Tom Robbins (tr1286)

Computational Physics Exercise 4 Report

The code for all three problems is essentially the same. By splitting the problem into four modules, we obtain powerful multifunctional code whereby each module can be swapped out to alter the behaviour without having to rewrite anything. The four modules are as follows:

The iterator: Simply takes the starting values, copies them into two buffers, previous and next. These are then passed into the interface module which is specified by a function pointer in one of the iterator's arguments. On each iteration, the iterator asks the interface module whether the terminating condition has been reached. If it has not, it swaps the buffers and writes the previous set of values to the specified file. Otherwise it terminates. This is virtually the exact same function as used in the previous exercise for Gauss' method, only with a couple of bug fixes.

The interface: This is the function which simply calls whichever solving method is required and passes into it a function pointer to the system of equations' function set. It is also responsible for checking the terminating condition, which is returned to the iterator as either 0 to continue or 1 to stop. Other useful variables can also be calculated here and added to the variable buffer, such as in problem 1, where the total energy is calculated.

The function set: A single function representing all of the different functions in the system of differential equations. It takes as its argument a temporary buffer with every variable in the system already transformed by the solution-method function. The other argument specifies the function reference. It returns the computed value.

The solution-method function: This is the implementation of the solving method required. It has a function pointer pointing to the function set as one of its arguments, as well as 'in' and 'out' variable buffers. All problems solved in this exercise use the exact same solution-method function, which is an implementation of the 4^{th} order Runge-Kutta method for any number of variables and constants.

Different problems can be solved simply by swapping out modules. In the case of all the problems in this exercise, only the function set and interface modules had to be written. The rest of the code works as-is. There are many benefits to this, and the result is a suite of functions which can be reused bit-for-bit to solve a wide variety of problems.

For efficiency reasons, all of the variables and constants for any given iteration, including temporary variables for passing into the solution-method function, are allocated into the same buffer so that malloc need only be called once before starting a simulation and then freed at the end.

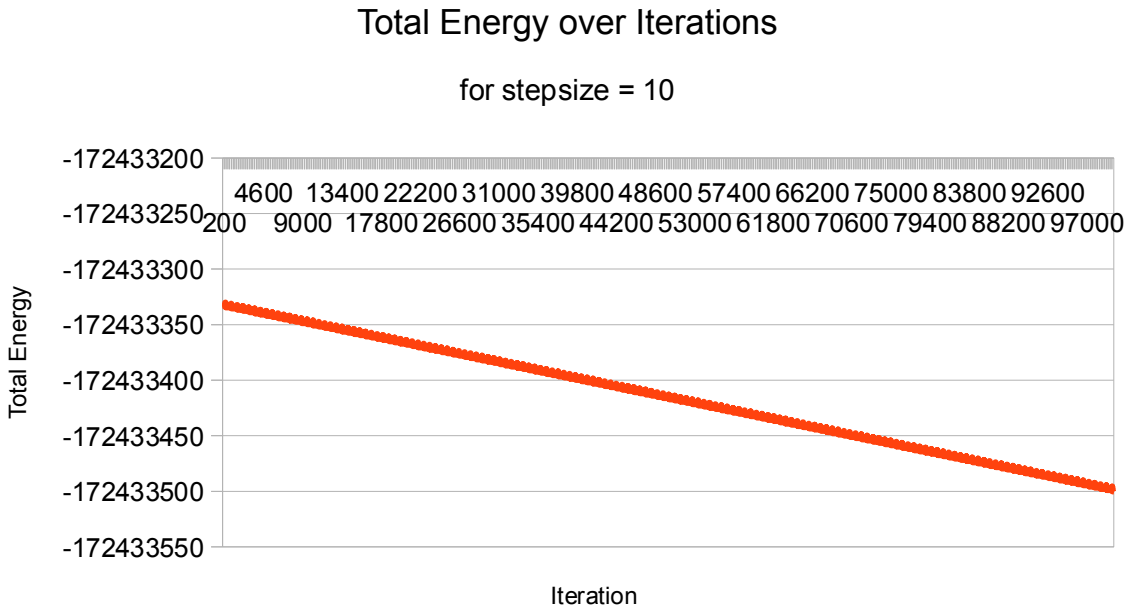
Problem 1:

Problem 1 involved implementing a simple system with only 4 variables and 1 constant. It can be run as follows:

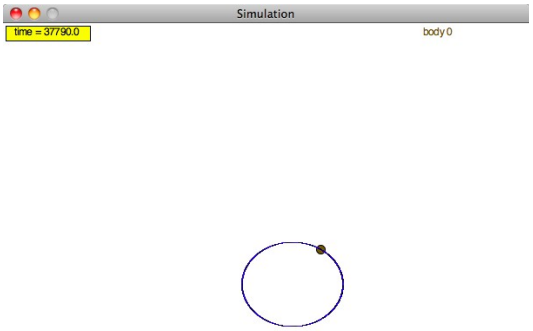
./simulator ./simpleout.csv --simple --orbit --2D timestart: 0 object position: 3E7, 0 object velocity: 0, 10E3 body mass: 1E26 timeend: 1E6 stepsize: 10

Note that the annotations, such as 'timestart:', 'object position:' are not required and are completely ignored by the program. Only the ordering of the numbers is important. However, including them

makes it much easier to work out which number is which. Units could also be included. Examining ./simpleout.csv, we see a steady reduction in the total energy of the system:

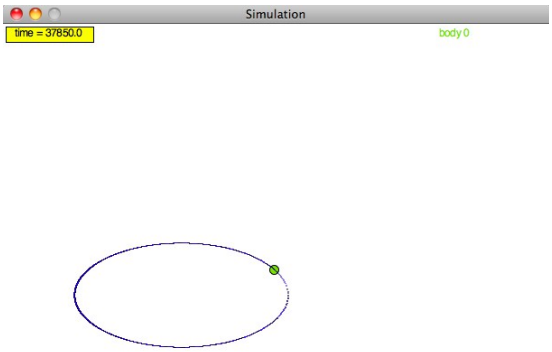


The visualised orbit looks like this:



Eccentric orbits can also be produced:

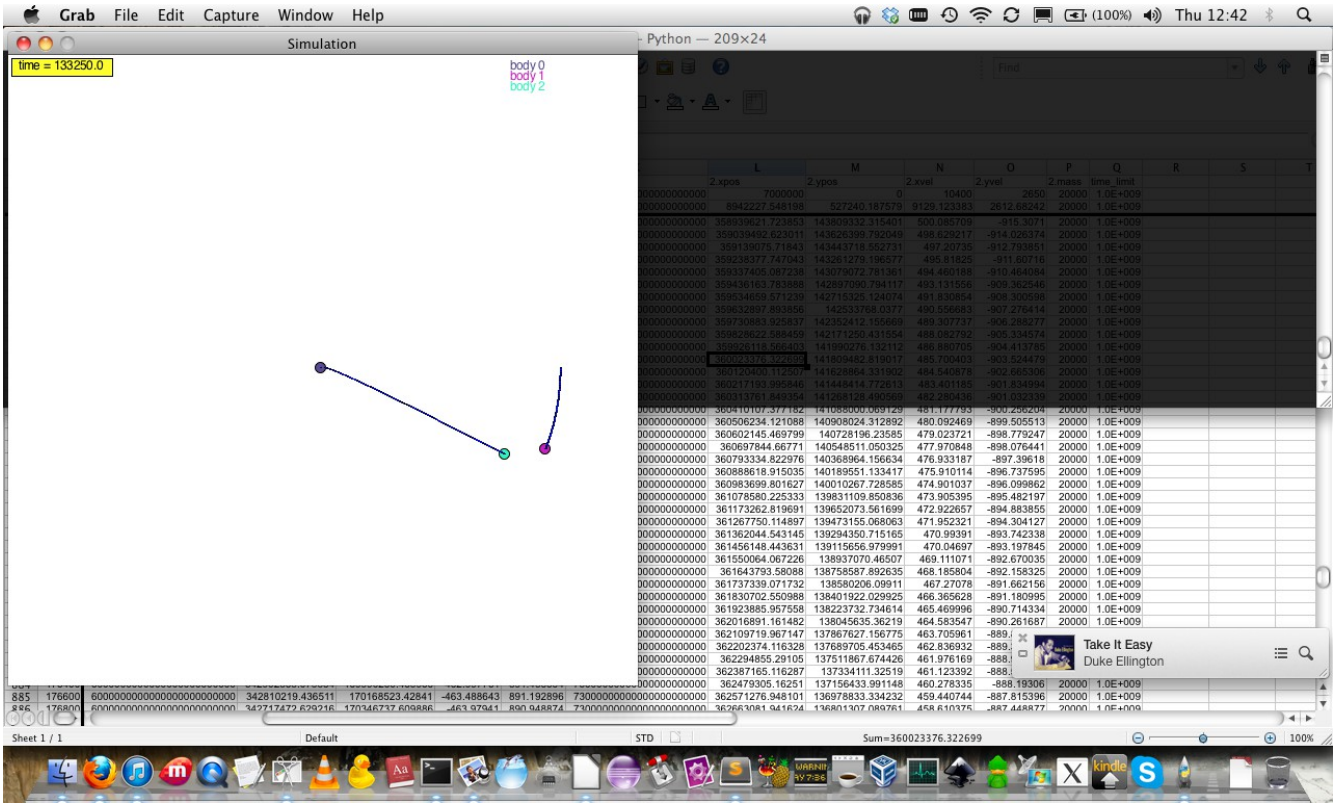
./simulator --stdout --simple --orbit --2D timestart: 0 object position: 0.5E7, 0 object velocity: 0, 50E3 body mass: 1E26 timeend: 1E6 stepsize: 10 | ./visual.py 2E8

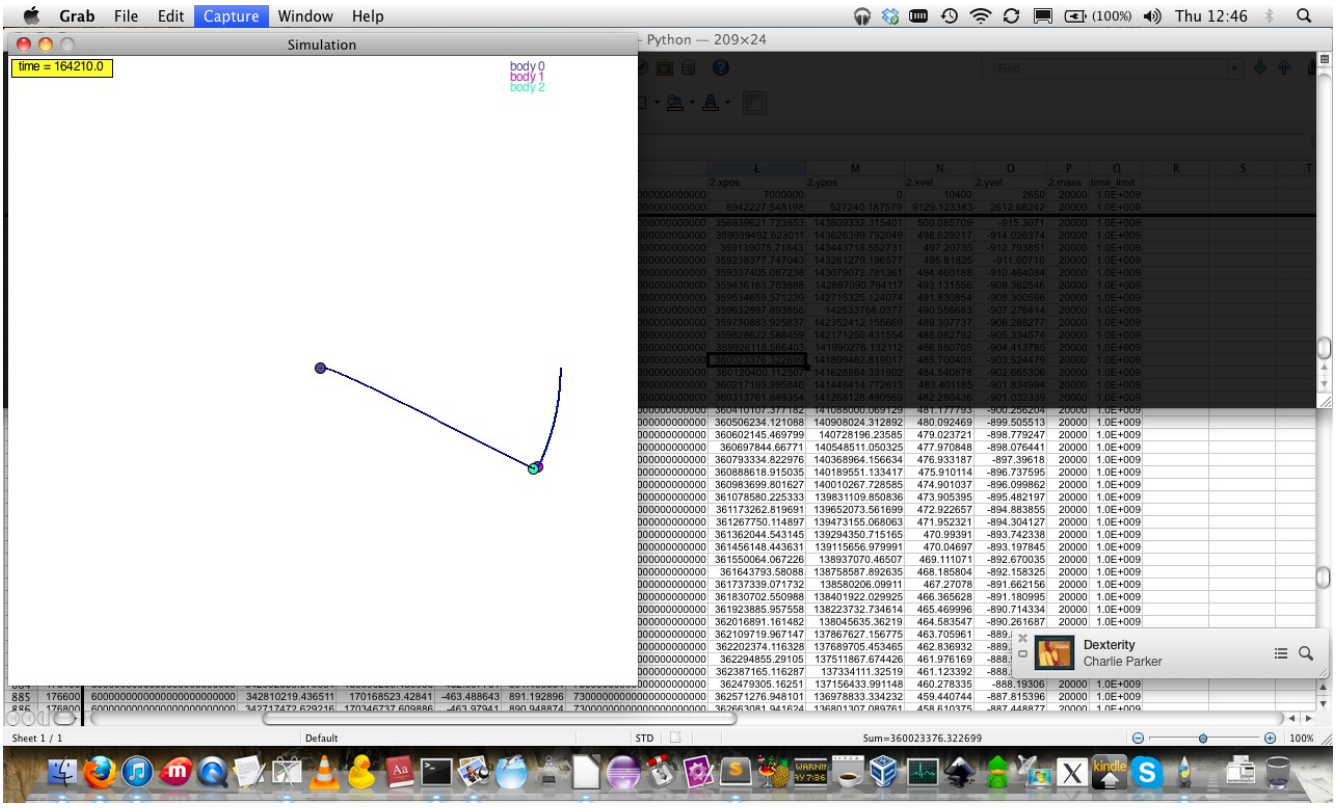


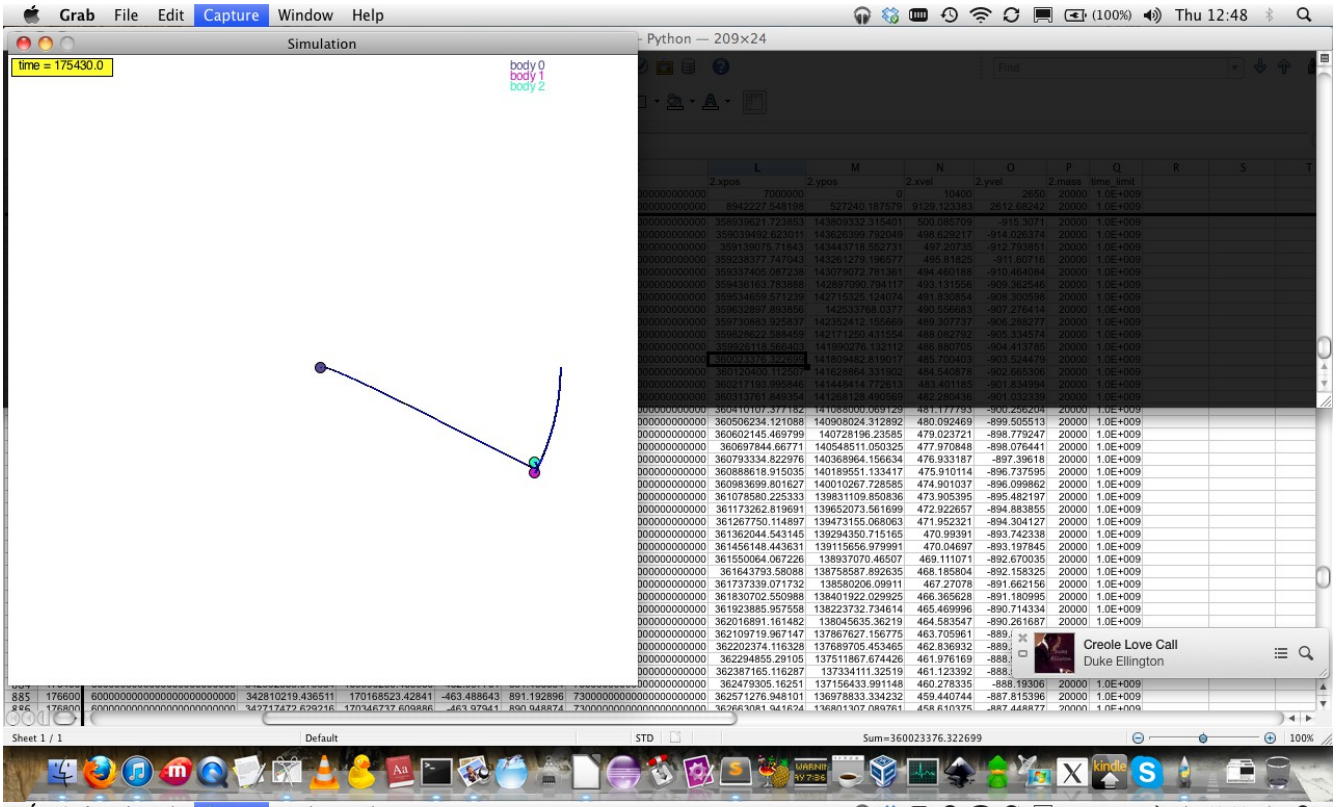
Problem 2 was solved as a 3 free-body problem. Using data obtained relating to the moon's orbital distance and average velocity, a successful moon shot was achieved by the following command:

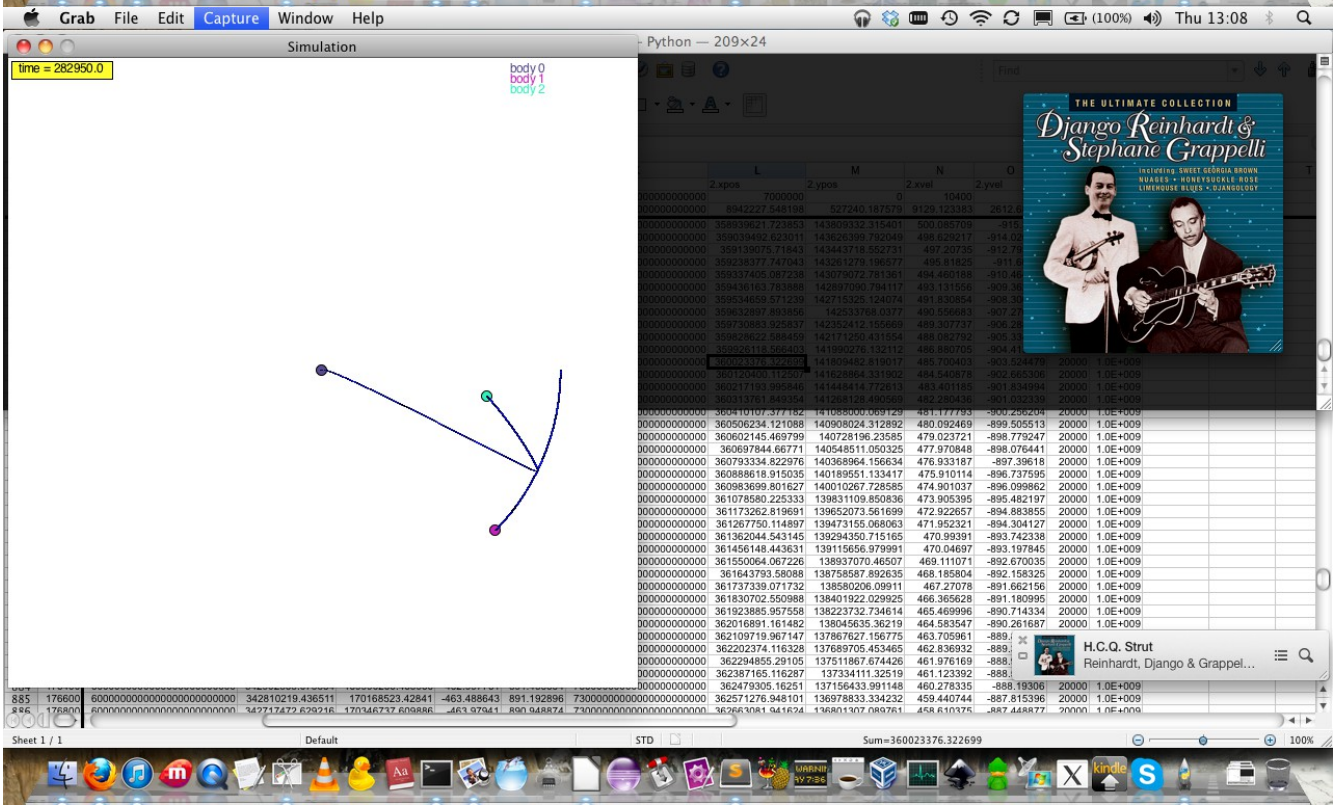
./simulator --2D --free --orbit timestart: 0 earth: pos: 0, 0 vel: 0, 0 mass: 6E24 moon: pos: 3.84E8, 0, vel: 0, 1E3 mass: 7.3E22 probe: pos: 7E6, 0, vel: 1.04E4, 2.75E3 mass: 2E4 timelimit: 1E9 step:1

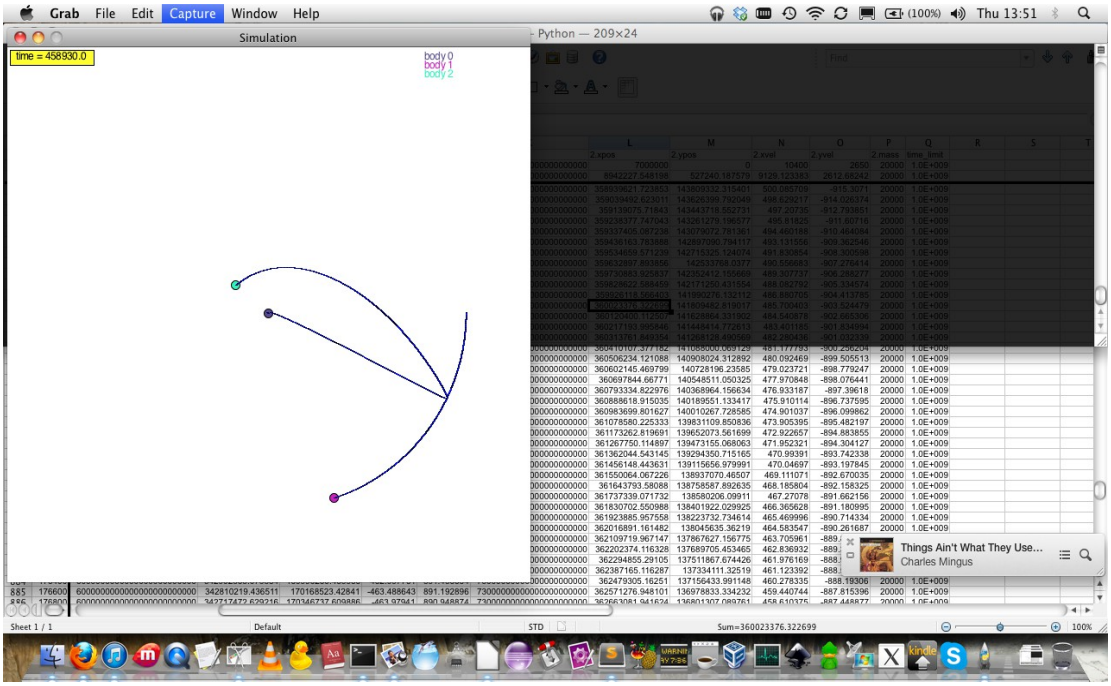
Crucially, the initial kick once the probe is in orbit around the Earth was 1.04E4m/s towards the moon and 2.75E3m/s perpendicular to the line connecting the Earth and the moon. Some screen caps from the simulation:


Grab File Edit Capture Window Help
Thu 12:42
Simulation
time = 133250.0
body 0
body 1
body 2
Python — 209×24
Take It Easy
Duke Ellington
Sheet 1 / 1
Default
STD
Sum=360023376.322699
100%


Grab File Edit Capture Window Help
Thu 12:46
Simulation
time = 164210.0
body 0
body 1
body 2
Python — 209×24
Dexterity
Charlie Parker
Sheet 1 / 1
Default
STD
Sum=360023376.322699
100%


Grab File Edit Capture Window Help
Thu 12:48
Simulation
time = 175430.0
body 0
body 1
body 2
Python — 209×24
Creole Love Call
Duke Ellington
Sheet 1 / 1
Default
STD
Sum=360023376.322699
100%


Grab File Edit Capture Window Help
Thu 13:08
Simulation
time = 282950.0
body 0
body 1
body 2
Python — 209×24
THE ULTIMATE COLLECTION
Django Reinhardt & Stephane Grappelli
H.C.Q. Strut
Reinhardt, Django & Grappel...
Sheet 1 / 1
Default
STD
Sum=360023376.322699
100%



As this was simulated by 3 free bodies, this also satisfies the premise for problem 3.

The program was easily extended to work with 3 dimensions. Here is a screenshot of one such simulation drawn by a top-down 2D projection:

./simulator --stdout --3D --free --orbit timestart: 0 earth: pos 0, 0, 0 vel 0, 0, 0 mass 5.97E24 moon: pos 0, .5E8 -3.84E8 vel 1E3, 0, 0 7.3477E22 timelimit: 5E6 600 | ./visual.py 4E9 ./myout.csv