ENGR 3390: Fundamentals of Robotics - Final Report

May 12, 2021

Charlie Team: Isabella Abilheira, Chris Allum, Olivia Jo Bradley, Oscar De La Garza, Nabih Estefan, Maya Sivanandan

Contents

1 Introduction

For the final FunRobo project, our team was tasked with creating a rover that could complete several missions in The O, the center courtyard of Olin's campus. The three main challenges were: Navigation of unobstructed terrain, Docking in a supply station, and Delivering a payload to a drop box. Besides this, our team decided to try and do 3 extra stretch goals: Navigate through stationary obstacles, park within the medium box of the supply station, and deliver our payload to a specific dropbox using April tags (with a possible one being finding payloads around the O and picking those up for extra deliveries). Of these, we were able to complete the first two during our demo, with the two payload stretch goals not being tried since the payload rules we're re-instated halfway through the development of the project, and stretching ourselves to complete these seemed unnecessary.

To develop our rover and tackle these challenges we divided our team into 3 initial sub-teams: mechanical design, electrical design, and control software (Knowing that close to the demo we would all eventually be working on software design). The mechanical design team was in charge of designing and fabricating the body of the rover, the electrical team chose what sensors and motors would be used on the robot and how best to wire them to avoid damage, and the software team was in charge of writing the SETUP, SENSE, THINK, and ACT functions, and the main demo code that would allow the rover to operate autonomously. Eventually, the merge we had expected occurred, and even though the software team still did most of the coding, everyone was able to help.

2 Rover: Mechanical Design

Figure 1: Final Rover Image

For the mechanical design, there were two main components to focus on fabricating and assembling: the body of the car and the grabber.

2.1 Body

As a group, we decided on creating a dragon themed rover. Our house hold group name was related to fire, so we wanted to continue that theme in our rover. To start designing our rover, we looked at the general basic design given to us by the teaching team, and modified it to match our aesthetic choices. You can see what our CAD looked like at the end of this stage in Figure 2 below. After all the pieces had been

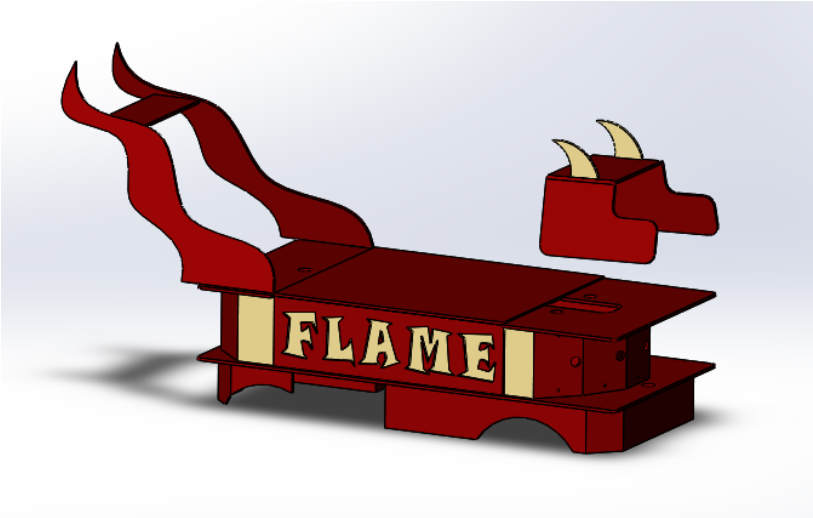

Figure 2: Final CAD

designed, we decided to create a sketch model prototype of our rover out of cardboard. This choice was made because it allowed us to learn if our dimensions were correct and fit with the RC car frame, which portions were easier to assemble, which portions we should consider 3d printing, and what holes we should add for electrical wiring. Below in Figure 3 is a photo of this sketch model prototype.

Figure 3: Rover Sketch Model

After having done this, we decided there were many things we could 3-D print. One of the big ones was mounts for our lidar array mount, pan-tilt servo, Raspberry Pi , battery mount, and PiCam. We chose to 3d

print these as they were either difficult to create with sintra (such as the lidar array or the pi cam mount) or we felt more comfortable mounting them with bolts rather than glue (such as the battery mount).

We also chose to make several ease-of-use and accessibility changes. One major change was the back cover. While the decorative sides were fun, we found that sliding it on top was a bit of a hassle, and the letters were easily bumped and bent. Instead we chose a hinged back cover that we can flip open and close easily. This allowed us to protect the electronics while it was drizzling and hold the cover in place when we were testing with hard winds. We also chose to mount many of our components with bolts. This allows us to remove panels easily to help organize wiring, swap out components, and mount things securely.

For the rest of the body panels, we chose to shopbot them for accuracy and speed. This not only saved us time, but allowed our parts to have straight and accurate dimensions. You can see a photo of the shopbot cutting our parts in Figure 4 below.

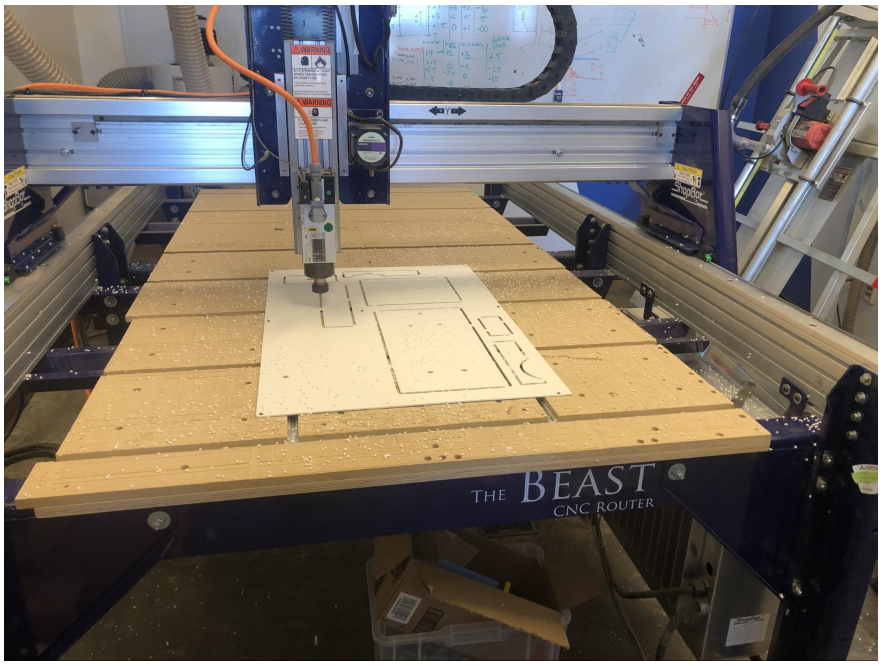

Figure 4: Shopbot cutting Parts

We assembled our rover with PVC glue and Cyanoacrylate glue in addition to bolts. We painted our rover red and gold for the aesthetic, colorful dragon look. Overall, we were able to complete this portion early in the timeline which meant that the rover was semi-functional early and both the software and electrical team were able to use it to start testing servos, wiring, and code. Below in Figure 5 you can see a photo of the rover outside on one of our testing days.

Figure 5: Final Rover build

2.2 Grabber

As mentioned at the start, the main body of the rover was one of two components we needed to design and fabricate. The second component was a grabber that could hold the payload as the rover traversed the course and release the payload once the rover reached a drop box. We decided to simplify this mechanism as much as possible, so we decided to make a grabber with a spring. This meant the servo motor would only have to function as the opening mechanism, and the grabber would not accidentally drop the payload in the wrong location due to a loss of power or a wrong signal sent to the servo. We also had to make sure the grabber arm did not obscure any of the sensors the rover was using for guidance. In order to avoid this issue, we initially though of making the grabber turn sideways while we were running, but once we clarified that we only had to hold one payload at a time, we decided to make the arm as short as possible and mount it statically. This meant we were able to avoid a longer arm blocking the view from the PiCam, and the need for the extra rotation servo to turn it sideways. Below in Figure 6 you can see photos of our grabber.

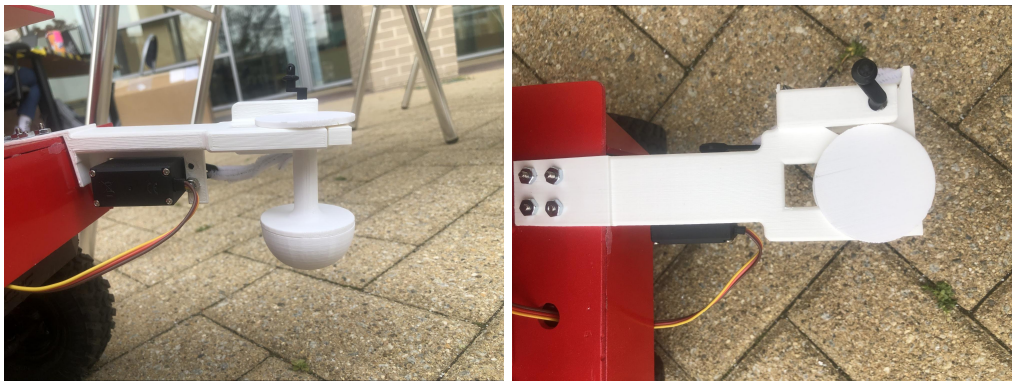

Figure 6: Final Grabber Design

3 Rover: Electronics Design

For the Electronics Design, we started off with the Wiring Diagram given to us at the start of the project, which can be seen below in Figure 7.

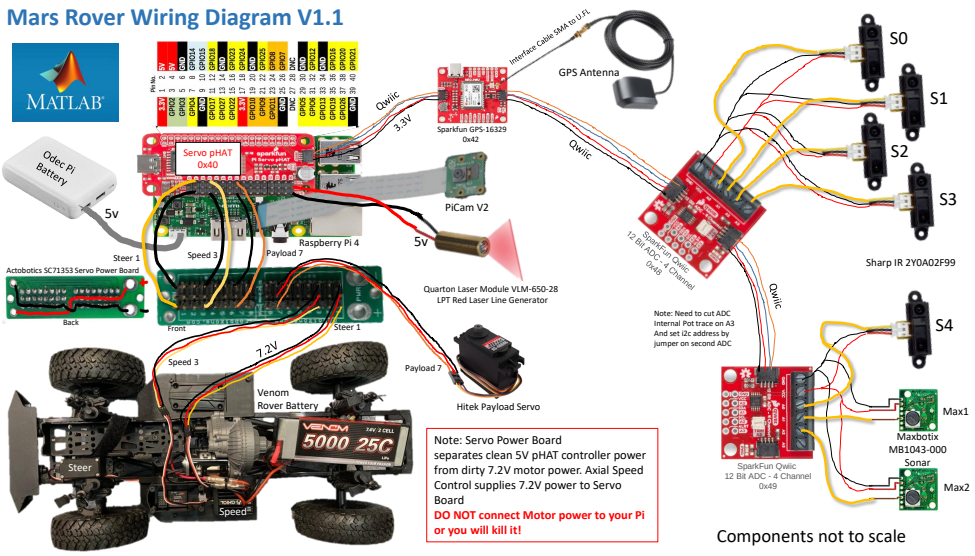

Figure 7: Original Wiring Diagram

From this, we kept a lot of the things we can see, with big changes being some pins being moved around, not including the laser, and the inclusion of the E-Stop. You can see our final wiring diagram in Figure 8 below.

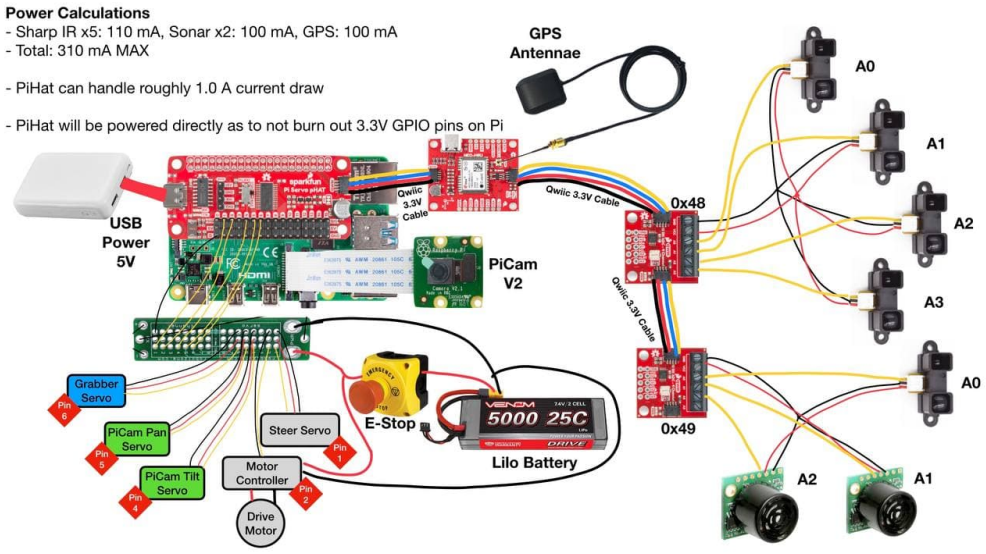

Figure 8: Final Wiring Diagram

After having done the diagram and power calculations the path to finish wiring was pretty straightforward. First of all we did a Roadkill Harness by connecting all the sensors outside of the rover body and making sure everything was working as expected. After this had been done and the body of the rover had been finished

by mechanical, we were able to bolt down our pi onto its placeholder and mount all sensors on the rover as we had designed. When mounting our sonars we had to do some un-soldering and re-soldering because we had soldering things for the Roadkill Harness and didn't realize the sonars wouldn't fit through our wiring holes. After we got through this hurdle and finished wiring everything else, we zip-tied and cleaned up wiring as much as possible so it didn't look like a mess. Below in Figure 9 you can see a photo of the final wiring in the rover.

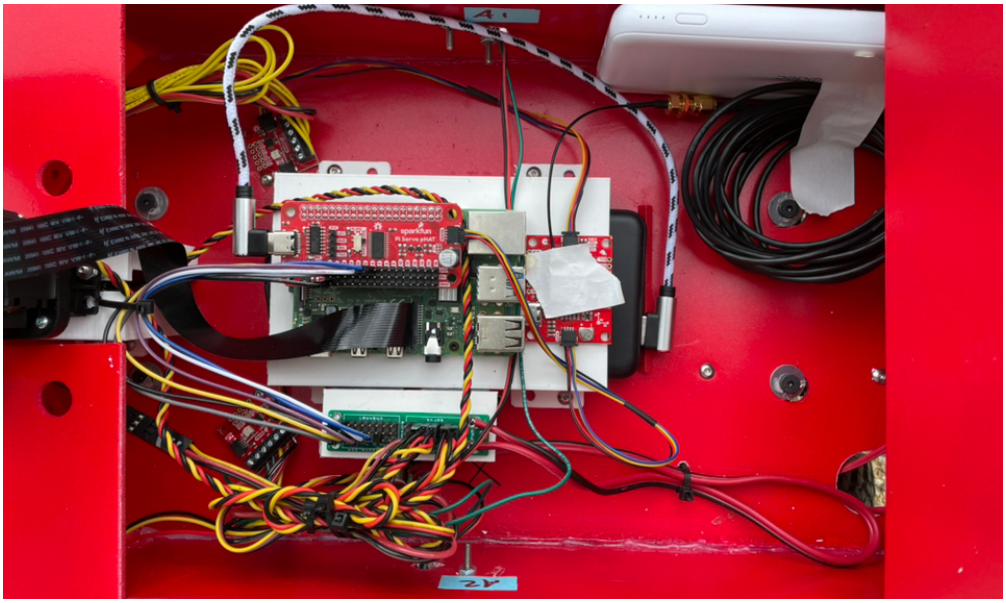

Figure 9: Final Wiring in Rover

4 Rover: Control Software

For the start of our Control Software development, we began by creating our Finite-State Machine (FSM) that defines the way the rover is thinking and acting. Below in Figure 10, you can see our FSM.
As you can see in the FSM, our code is broken into multiple different parts. In our actual development, we separated our files into 4 distinct sections: SETUP, SENSE, THINK, and ACT. You can see in Figure 11 below that we separated them into 4 distinct folders for more clarity when working.

4.1 SETUP

Our setup functions are divided into two parts. First, we have the functions we were given by default, those two are:

- $ads1015.m$ to setup the sensors through the Qwiic
- $I2C\_Servo\_pHat.m$ used to setup the servos we are using through the pHat and return the roverServos object which we use to control them.

On the other hand, we have the setup functions we created, of which there are 4:

- $SETUP\_adc.m$ which uses $ads1015.m$ to create our adcDevice object for the different connections we are using to the sensors.
- $SETUP\_gps.m$ which we use to setup and calibrate the GPS in the cases where we use it.

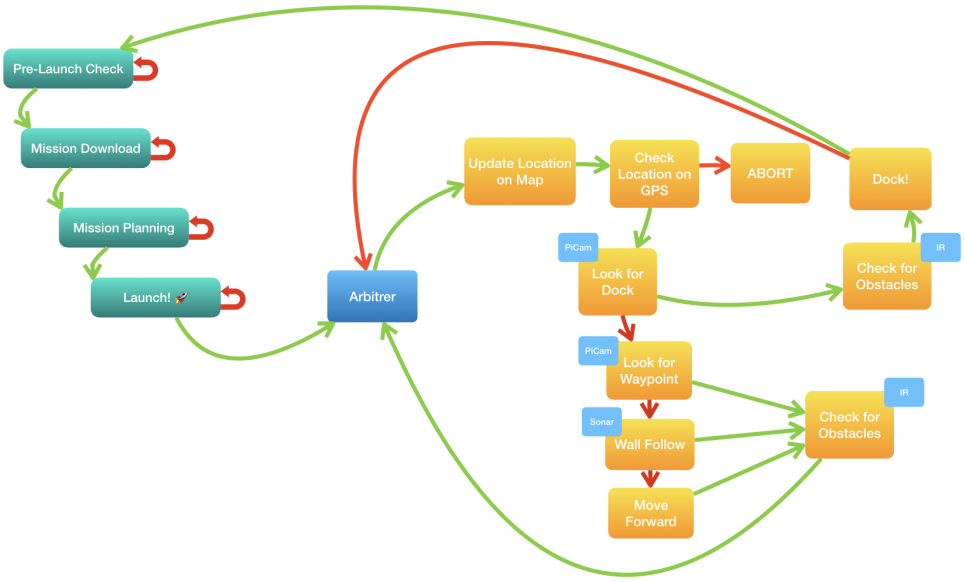

Figure 10: Finite-State Machine (FSM) Diagram

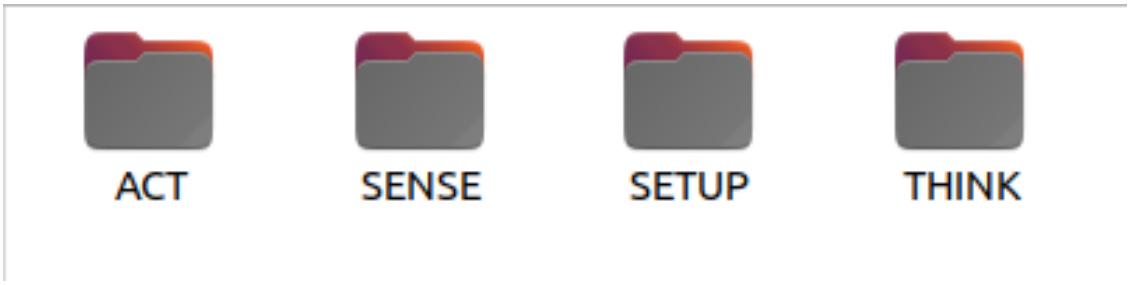

Figure 11: Function Folders

- *SETUP_pi.m* which connects to our Raspberry Pi on the rover and sets up the blink LED on it and returning our robotPi object.
- *SETUP_piCam.m* which sets up the piCam when its connected to our Pi, giving it our preferred settings and creating the robotCam object.

4.2 SENSE

On the side of our SENSE functions, we have created a good amount of them for different sensors and cases of use. For starters we once again have functions that were given to us, which in the case of SENSE are all GPS-related functions. These are: *MEO_M8U.m*, *UBX.m*, and *UBX_CLASS_ID.m*. On the other hand, we have a bunch of SENSE functions we created ourselves which can be separated into 4 categories, piCam, IR, Sonar, and General Use.

PiCam The PiCam is the sensor with the most SENSE functions. The reason for this is that the PiCam is our main method of sensing and choosing where to move the robot. For starters we have a base April Tag detector function (*detectAprilTags.m*) which senses where the April Tags for docks are. This function is paired with *SENSE_piCam_Dock.m* and *SENSE_piCam_PayloadDock.m* which sense where the docks are and tell the rover to move to them if they can be seen. Besides that, we also have a general *SENSE_piCam.m* function that is used when we pause and look around and a *SENSE_piCam_Waypoint.m* which are used

to create movement curves by looking for the waypoints in the ground. For a good while we debated what type of navigation to use, we tested GPS but it was not accurate and consistent enough for out practice. We eventually got a Waypoint follower working with $SENSE\_piCam\_Waypoint.m$, but the masks were very peculiar about the lighting, and being outside with shadows and sun is not super useful. We also tried a line-following method in $SENSE_piCam_Line.m$ by putting the piCam at 90 degrees to the right and following the border of the O. This one also was a useful approach, very similar in quality to the way point follower. Eventually we decided to use $SENSE\_piCam\_Waypoint.m$ for our final demo because it was slightly more consistent in the long run.

IR On the other hand, the IR functions are very simple. We only have one function for sensing IR, called $SENSE\_IR.m$ which senses distances using our five IR sensors and creates a movement curve to evade objects close to the rover.

Sonar Sonar is very similar to IR. We only have one function for Sonar sensing ($SENSE\_Sonar.m$) which senses to both sides of the rover and creates movement curves to avoid walls/objects if they're any walls to the sides of the rover.

General Use Finally, we have one big general use function which is used by pretty much all of the SENSE functions: *createBellCurve.m*. This is a special function we created that receives a point from 1-60 and creates a bell curve with that point being the maximum. We use it to create movement curves that define where our rover is going to move.

4.3 THINK

The biggest section of functions we have, and the one we had the hardest time developing, is the THINK functions. We have one general $THINK.m$ function which is the one we call on every Robot Loop. This function uses the current roverBehaviour defined by the MDF to choose which specific THINK function is actually going to be run.

Before we go into the specific THINK function it is worth explaining what an MDF is and how ours is used. MDF stands for Mission-Definition File and what we have done is create a table that defines the different behaviours our robot has to do during our demo. Then, we have a WPn variable that defines what the current roverBehaviour is. This way, we can have the rover do multiple things with the same code, we just need to change what WPn is.

For a long while we debated what behaviours to put on the MDF, and how broken down or general they should be. Eventually, we decided to put the behaviours you will see shortly because they were a detailed enough breakdown that we could separate different parts of the mission, but general enough that it wasn't re-coding of things over and over again.

You can see an image of our MDF table below in Figure 12.

Before we go back to talking about the specific THINK functions, it is important to mention one big design decision we made with these: The Arbitrer. We decided to have an Arbitrer built-in with each THINK function. This was a hard decision because that's not how we were taught to do it, but we decided to go down this path because our different functions check many different sensors (and sometimes motors) so having one general Arbitrer would just have meant we would have to generalize things to a level that seemed unreasonable. Back to the specific THINK functions, there are two sections of THINK functions that can be chosen by $THINK.m$ which are the same two sections our MDF is divided into: Navigation, and Payload.

Navigate There are four navigation THINK functions in our MDF:

- **PauseDock**_____, which calls $THINK\_pause.m$, which stops the rover and has the piCam look around and turn to whichever way we should be looking.

1 WayPoint	2 xLongitude	3 yLatitude	4 Behavior
1	42.2763	-71.2279	PauseDock_____
2	42.2763	-71.2279	NavigationDock_
3	42.2763	-71.2279	FindDock_____
4	42.2763	-71.2279	Dock_____
5	42.2763	-71.2279	PauseNav_____
6	42.2763	-71.2279	NavPayload_____
7	42.2763	-71.2279	PayloadFindDock
8	42.2763	-71.2279	PayloadGoDock__
9	42.2763	-71.2279	PayloadDrop____
10	42.2763	-71.2279	PayloadFind____
11	42.2763	-71.2279	PayloadPickup__

Figure 12: Mission-Definition File (MDF)

- **NavigationDock**_, which calls $THINK\_navigate.m$, which is the main navigation function, using piCam, Sonar, and/or IR sensing to move around the Olin O.
- **FindDock**_____, which calls $THINK\_findDock.m$, which looks around sensing April Tags to point our rover towards the dock once its been found.
- **Dock**_____, which calls $THINK\_go2Dock.m$, called only once $THINK\_findDock.m$ has found a dock, it is used to actually get our rover into said dock.

There are many important things about these previous functions that are worth mentioning. First of all, besides deciding where and how fast the robot should move, they also define what the roverBehaviour should be in the next loop. They do this by changing WPn and they do this in respect to the current WPn which will come in handy in a second. The second important thing about these functions are that they aren't specific to the navigation MDF section. Both $THINK\_pause.m$ and $THINK\_navigate.m$ are used in Payload missions, they just are in a different position in the MDF so they go to different roverBehaviours afterwards.

Payload Talking about the Payload THINK function, we have many of them, which can be seen listed below:

- **PauseNav**_____, which calls the same $THINK\_pause.m$ function as above.
- **NavPayload**_____, which calls the same $THINK\_navigate.m$ function as above.
- **PayloadFindDock**, which calls $THINK\_findPayloadDock.m$ a function very similar to $THINK\_findDock.m$ which looks for the smaller Payload Docks instead of the big rover ones.
- **PayloadGoDock**__, which calls $THINK\_go2PayloadDock.m$, which has the same structure as $THINK\_go2Dock.m$ but instead of going to a bigger rover dock it goes to the smaller Payload docks.
- **PayloadDrop**____, which calls $THINK\_dropPayload.m$ which takes care of dropping the payload into the dock .
- **PayloadFind**____, which would call $THINK\_findPayload.m$, this was one of our stretch goals to be able to find a pickup Payloads around the track. We weren't able to implement this because of many different reasons not worth mentioning.

- **PayloadPickup**____, which would call $THINK\_payloadPickup$, this was one of our stretch goals to be able to find a pickup Payloads around the track. We weren't able to implement this because of many different reasons not worth mentioning.

4.4 ACT

Finally, we have our ACT functions. As mentioned above on the Wiring section there are 5 servos on our rover. We have the servos that move the rover itself (speed and steering), the servos that move our piCam (pan and tilt), and finally the servo that controls our grabber to drop payloads. This means that we have separated our ACT into three functions that each control one of these three types of servos: $ACT_moveRover.m$, $ACT_moveCam.m$, and $ACT_movePayload.m$.

4.5 Actual Development

The actual development of this code was interesting. For the first week we started by creating the paths and structures for all of the functions mentioned above, and spent a lot of time understanding the Demo code we were given, and adapting it to our needs. A lot of this time was spent on THINK function structure and the MDF and how we would use it to complete our different missions.

Afterwards, we started by developing our ACT functions. We decided to start here because once those were working we would be able to drive the rover around with the joystick, which was already a huge step. These functions were really simple to get, $ACT\_movePayload$ was the last one we did because the grabber attachment hadn't been made when we started but besides that all the servos were easy to get working. Once we had these functions working, we moved first into SENSE and then into THINK, but not completely linearly. We started with crucial functions and then separated our team to get these done at the same time. This meant we start with some people doing SENSE calibration and curve creation, while others took care of populating the THINK functions so they were more than skeletons.

Besides the function, our code also has a big folder called **test_code** where we have multiple tests for different function (both SENSE, THINK, and ACT) which we used constantly throughout the project to make sure our functions were working properly. There are multiple types of tests here, some are just one-off live scripts to get the setups running fast and easy, while others are full on tests that check all of the rovers functionalities. Eventually, we had enough code that we were able to test straight from our demo live script by changing which roverBehaviour the system started at.

Finally, one important thing to mention is our starting J-turn to leave the Dock. After we had manufactured and built our rover, we realized that the tail we had designed for the GPS meant that he didn't fit completely inside the dock for the starting position. Our fix for this was simple: We started with him leaving the dock backwards. We knew his fit in this direction because we had planned it for the docking mission, so, knowing roughly where the starting dock was going to be, we pre-programmed the rover to do a backwards J-turn to leave the dock at the start and line up with teh waypoints around the O. This was a simple, easy fix that saved us a lot of trouble in the long run.

5 Final Demo: What Worked and What Didn't

Some important things to mention before getting into the actual demo is that our servos and wiring had a bad day before the demo. Our steering servo burnt out the night before so we had to basically take the whole rover apart to fix the RC setup for steering. This also meant we had to re-calibrate most of our movement the night before which didn't change anything drastically but it did affect some of our accuracy (because we just didn't have time to calibrate as thoroughly as we had before). Besides that one our Qwiic connectors decided to output a string of 0 volts read to our pi, which we couldn't debug because we just didn't know what was causing it. We tried re-wiring it, changing the sensors, everything, but even the potentiometer was reading 0 volts instead of the normal 3.

On the actual demo Day we were able to replace our Qwiic connector but couldn't test it, so we decided to keep it as a backup but not rely on it working. MATLAB was not being helpful and crashed multiple

times during the demo, but overall we were very happy with our results. On the first mission (Navigation around the O) our rover didn't do very well. It didn't sense the waypoints as we expected him to, and it took a longer time to run than we expected (because MATLAB was having some problems that eventually fixed themselves), so we didn't get very far. On the bright side, for Mission 2 (Docking) we had planned to do the whole navigation from Mission 1 to get to the dock we needed to reach for Mission 2. Thus, we were able to showcase our waypoint following code, and our obstacle avoiding code which had failed the first time through, and it docked almost perfectly. Finally, on Mission 3 we had problems with our computer that was running our code, so we had to change, but we were able to get the rover to run up to the dock and open the grabber we designed to drop the payload. For some reason it didn't actually drop (The servo might have needed re-calibrating after all the wiring we had to change last second), but you could see that the rover had recognized everything properly.

Overall, we're very happy with how the Demo went, and even though it wasn't perfect, our code worked close to what we expected, even through all the last minute changes/fixes we needed to do.