ASEN 4057 - Assignment 6

Aubrey Mckelvy, Connor Ott

I. Assignment Report

The motivating design process was to re-purpose the MATLAB code developed for Assignment 2 such that it could be implemented in C. This means that much of the process flow remains the same. There is a main function which hosts the call of the optimize function. The optimize function uses a grid search which seeks out optimal solutions and narrows its search about found solutions. The solutions are generated by calling the integration function repeatedly with varying increases in velocity in varying directions. The Integrator subsequently calls a function which computes the rates of change of position and velocity at a given state of the system. For this particular program, the main function is "threeBody" which calls "guessOpt", the optimizing function, which calls "rungeKutta", the integrating function, which calls "yPrimeFunc", which computes the rates of change. Note that, though the name "rungeKutta" may imply a 4th-order Runge-Kutta integration method, rungeKutta employs a 1st-order Runge-Kutta method, which is identical to Euler's method.

The optimizing function "guessOpt" is called in a while loop restricted by the user-specified accuracy. As guessOpt iterates and narrows its search, its search window decreases while resolution increases. Once the resolution of the grid (that is, the distance between velocity guesses) is less than that of the user specified accuracy, the solution is guaranteed to satisfy the accuracy requirement, and the while loop exits.

Within MATLAB, it is convenient to have each function saved as an independent file. This is not the case for C. Instead, all the functions used by "threeBody" are defined in one tool box header file, "tblib.h", which is included at the beginning of "threeBody". This is nice because it combines the utility of modularized code with the convenience of having fewer files to drag around. This turns at least four files into two, "threeBody.c" and "tblib.h", which are used to create the compiled program "threeBody".

A flow chart for the components of the program is given below.

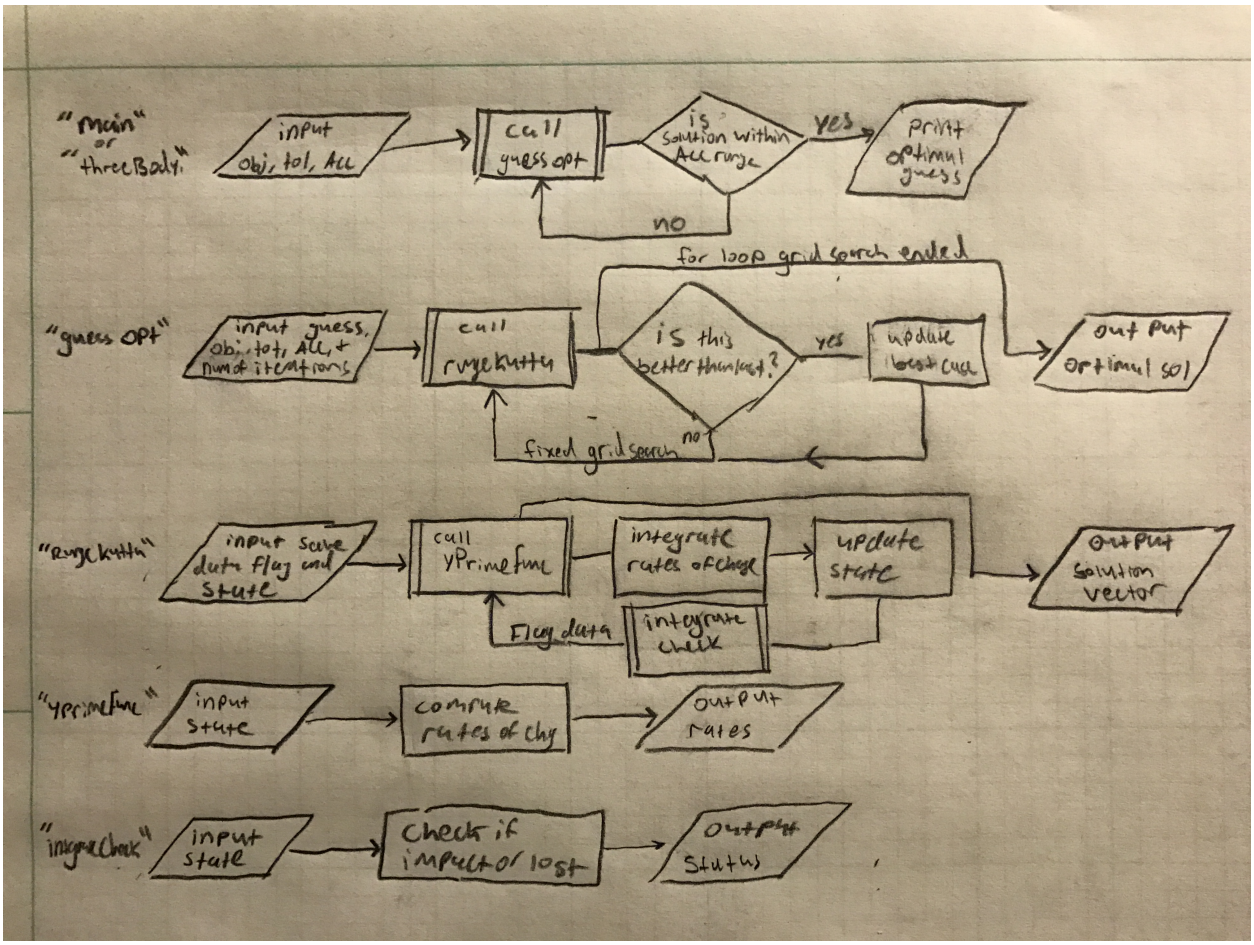

Figure 1: Code Flow chart.

II. Part 2 - Solution Plots

The program executable, threeBody, should be executed as follows:

```
./threeBody Obj Clearance Accuracy
```

Where Obj is either a 1 or a 2, corresponding to the optimization the user wishes to perform. Clearance is a distance in meters specifying how close to the Moon's surface the spacecraft is allowed to pass. Accuracy is an absolute accuracy in m/s which specifies the maximum error between the actual and output solutions.

The figures below show the solutions found when optimizing for objectives 1 and 2 with clearances of 10,000 m and accuracies of 0.5 m/s.

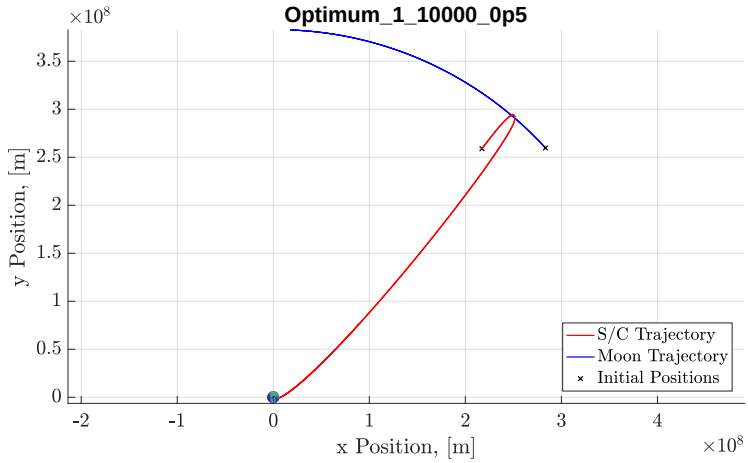

Figure 2: Objective 1 with clearance of 10,000 m and 0.5 m/s accuracy.

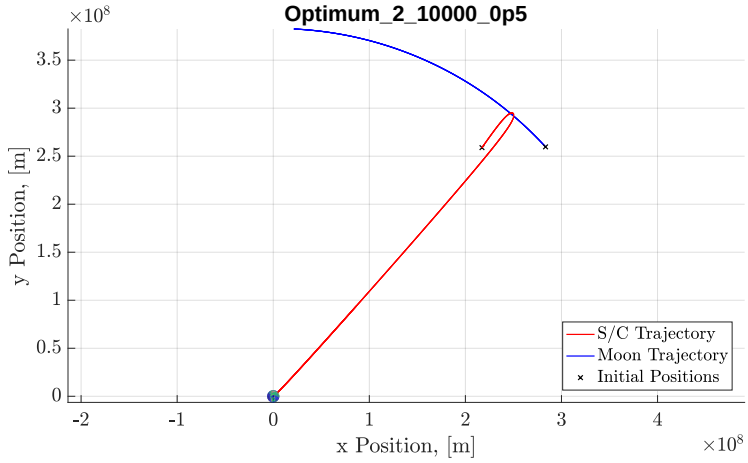

Figure 3: Objective 2 with clearance of 10,000 m and 0.5 m/s accuracy.

The following are the solutions determined by the optimization processes.

Table 1: Optimization Results

Objective	Velocity Magnitude [m/s]	Angle From +x Axis [°]	Time to Return [days]
1	79.25	90.05	3.430
2	92.74	122.15	3.387

III. Part 3 - Bash Script

In order to test the program's ability to accept user input and work properly, the group developed a bash script titled "TaskList.sh" whose purpose is to execute the function for both objectives 1 and 2 at clearances ranging from 0 to 100,00 m with 0.5 m/s accuracy. For each solution, a text file was created and the path of the moon and spacecraft as functions of time were written and saved. Note that since the motion of the Earth was assumed to be negligible for this simulation, the written files do not include the Earth's position as a function of a time, as it would only read 0s for the entire time span. The plots for each of these solutions can be found in the Appendix. The written files can be found in the submission directories of Assignment_6 in both the /Ott and /Mckelvy directories.

IV. Profile Report

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
44.75	0.21	0.21	588	0.36	0.78	rungeKutta
44.75	0.42	0.21	9071742	0.00	0.00	yPrimeFunc
8.52	0.46	0.04	9071742	0.00	0.00	integrateCheck
2.13	0.47	0.01				frame_dummy
0.00	0.47	0.00	12	0.00	38.39	guessOpt

Figure 4: Gprof's flat profile output.

Figure 4 shows the flat profile of the code outputted by gprof. The right most column lists the function names and the preceding columns give information regarding a functions time per call for both the function alone and the function with its sub-functions, the number of calls for each function, the number of seconds the function took in total, the cumulative run time for each function, and the percentage of the total run time occupied by that function.

The rows are listed such that the function with the largest time consumption is listed first and the lowest time consumption listed last. This allows a quick understanding of which functions are the most computationally expensive. In this case, the "rungeKutta" function and the "yPrimeFunc" tie for most expensive at 0.21 seconds each. Together they make

up 89.5 percent of the total runtime. This is because these functions are responsible for integrating the rates of change and computing the rates of change respectively; these are the primary computations required to solve the three body problem and the only computations which need to be applied for every single time step. The calls column shows 588 calls for the integrator and 9,071,742 calls for both the rate of change computer and the state check function "integrateCheck". "rungeKutta" calls both of these functions once, which means it calls each of them approximately 15,500 times each time it runs. Despite the fact that "yPrimeFunc" and "integrateCheck" are called the same number of times, "integrateCheck" takes only 0.04 seconds compared to "yPrimeFunc"'s 0.21 seconds. This is again because "yPrimeFunc" is performing a computation and "integrateCheck" is just checking the state of the system, which requires only a set of conditionals and no actual computation. The least expensive function is the "guessOpt" function which is hosted in the main "threeBody" function and is the parent to all the other listed functions. This is on the bottom because it performs practically no operations aside from calling "rungeKutta" and using the outputs to adjust its input for the next call. This is reflected in the time per call of the "guessOpt" function, which is nearly zero seconds for just the function itself, but 38.39 milliseconds when including the time spent within its children. Its also worth noting that "rungeKutta" and "integrateCheck" take nearly no time per call, but when they are called nearly 10 million times each, that nearly adds up to 0.21 and 0.04 seconds respectively.

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 2.12% of 0.47 seconds

index % time    self  children    called     name
                0.21    0.25     588/588         guessOpt [2]
[1]     97.9    0.21    0.25     588         rungeKutta [1]
                0.21    0.00 9071742/9071742     yPrimeFunc [4]
                0.04    0.00 9071742/9071742     integrateCheck [5]
-----------------------------------------------
                0.00    0.46      12/12          main [3]
[2]     97.9    0.00    0.46      12         guessOpt [2]
                0.21    0.25     588/588         rungeKutta [1]
-----------------------------------------------
                                                 <spontaneous>
[3]     97.9    0.00    0.46                 main [3]
                0.00    0.46      12/12          guessOpt [2]
-----------------------------------------------
                0.21    0.00 9071742/9071742     rungeKutta [1]
[4]     44.7    0.21    0.00 9071742         yPrimeFunc [4]
-----------------------------------------------
                0.04    0.00 9071742/9071742     rungeKutta [1]
[5]      8.5    0.04    0.00 9071742         integrateCheck [5]
-----------------------------------------------
                                                 <spontaneous>
[6]      2.1    0.01    0.00                 frame_dummy [6]
-----------------------------------------------
```

Figure 5: Gprof's call graph output.

Figure 5 shows the call graph generated by gprof. In this table, each grand row is devoted to one of the 6 functions called during the profiling. The functions listed above the index number are that functions parents and the functions below are the functions children. Each column gives information similar to that given in the flat profile; the index column lists the index number of the function being analyzed; the percent time column lists the percent of the total run time occupied by the function and its children; the self column lists the time that was used by the function, the time that was propagated to the functions parent, and the time that the children propagated to their direct parent; the children column lists the time propagated to the function by its children; and the called column lists the number of times the function was called, the number of times the parent function called the function over the total number of times the function was called, and the number of times the children were called by the function over the total number of times the child was called.

Again, the rows are organized from the most expensive on top and the least expensive on bottom. Now the percent of total runtime spent within each function can be analyzed. "rungeKutta", "guessOpt", and "main" tie for first place, each having 97.9 percent of the runtime spent within them. The reason for this three way tie is that each of these three

functions are the first three functions called and only host and organize data rather than performing computations. Working from the bottom up will illustrate the function hierarchy. The least expensive function is "frame dummy". This function is used by the profiler and is not actually a function used by "threeBody". Next is the "integrateCheck" function. This function has no children and it propogates no time to its parent function. This function spent 0.04 seconds and hosted 8.5 percent of the process. Third from the bottom is "yPrimeFunc", which also has no children and propagates no time to its parent. It spends 0.21 seconds and hosts 44.7 percent of the runtime. Next is "main" or "threeBody", which has one child and no parents. It spends nearly zero seconds, but hosts 97.9 percent of the process. Its child propagates 0.46 seconds into this function. "guessOpt" is the child of "main" and is responsible for pushing those 0.46 seconds by way of its child "rungeKutta". "guessOpt" spends nearly no time but receives 0.46 seconds from its child. Finally, "rungeKutta", which is the child of "guessOpt" and the parent of "yPrimeFunc" and "integrateCheck". It receives 0.25 seconds from its children and spends 0.21 seconds itself.

V. Appendix

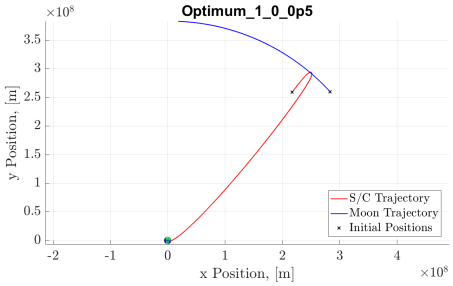

Figure 6: Solution for optimization of objective 1.

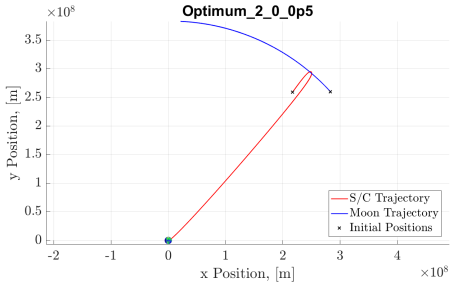

Figure 7: Solution for optimization of objective 2.

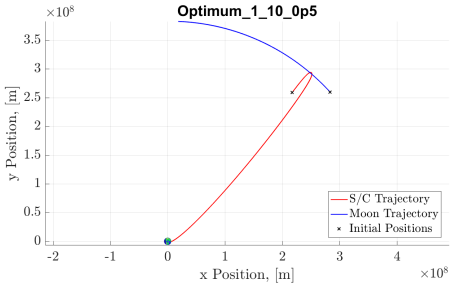

Figure 8: Solution for optimization of objective 1.

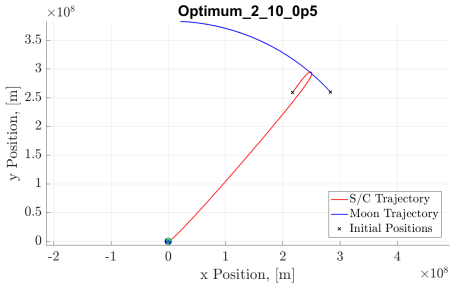

Figure 9: Solution for optimization of objective 2.

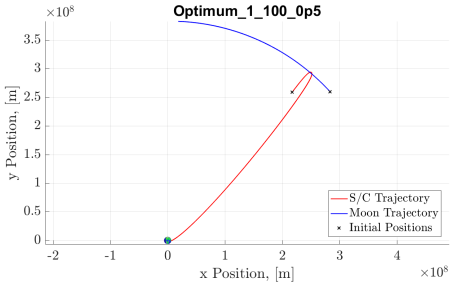

Figure 10: Solution for optimization of objective 1.

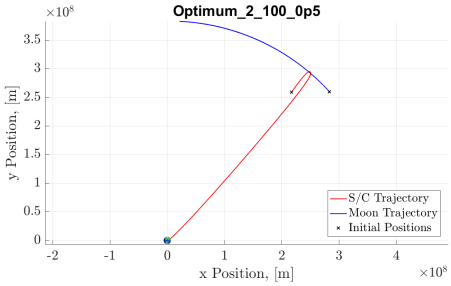

Figure 11: Solution for optimization of objective 2.

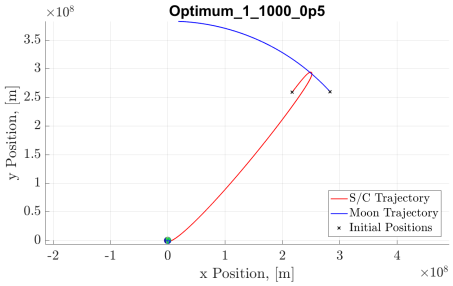

Figure 12: Solution for optimization of objective 1.

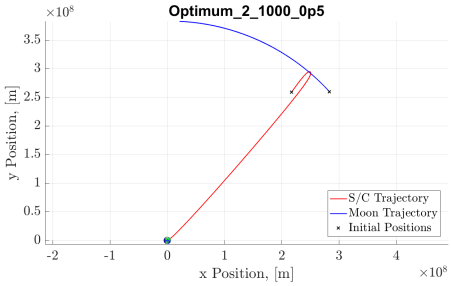

Figure 13: Solution for optimization of objective 2.

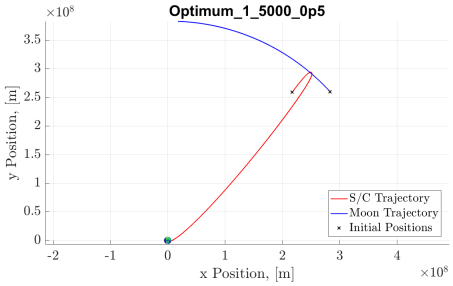

Figure 14: Solution for optimization of objective 1.

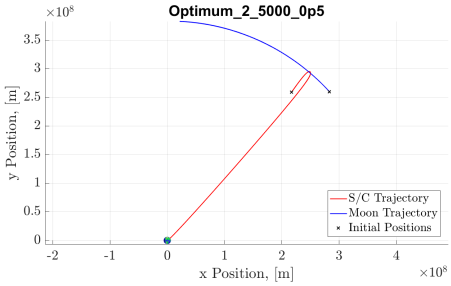

Figure 15: Solution for optimization of objective 2.

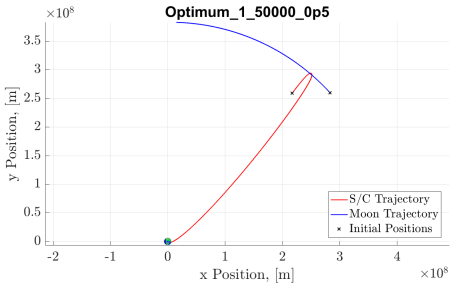

Figure 16: Solution for optimization of objective 1.

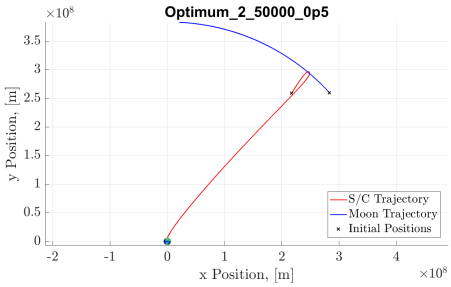

Figure 17: Solution for optimization of objective 2.

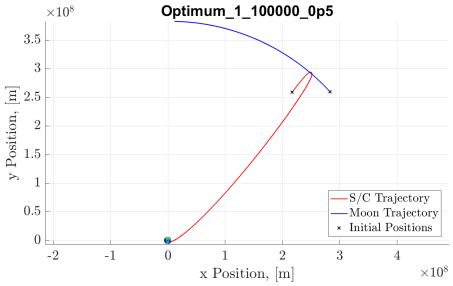

Figure 18: Solution for optimization of objective 1.

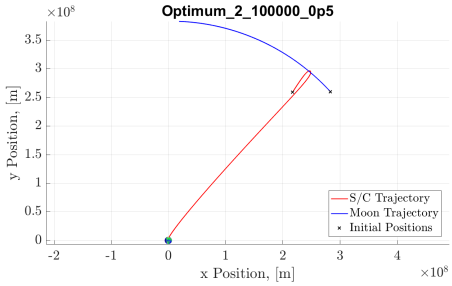

Figure 19: Solution for optimization of objective 2.