visual representation of how the position and velocity of each of the bodies, and provides a more intuitive way of watching the system evolve over time. Each body is also given its own colour, though the method used to do this prevents the use of a legend to identify each body by name.

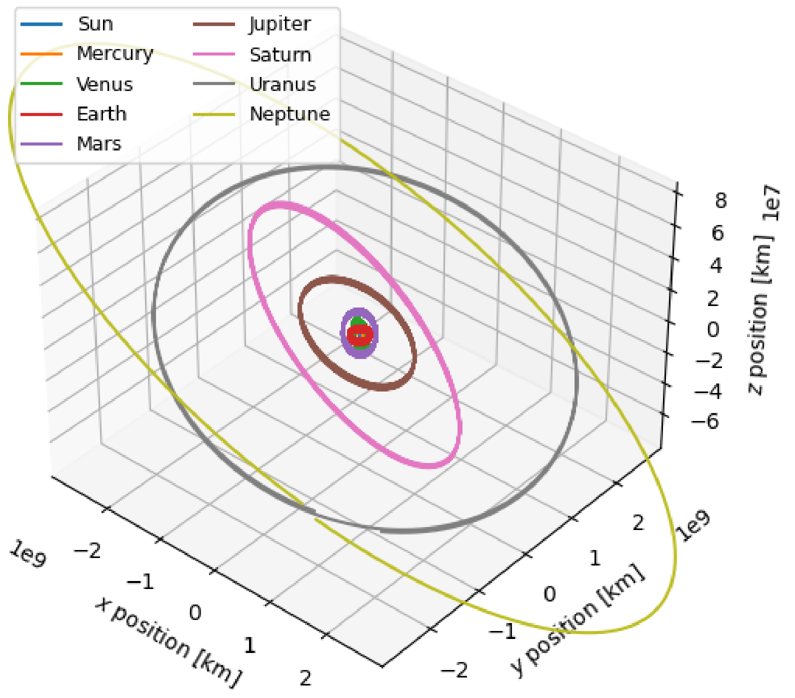
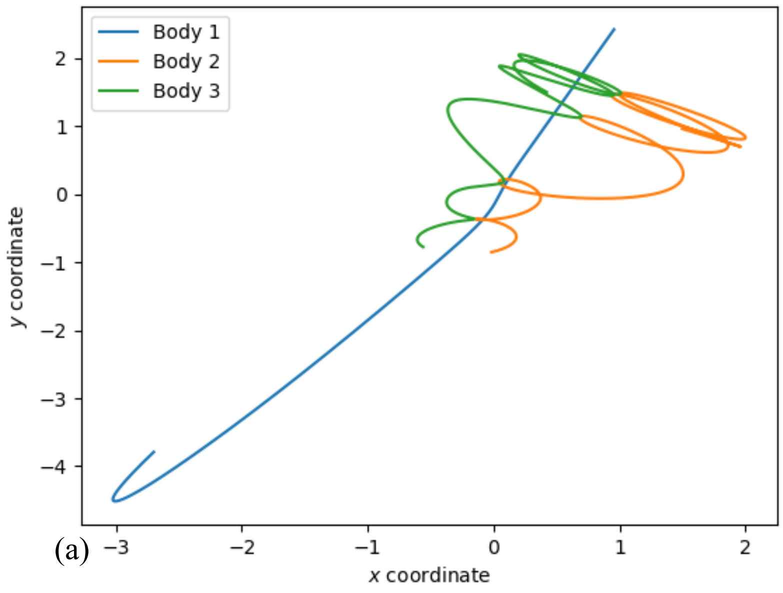
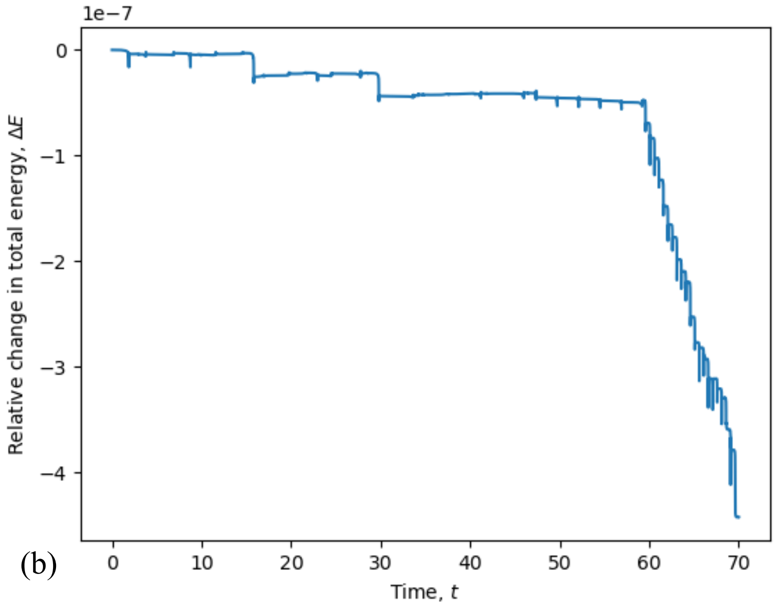


Figure 6: This figure was generated using solar_system.py, with the "bodies_to_omit" and the "newbody_…" sections of code commented out. It manages to simulate 9 bodies over ~164 years in under 2 minutes, which is quite fast considering the scale of the problem. This figure shows the same line broadening as in Figure 3, for the same reason. I would recommend having a look at the animations in this script, though only if bodies beyond Jupiter are omitted, as otherwise the animation runs too slowly to watch.

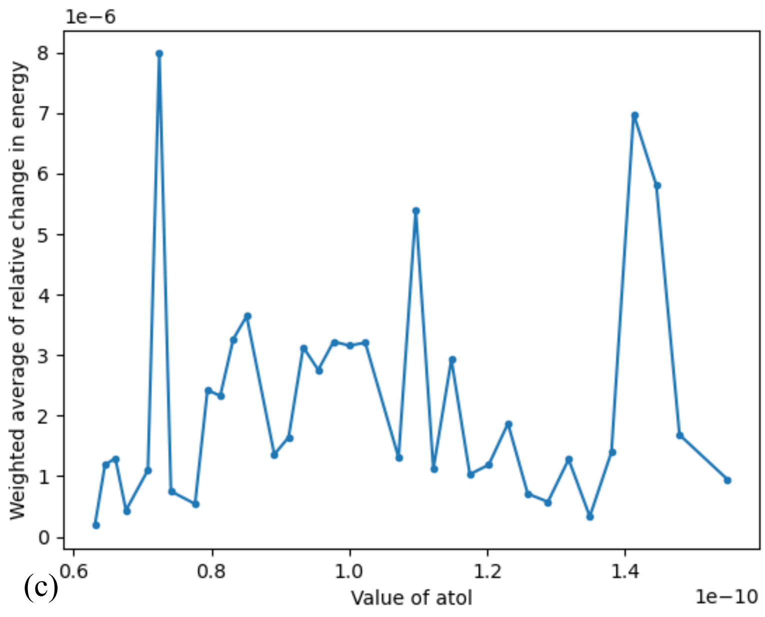
Figure 7 (Below): Optimising Burrau's problem. In order to find and optimal solution to Burrau's problem, we need to find values for the integration tolerances (rtol and atol) which produce a solution which conserves energy better than any other. To achieve this, I first used trial and error when adjusting these tolerance values until I found a solution which looked similar to those from the paper in the interval t = 50 to t = 60, eventually settling on rtol = $1*10^{-26}$ and atol = $8.144*10^{-11}$. This was used as a model for a good solution. The path of the bodies as well as the energy conservation graph for the whole simulation can be seen in Figures 7a and 7b, plotted using n_body.py and simulation 4 with the tolerances set accordingly. Using optimising_burraus_problem.py, I then tested a range of possible values for atol (since varying atol produced a more significant effect than varying rtol), solving the system of differential equations for each one and assessing which was best by finding the average energy difference from the initial over the course of the simulation.
Since energy conservation is of upmost importance, particularly towards late times where close encounters are more frequent, this average was weighted according to the energy of my best trial and error solution. Figures 7c and 7d show the results of this process, with the latter being a refined and more detailed version of the former. According to this process, the best solution comes from a value of atol = $9.549925860215698*10^{-11}$, which when plotted using n_body.py gives Figures 7e and 7f. Figure 6f indicates this solution conserves energy better than Figure 6b, but their respective path plots 7e and 7a tell the opposite story, since 7e is markedly different from the figure in Szebehely's paper. This implies there is that there is either a flaw in my methodology (likely, but I haven't the time now to go looking for it), or that energy conservation is not the only metric by which a good simulation should be judged.


Body 1
Body 2
Body 3
y coordinate
x coordinate

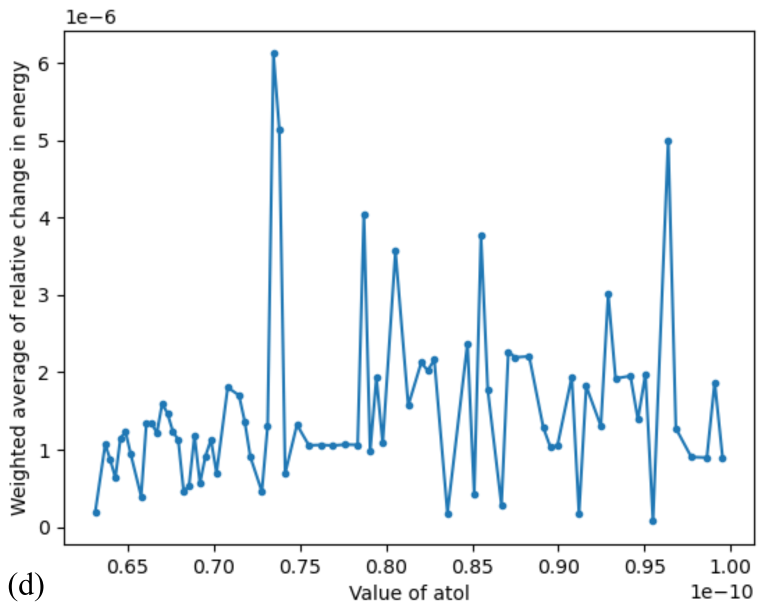
(a)


1e−7
Relative change in total energy, ΔE
Time, t

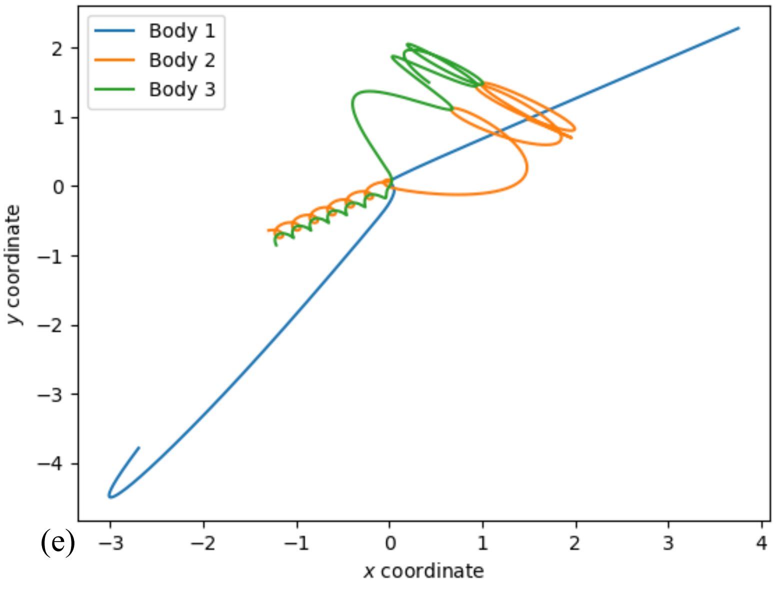
(b)


1e−6
Weighted average of relative change in energy
Value of atol
1e−10

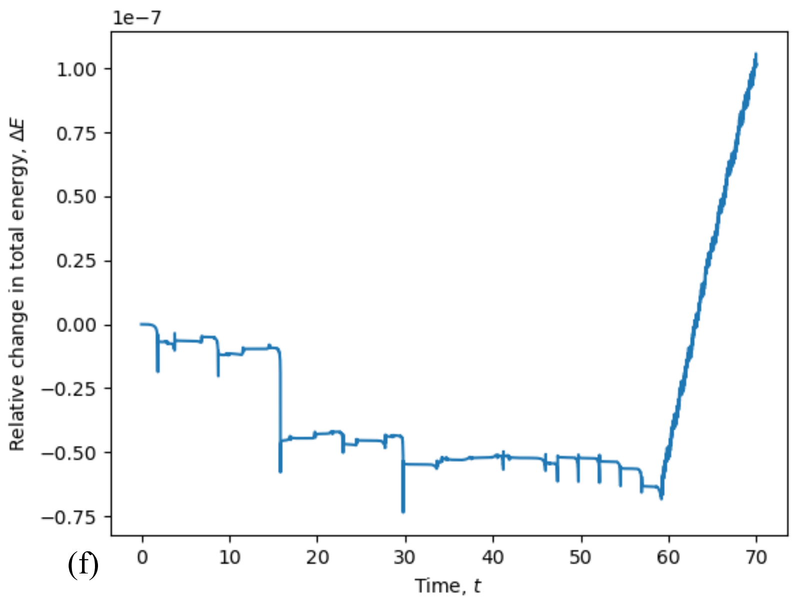
(c)


1e−6
Weighted average of relative change in energy
Value of atol
1e−10

(d)


Body 1
Body 2
Body 3
y coordinate
x coordinate

(e)


1e−7
Relative change in total energy, ΔE
Time, t

(f)

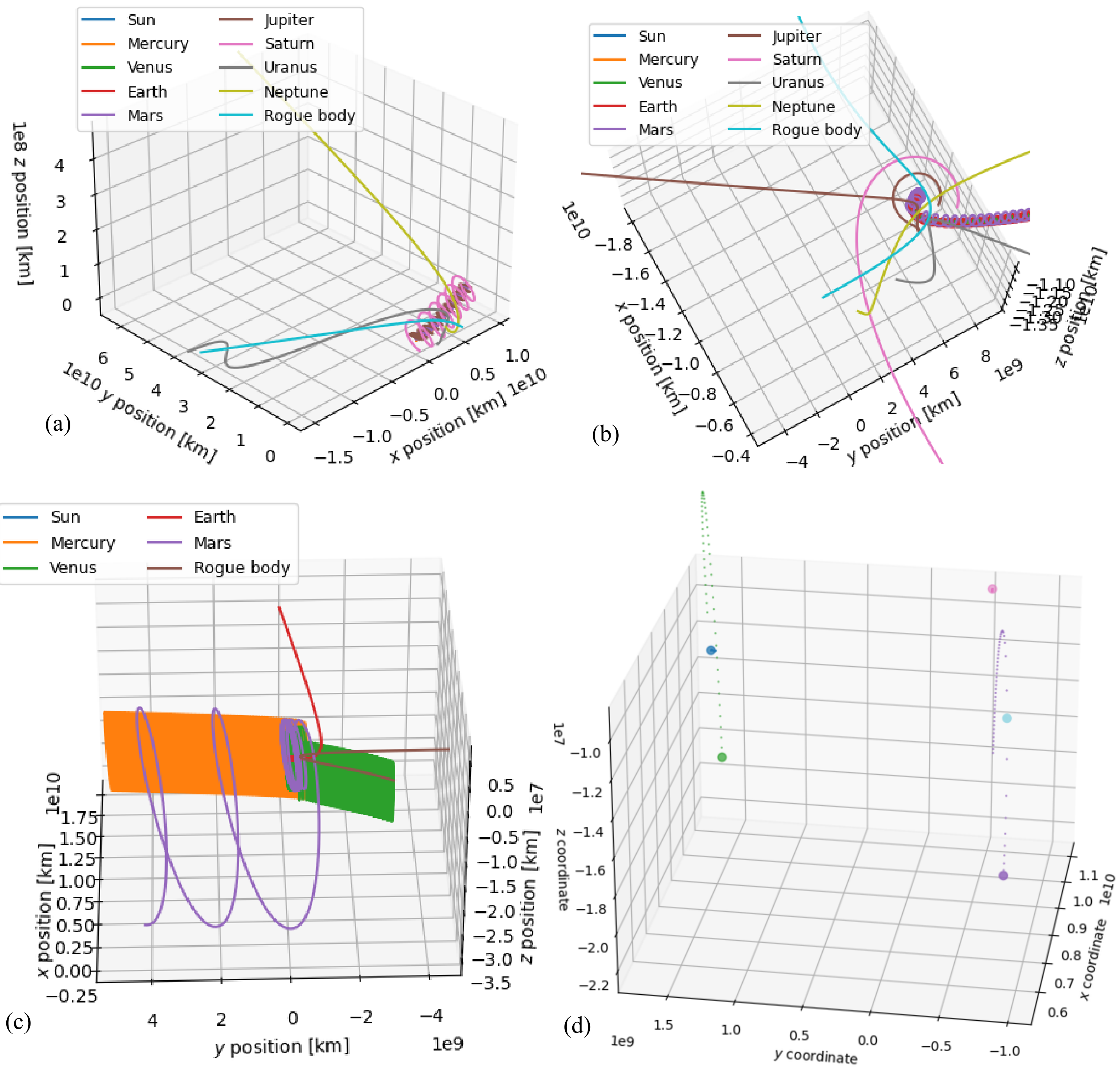


Figure 8: Close encounters with rogue stars. These figures were generated using solar_system.py, with the first two including all the planets (i.e. the "bodies_to_omit" section of code was commented out"), and the second two featuring only the inner solar system (bodies 5-8 omitted). The first two use tolerances of 1e-4, while the c & d use 1e-6. In all cases, the rogue star has a mass of 0.5 solar masses and a velocity of 10km/s in the +y direction, and an initial y coordinate of 30AU. The only initial coodinate which varies between the simulations is the initial x coordinate, which would give the distance of closest approach to the sun if gravity were switched off. For 6a this is 30AU, for 6b this is 10 AU and for 6c&d this is 5AU . Figure 6c & 6d show the same simulation, but d is a screenshot of the animation generated.

From these results, we can see that if a rogue star were to pass through the outer solar system (as in Figures 6a and 6b), Earth would go largely unaffected but some of the outer planets would be kicked out of the solar system. If a rouge star were to pass within the vicinity of the inner solar system however (such as in Figure 6c), Earth would be in much greater danger – as indicated by its ejection in the figures. Interestingly, in this particular simulation the rest of the inner solar system remains in orbit around one body or another, with Mercury and Mars gaining highly inclined orbits and Venus being captured by the rogue star.