For the UAV Controller script, I started with designing the main infinite loop that will be handling the starting, stopping and monitoring of processes, and abstracted methods away to simplify the visualization of the code. The processes this part of the script is monitoring are the receiver module, the timeout module and the transmitter module. This forever loop was based on a previous rendition, the “Exec\_mod\_file.py” script, but puts a lot more control into what is happening into the scripts hands. The first loop waits until either the file to be modified that the receiver is sending data to, or for the timeout module to complete. If there is a timeout situation (where the “timeout\_module” exits on its own), then the UAV will go into a “phone home” mode, where it sets its frequency, modulations scheme and timeout back to their programmed defaults. The second part of this forever loop does nothing if a timeout occurred, and if there was no timeout, which means that the file was modified and there is a command waiting to be executed, the script decodes the command and transmits the proper data back to the ground station if necessary. The rest of the methods and functions within this class are abstractions from this first constructor module, or even abstractions from abstractions and may even be simply useful methods or functions used throughout the script. The “run\_to\_module”, “run\_tx\_module” and “run\_rx\_module” methods simply put together the necessary argument stings, and run them using the “os.spawnl” command. This command can be ran with “os.P\_NOWAIT” or “os.P\_WAIT” to run the spawned processes in parallel with their parent process, or have the parent process block until the child process has exited respectively. The “pid\_exists” and “kill\_pid” functions are the next two logical steps to manage all the child processes. The “pid\_exists” allows the script to determine if a specific process exists on the system, then it determines if it is not a zombie. If the process exists, and is not a zombie, then the function returns a “True”, if the function exists, but is a zombie, this function will read the exit status of the process, and therefore killing it altogether and return a “False” and, finally, if the process just doesn’t exist on the system, it will return a “False”. This does this by tapping into the “/proc” directory of the Linux file structure. In this directory, there is a pseudo directory for each process running on the system, named by the processes pid, that contains files with statistical data and other necessary information on the running process. The “kill\_pid” function simply sends an operating system call to the given process to exit, then checks to make sure that it fully exits by calling the “pid\_exists” function, and returning the inverse of that function call. The “update\_rx\_opts” and “update\_tx\_opts” are two methods that set the option lists for the transmitter and receiver “os.spawnv” calls with the most up-to-date values of variables. The “exec\_command” function is one of the key functions. This function, which is only executed when the receiver file has been modified, initially reads off the first line of the file, then uses this to determine what to do next. There is an “if … else if” ladder to execute specific blocks of code deterministic of what command is in the file. If the command is not of any of the specified forms, then it would set the script up to tell the ground that. If this occurs three consecutive times, then the UAV would go into “phone home” state. The “settings” command will take in three more lines of data, and change some of the operational settings, the “picture” command will take a picture, the “fft” command will take in a vector of data for a FFT operation, and the “sensors” command will collect the temperature, battery and GPS data to be sent back to the ground. All of these command blocks of code need to update the modify time of the receive file so there will not be an implicit infinite loop formed, as well as reset the count for receiving erroneous data. The “comb\_misc\_data” method simply combines the individual files from the scripts that gathered the sensor data, into one file to be sent down to the ground station. Finally the “init\_vars” and “init\_files” methods simply initialize the variables used throughout the class, and the external files as to not need “try … expect” blocks for the existence of files external files.

Problems:

The main problem I had with this was dealing with zombie processes. My first challenge was to figure out how to kill a zombie process. I learned from a thread on stackoverflow.com that in order for a process to completely die, its exit status must be read. After finding out this, there was a race to figure out how to do this with code, and the answer was one simple line of code: “os.waitpid(pid, os.WNOHANG)”. The last thing that was giving me a challenge was to figure out how to properly call the “os.spawnl” and “os.spawnv” commands. For these, I found source code on the python docs that showed what all of the different parts of the parameters for the function call were, and what exactly should go into them. I had one problem with the timeout module still; by putting an integer value into the argument call from the “os.spawnl” command, and taking this value into the option parser. I ended up needing to send the timeout value as a string, with the following code: “ ‘%d’ % self.timeout\_value”, and typecast from a string to an integer within the timeout module.