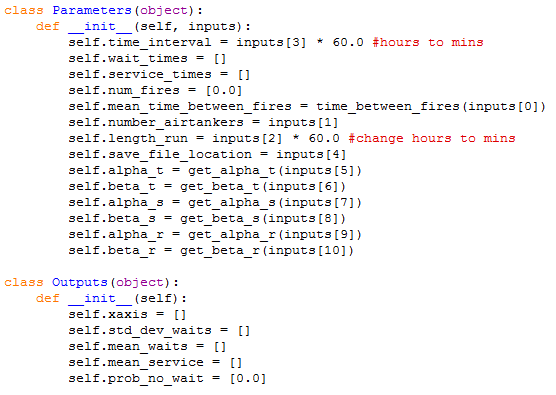
Erlang Queueing Model

This Erlang Queueing Model simulates fires occurring over a long period of time, and how airtankers handle these Fires. It then graphs the results over time.

Classes

First we will start off with the two classes used in this simulation.

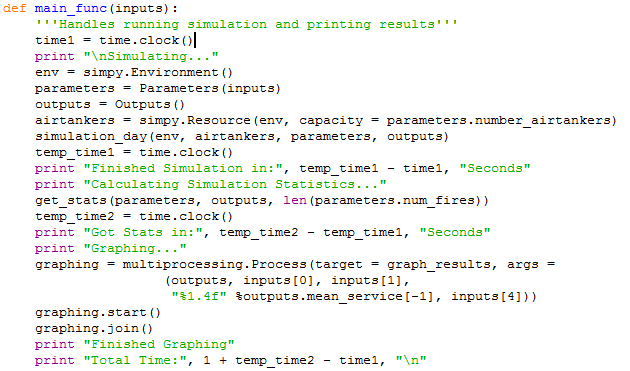


The parameters class is used to store all of the parameters for running the simulation. These are all of the values that the user would enter for control how the simulation runs. This class also stores the intermediate values in the wait\_times, service\_times, and num\_fires lists.

The Outputs class on the other hand is used to store all of the statistics that we care about tracking. These are all of the lists that we will use at the end of the simulation to generate our graphs of the simulation results.

Functions

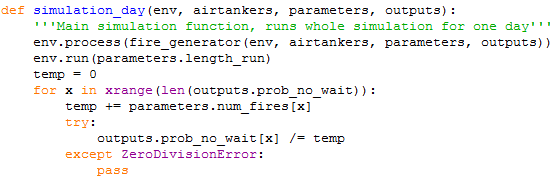
First, we start off with our main function that is responsible for calling all of our other functions.



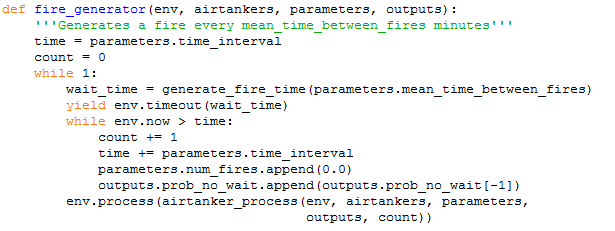
\*All of the time.clock() statements can be ignored, they are just so the simulation times how long different parts take to complete. The print statements can also be ignored, they also are just to provide the user feedback that the program is running properly.

First the function creates a Simpy environment that will keep track of the time and handling of processes in this simulation. We then create a Parameters class based on the user given values for all of the parameters, as well as a default output class that we can use to save our results too. Next, the airtankers line creates a simpy resource for the number of airtankers we have. The capacity is the number of things that can use this resource at the same time and therefore in this context refers to the number of airtankers there are available to fight the fires. Now that all of these have been created, we can run the simulation!

I will go over the function of this simulation\_day function.



This function starts off by creating a simpy process that calls the fire\_generator function. The simulation is then run for a very long time based on user input. Finally, the percentage of fires in each timestep of the simulation that had no wait time is updated and saved to our ouput class for graphing later on.

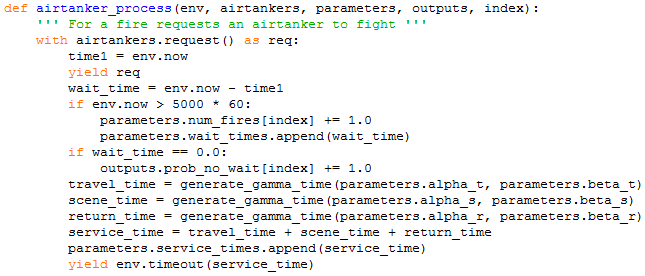


The main process run by simpy is this fire\_generator function. It’s mainly responsible for creating fires randomly, and then calling a different function to try and fight this fire.

The time is used to keep track of the time of the next time interval for the whole duration of the simulation. The count variable keeps track of which time interval we are in. Since we are using this with simpy, we cause the function to repeat as an infinite loop. This means the function will keep running for the whole simulation length of time that the user specified. First this function calls the generate\_fire\_time function to get a wait time until the next fire starts.

C:\Users\Cam\Documents\Airtanker Simulation Model\Airtanker Queueing Model - B\Detailed Report\generate_fire_time.png

This function just currently generates fire times based on an exponential distribution with a user inputted mean time between fires. Next the fire\_generator function yields for this amount of time in the simulation environment. The next while loop simply checks if the new simulation time is in the next time interval (or a few intervals later), and if it is then the loop keeps advancing the time interval until it is in a correct time interval. Next, for this new fire created, a new process is called to try and fight this fire.



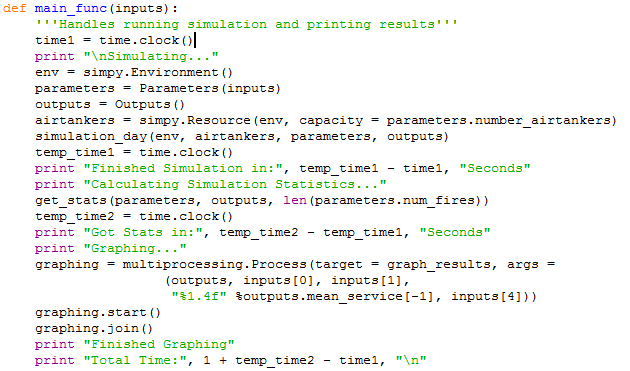
This airtanker\_process function is responsible for requesting an airtanker to fight a fire and handling this fighting of the fire.

The parameters to this function are the environment (so we can get the current time), the airtankers resources (so we can request their use), our input parameters (for the values to use when calculating service times), the outputs class for saving the no\_wait probablility to and the index of which time interval this fire is in.

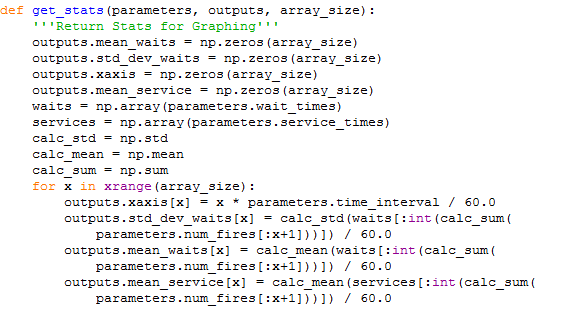
The first line puts in a request for one of the airtankers we have. We then take the current simulation time, and then yield until the request is granted (the request is granted when there is an airtanker available to fight this fire). The new simulation time is then recorded, and the wait time is defined as the amount of time the fire had to wait for an airtanker to become available to fight it. If there was an airtanker that wasn’t already busy, this time will be 0. The next if statement just makes this function only record data from fires after the first 5000 hours of the simulation run time. If the wait time is 0.0 then it is recorded.

Next the function attempts to calculate the total service time of the fire. It does so by summing the calculated travel time, on-scene time and return time of the airtanker. Each of these values is randomly calculated based on an erlang distribution with user entered parameters. The two required parameters are a shape parameter Alpha (that must be an integer) and a rate parameter Beta that is the inverse of the scale parameter Theta. The function then yields for the service time meaning the resource is unavailable to other fires during this time in the simulation.

This concludes the steps of the simulation part of this model. We now continue on with our main function.

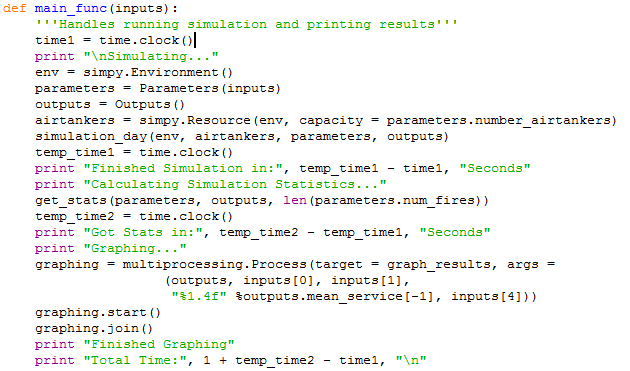


The next function that is called is the get\_stats function which is responsible for organizing the simulation results into meaningful data we can later graph.

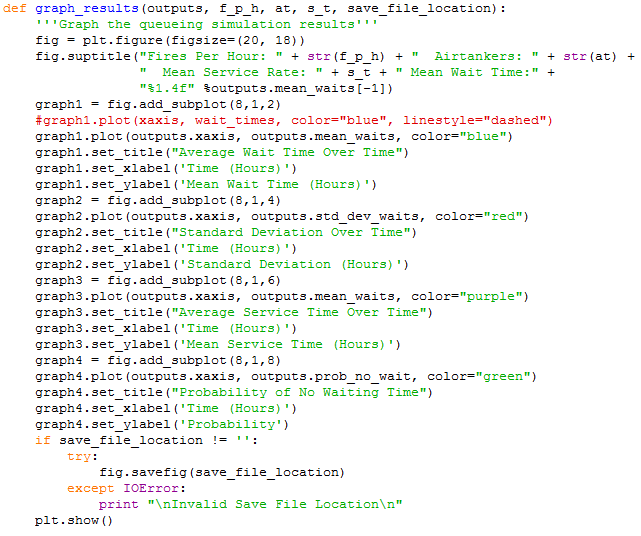


This function uses numpy to do all of the numerical work for improved performance (and potentially accuracy).

The first four lines create empty numpy arrays of 0s all with a number of elements equal to the number of time intervals that were in the simulation. Next, we create a numpy arrays from the wait\_time and service time lists that we intermediately calculated in the simulation, these arrays store our data. Finally the next 3 lines are simply to improve performance (so they can be ignored). We then loop through the number of time intervals we have. In this loop we calculate a value for the xaxis that is just integer multiples of the time interval used, the standard deviation of wait time up until that point in time, and the mean service and wait times up until that point in the simulation. This means that for these calulations they all calculate using all of the previous values, not just the new time intervals values. This finishes organizing the stats so we can now graph them.



The final step is to call the graph\_results function (ignore the multiprocessing stuff, it is to fix a bug when incorporating the GUI).



This is the function to graph the results. I use the matplotlib library for this. I won’t go into too much detail since it is not that important to understand. Basically I just create a figure and then add a subplot to with the x axis being the axis array stored in outputs, and the y axis being the other area stored in outputs (one for each average wait time, standard deviation of wait time, average service times, and the probability of there being no wait time. Aside from this, the other statements just add labels to the graphs and then at the end we finally show the graphs.