Design Process

The design process for our final code had a few key components. Rather than making a constant struct, all constants were declared as global doubles. The next step in being called is reading the user inputs and determining which objective function to run. Previously, the same objective function could have been run for both cases with a few changed options, but for the sake of speed, these functions were split up. C does not have “try; except” functionality built in like other languages so the different checks to speed up the code would have required a complete redesign. For this reason objective1 and objective2 are separate.

Once major design change from the MATLAB code was designing a way to change values without being able to return more than one variable per function. Inside of the euler function, the structures containing the state spaces of each object are changed inside of the loop rather than returned. Also the force function is run for each direction (x,y) rather than return a force vector.

There were a few methods for speeding up the code. For each objective, no net velocity change>100m/s was run. For the first objective, once a solution has been found, if the new net velocity is greater than that solution, the new velocity is also skipped. For the second objective, a similar check was implemented. Once a solution was found, if any simulation ran past the previous solution time, the loop was stopped and the next velocity was considered. This allowed the first objective to be solved in approximately 3.5min as opposed to 1hr and the second objective to be solved in 8 min rather than 1hr.

There weren’t many options for creating the files names. C has incredibly limited functionality when it comes to strings, treating each string as an array of characters. Creating the file names for each output took 25 line while in comparison, creating the file and writing the entire trajectory for all 3 objects to them took 4.

Profiler

The use of a profiler allows us to examine how often a specific function ran and how long each one ran. This was studied in an earlier assignment using Matlab, except now we will be using the GNU C profiler, specifically, the function named *gprof* to time our code compiles and successful runs. The profiler in C is implemented just as easily as it could be in Matlab except now the script must be compiled with the “-pg”, the compiled function must be executed, then the profile viewer can be called using “gprof *function_name*”. Once this runs, the profiler outputs a flat profile and a call graph. The flat profile looks almost identical to what we see in Matlab: percentage of time ran compared to total time, cumulative time adding up each called functions self time, self time, number of calls, self seconds per call, the total seconds per call and the name of the function itself. Explanations of each of these titles can be seen slightly below the actual profile. Next is the call graph which can be seen slightly below the flat profile. The call graph essentially shows a small table profile for each function. Each function called is given an index where it is then compared to every other function. This comparison allows us to see how each function called other functions. The first column shows the index, the next shows what percentage of time of the total time was utilized within that function, next how much time the function itself ran, next how much time children function within the function called ran, then finally the next column shows how many times a function was called within each function. If there is only a single number, that represents the actual index of the profiled function itself, therefore that should represent the total number of times the function was called. If the number is represented as a fraction, the denominator represents the total number of times the function was called, and the numerator represents the number of times a function was called within the specified function index.