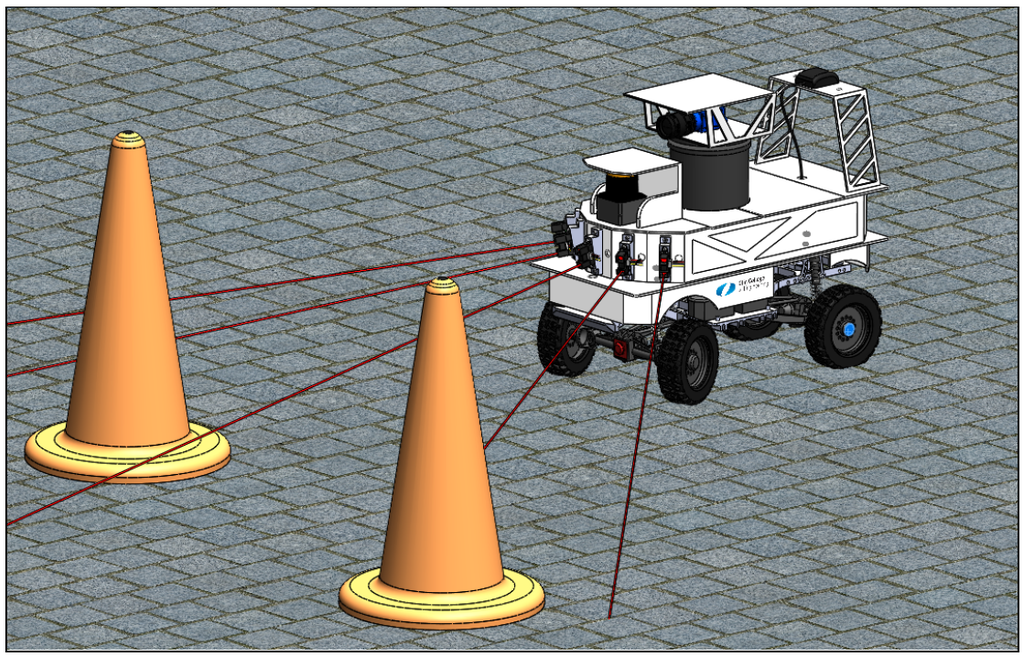
ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

Each year, Fun-Robo has a set of formative **Sense-Think-Act** labs that introduce you to the basic hardware of modern robotics as well as to help you build your software skills in using that hardware. Following this heavily scaffolded set fundamental skill building lab part of the course, we always have a summative, **build an actual robot**, team-based design-built project experience. This year it will be a (relatively un-scaffolded) design-build-demo of a fully autonomous Jungle Explorer rover doing outdoor missions around the Olin Oval.



Your course Ninjas have created a **great Jungle Explorer rover contest** to both stretch your current skill set and to help you integrate the many new skills you have picked up from the labs into an actual working robot that you will carefully design and build to perform a set of complex multifaceted autonomous missions.

Project Overview

Your team is exploring abandoned ruins in the jungle. It's too dangerous for people to explore so you send out your autonomous rover to scout the area and collect information. Your rover needs to start in a designated start-end dock, explore the area, and then return to a designated start-end dock. Along the way, there will be April Tags placed throughout the ruins to help guide you.

Project goals, Your team should:

1. Design & fabricate a body for your rover (you're provided with the chassis, but you will need a robot body to mount your sensors and electronics to)
2. Create sensor mounts to attach sensors to your rover
3. Wire up sensors & create a wiring diagram
4. Write code that will allow your rover to autonomously navigate using LIDAR, April Tags, GPS and your other sensors.
5. Choose additional stretch goals to attempt based on your team's interests
    a. Slalom course
    b. Bridge crossing
    c. Gate passage
    d. All three!

ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

Demo day format:

1. You will have two 15 minute time slots.
2. You can complete the course as many times as you'd like in your time slot but you can only do each challenge once per lap. Can double points by doing 2 laps.
3. You may "intervene" with your robot if absolutely necessary but you may not remotely control your robot and your code must not rely on manual interventions.
4. Completing tasks will give your team "mission points" - Note that your grade is NOT dependent on the number of "mission points" your team ends up with, but rather the quality of the report and the level of effort put into this project
5. At the conclusion of the event, the following awards will be awarded by a panel of guest judges who will oversee the race:
    a. Best engineered rover
    b. Prettiest rover
    c. Most daring rover design

We will keep track of progress using a table similar to this:

	Team 1	Team 2	Team 3
Round 1			
MVP: Circumnavigate the O	I	IIII	II
Slalom Course	I		II
Bridge Crossing	I		
Gate Passage	I		I
Round 2			
MVP: Circumnavigate the O	III	II	II
Slalom Course		II	I
Bridge Crossing		II	I
Gate Passage	IIII		II

Winning Team and runners up will be placed by the total number of points accumulated in their two time slots.

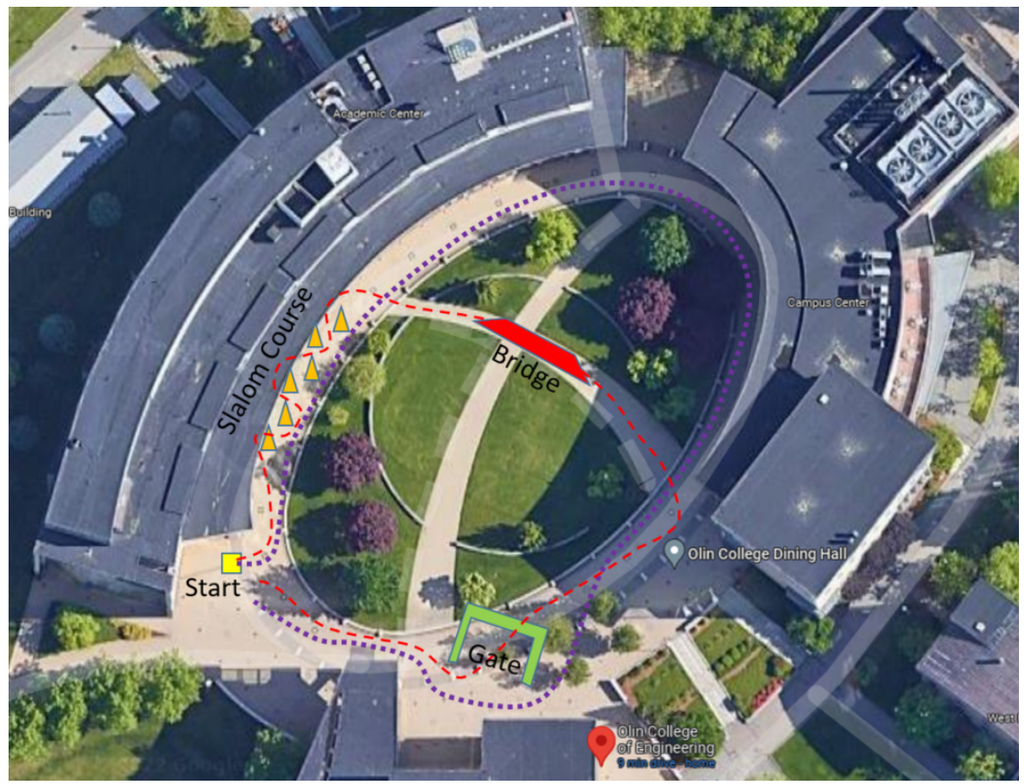
ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

Hardware, Project supplies list:

- A Mojave rover that has the low-level systems built, where the rover's steering and velocity can be controlled with PWM signals from the Arduino
- A NUC high speed mobile computer
- An Arduino BLE pro microcontroller.
- 2 sheets of Sintra board to make the rover hull.
- 2 Maxbotix Ultrasonic sensors (MB1340-000 sensors, see appendix for detection range).
- 6 Sharp IR sensors (GP2Y0A41SK0F Model, 4-30cm detection range)
- A high-resolution USB camera (8MP camera)
- A precision Hokuyo Lidar
- A micro GPS
- Servo Pan mechanical subassembly
- Two batteries

Note: The above supplies will be available to your team, but you are not required to use all of the supplies listed. Spec sheets for each of these sensors, sample code, rover details, and other specific hardware-related info will be available on the Canvas website.

Ruins RaceCourse



Rover travel will be on one of several possible (team decided) paths around the Olin Oval (as shown above). The Oval course will have an orange safety cone slalom course, a challenging bringe to climb and a gate to pass through. Teams get extra points for taking on more obstacles. Teams can decide which obstacles to take on and which path to take on race day.

ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

MVP: Circumnavigate the O

At minimum, your rover needs to be able to reliably return to you after its exploration. There will be a start dock and end dock placed near each other at the end of the MAC near Lot A. The docks will have large April Tags on them on a red background. There will also be April Tags and large Red Circles, distributed around the pillars of the ruins (the O). Your task is to begin in the start-end dock, autonomously drive all the way around the O, and return to the start-end dock. If circumnavigation is the only challenge you are completing on that lap, you must circumnavigate the entire O, passing through the section near Babson. If you are completing other tasks, such as the bridge, you may take a short cut across the path through the center of the O. If you are completing multiple laps during your time slot, you do not need to park in the end dock every time, only the last time. Parking is defined as having your front wheels on the end dock.

Challenge 1: Slalom Course

Part of the ruins have crumbled down. There's some valuable data there but you can't touch the ruins to preserve the site. Fortunately, these particular ruins are bright orange traffic safety cones:

Near the start of the course there will be a grouping of bright orange traffic cones. Your challenge is to safely navigate through the cones without touching any of them. The grouping will be in the same general place every time but these ruins are unstable and the individual cones will shift locations between teams/rounds.

Challenge 2: Bridge Crossing

The fastest path through the ruins goes right through the middle. But unfortunately, there was an earthquake earlier and the ground split open. Fortunately, there is still a small bridge left standing over the chasm from a past research team.

In the middle of the O, there's a low plywood bridge. You can't pass through the middle of the O without crossing this bridge. The top will be approximately 4 inches off the ground, and there aren't any railings. The flat section of the bridge will be 3 ft wide by 5 feet long and at either end there will be a ramp. Due to the earthquake activity, the bridge may slightly shift location, but luckily, there is a large April Tag on it to help you find it. Your challenge is to safely cross the bridge without falling off.

Challenge 3: Gate Passage

As you near the end of the course, an earthquake strikes. Fortunately, there's a sturdy looking arch nearby for you to take shelter under.

In front of Milas Hall, there will be a low arch made out of plywood. Your challenge is to drive under the arch, pause for 5 seconds to wait out the earthquake, and then continue on. The arch will be at least 3 feet wide and at least 3 feet tall. It will have a large April Tag on it to identify it.

ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

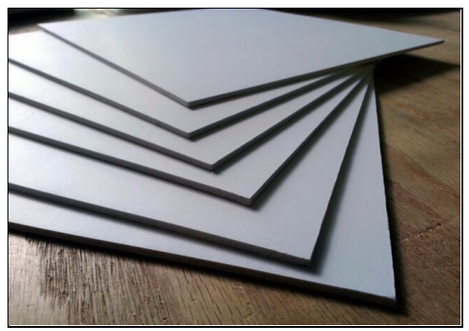
Having reviewed the contest, the following three sections will provide you with a bit of background and some technical help in the mechanical, electronic and software design of your planetary rover.

Mechanical Design

Your team's planetary rover will be built on top of a slightly used **ARRMA Mojave**

https://www.arrma-rc.com/rc-cars/latest/mojave/blx

This base vehicle has phenomenal traction and maneuverability. The fancy plastic car shell has been removed and your team's design task is to make a reasonably weatherproof robot shell out of 3mm Sintra board to house your electronics and sensors.



Sintra is an easily worked, more robust version of the foam core panels used for posters in science fairs and Olin's EXPO. It can be cut with an Exacto knife, shopbotted, solvent bonded, cnc-routed, hot-melted and thermally formed into a wide variety of 3d shapes. It is commonly used for making cosplay armor. Please see embedded videos for working with Sintra, before starting on your robot hull design.

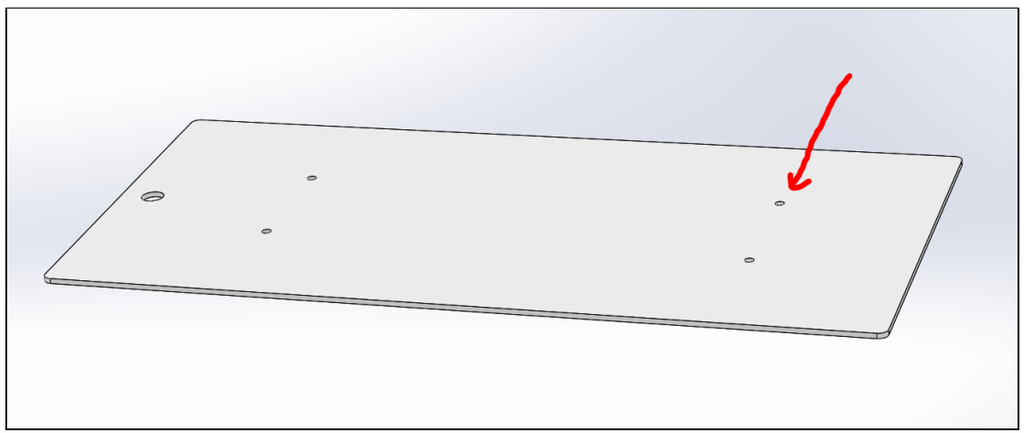
Bottom Hull Design

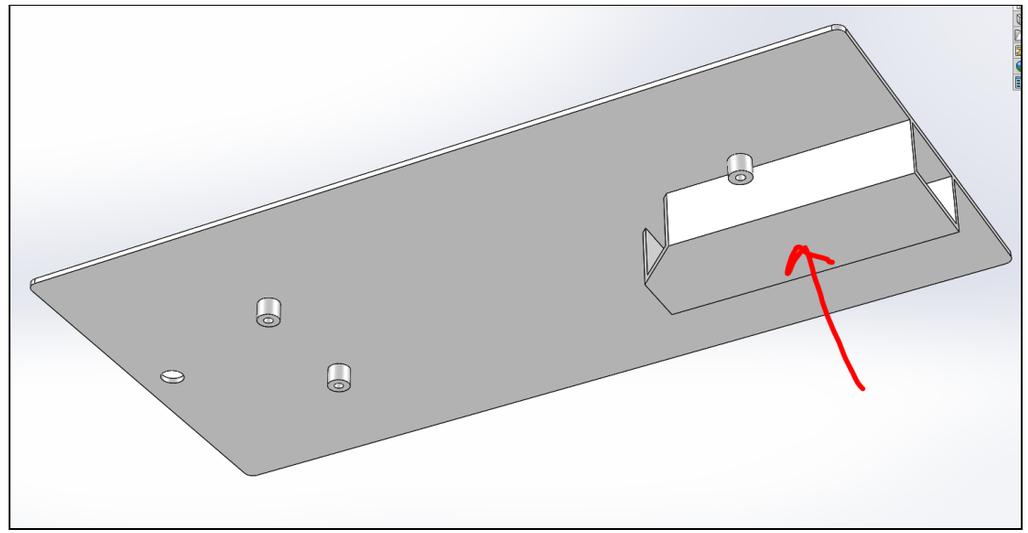
Your team can design any size and shape bottom hull for your rover. Your goal is to build a nice looking Sintra box that can hold and protect all of your electronics while providing appropriate attachment points for your sensor suites. You will want to make sure you have hand clearance to assemble components into the box and that you leave ample space for wiring so that you can connect up all of your electronic parts. Please also consider leaving space to add and remove batteries and to plug your NUC into shore power when you are coding for long periods of time.

The ARRMA Mojave chassis has 4 hardpoints that sit above the shock absorber towers to hold the shell in place. We will use them to attach the base plate of your Bottom Hull to it.

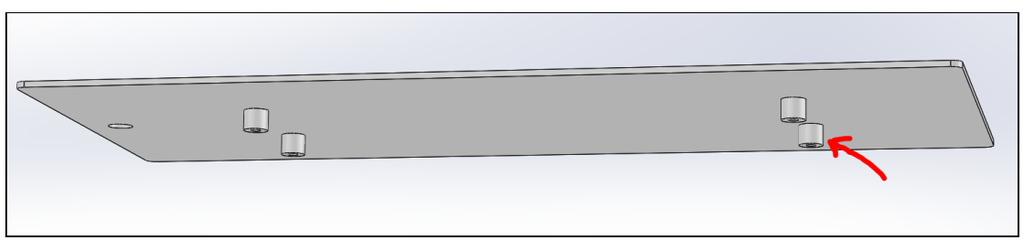
ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

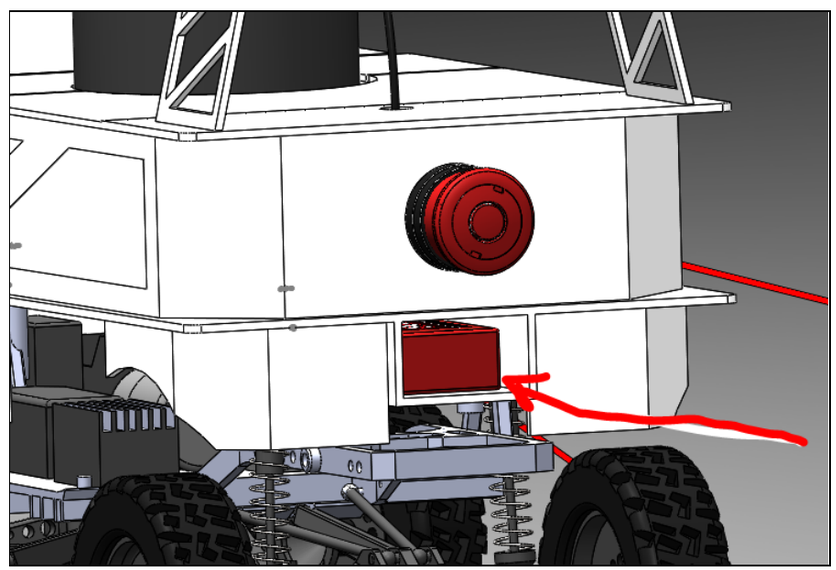
To start your design, lay out a 3mm thick (Sintra sheet is 3mm) bottom plate with the 4 attachment holes in place. Measure attachment holes carefully, you only get one shot at this:





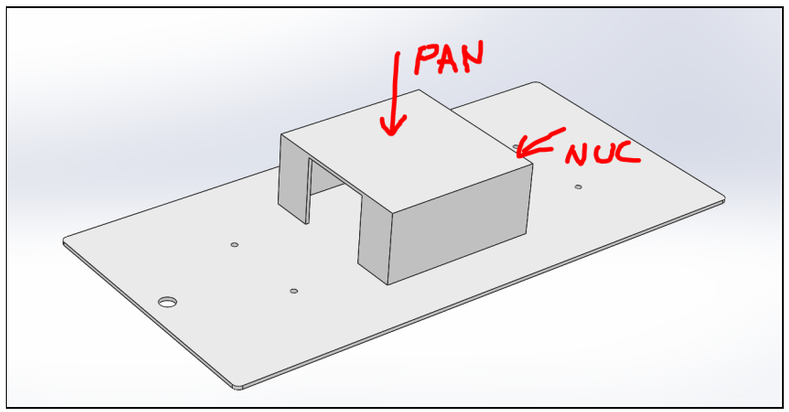
Add about (three sheets of 3mm Sintra) spacers on bottom to get your baseplate up over the Rovers internal parts (if needed):

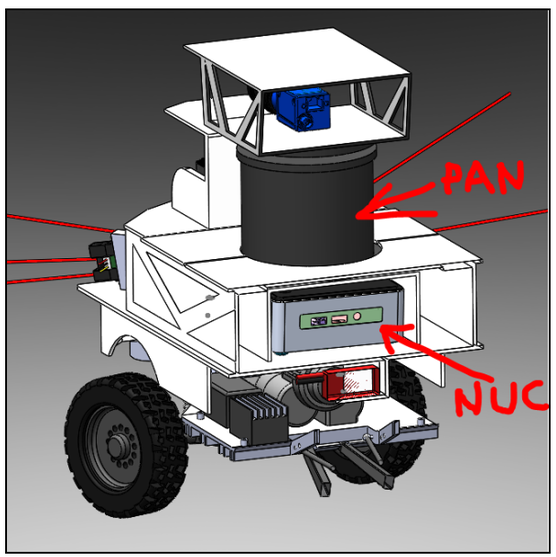




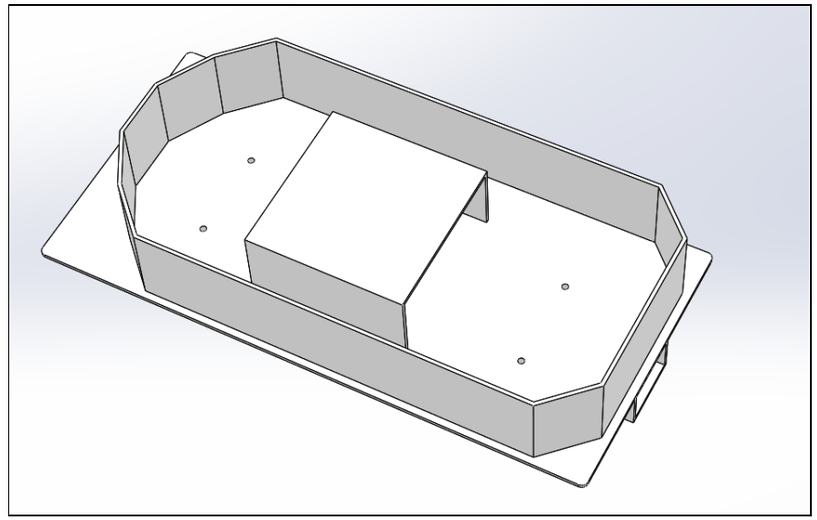
Then add a rover drive battery and computer battery (Venom,) holder down below the top deck to keep the weight low and centered. Make sure you leave a spot for power wires to exit the battery holder toward its front end. All construction needs to be done with cut and welded 3mm Sintra panels. Using tabs and slots for alignment is good.

Next add a raised platform in the center of your Rover to house your Team's NUC and to provide a place for the pan tilt mount. Every Rover will have one battery for clean computer and sensor power and one dirty battery for drive motor power. This raised platform can hold the pan tilt,

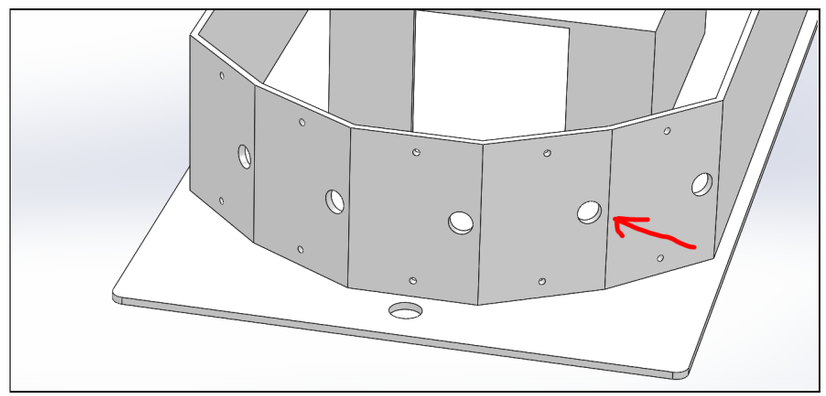




All around the NUC platform, create a raised wall (hull) to protect the internal electronics from the elements:

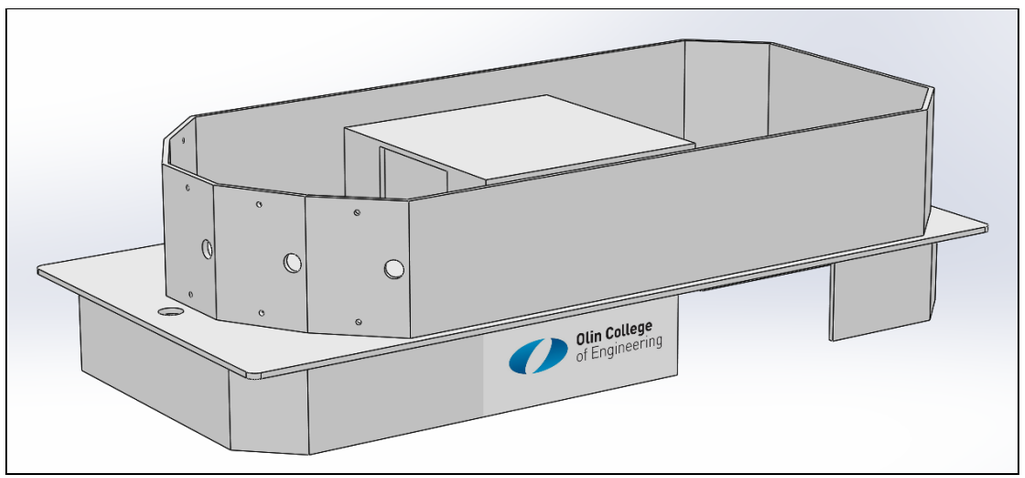


Cut a set of holes to mount your forward looking Sharp IR sensors and Sonars (the mounts themselves can be 3d printed later.

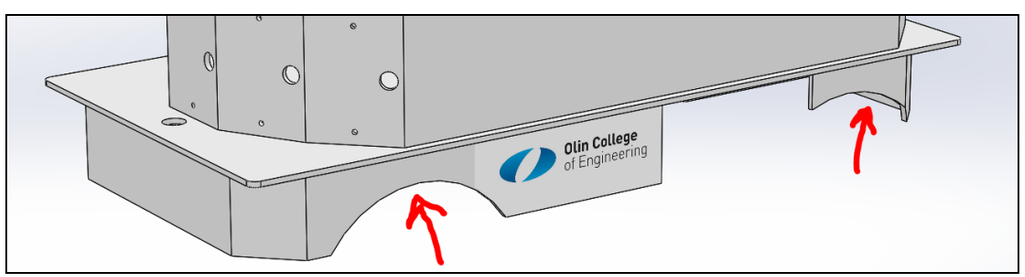


ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

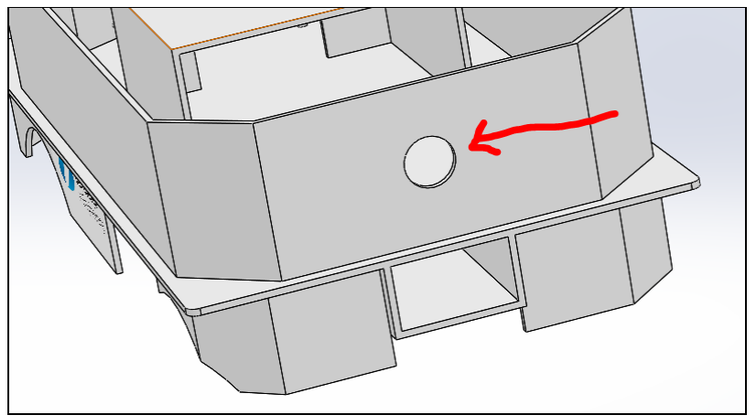
Having designed the main subassembly mounting features of your upper hull part, it's time to cover as much of the lower hull as you can to keep dirt and sticks out of the Mojave chassis. This box doesn't need to be downpour grade watertight, but it would be advisable to get it as sealed up as you can, while making it look as cool as your team is able. Please create a set of bottom hull walls to solvent bond to bottom plate as shown:



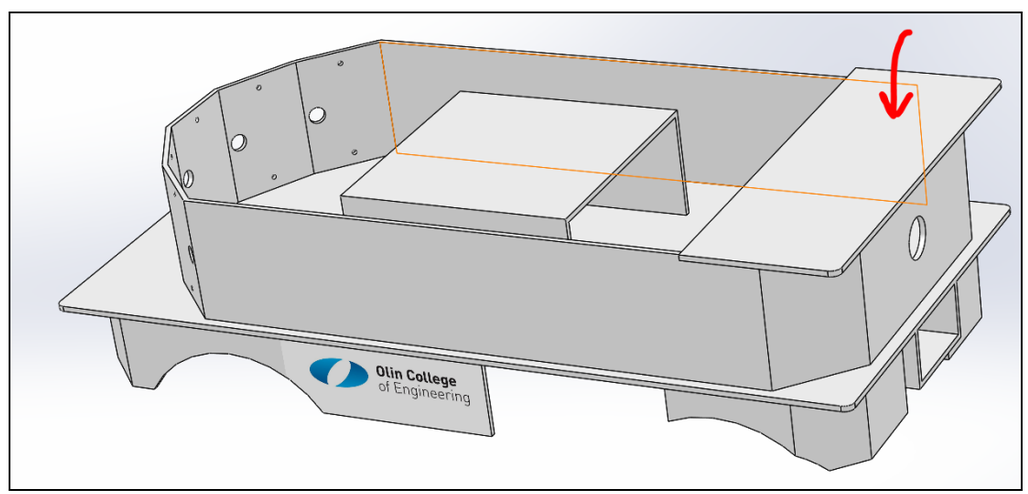
Take some time to measure carefully and get tire cutouts right so that body doesn't hit tires in normal operation:



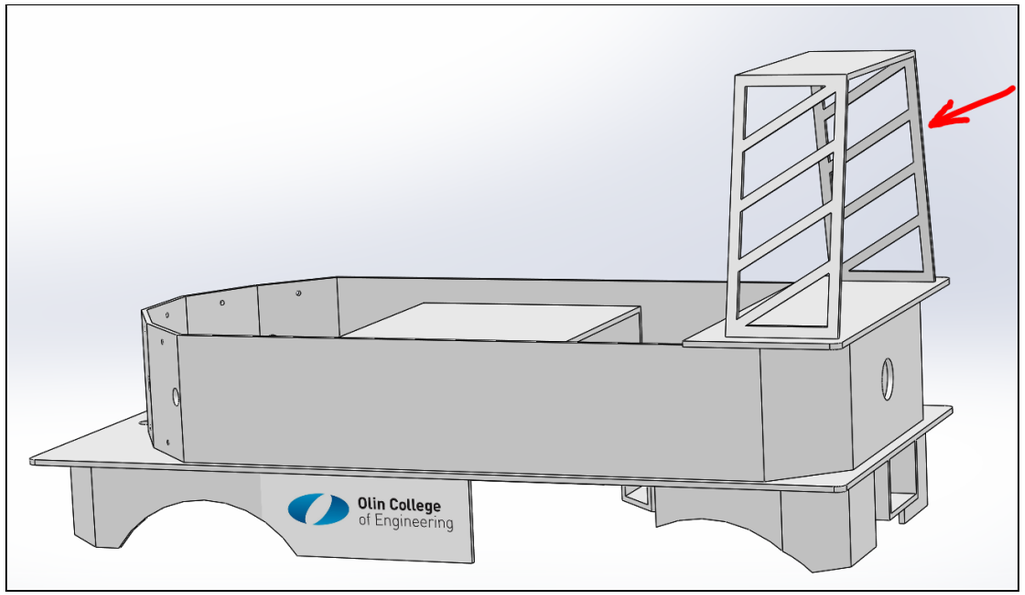
Add a back bumper hole for a big E-Stop switch (all robots need a blinky light and an E-Stop!).



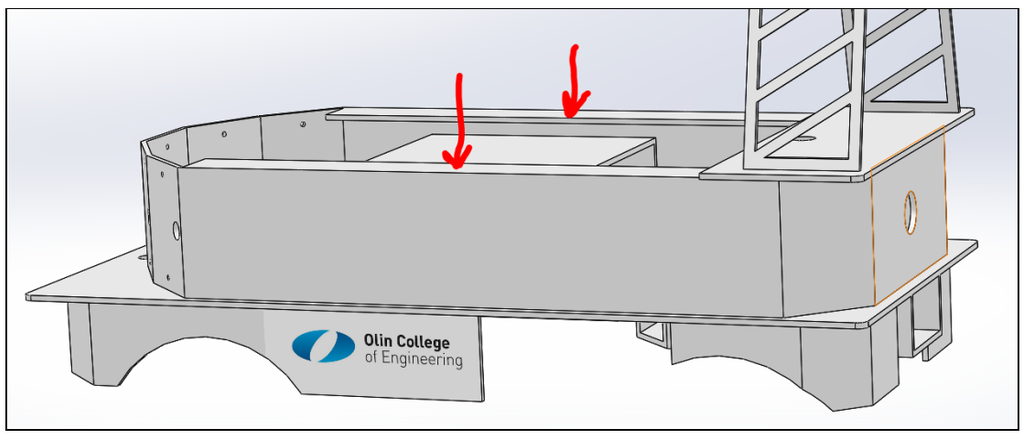
Add on a rear deck to support GPS tower:



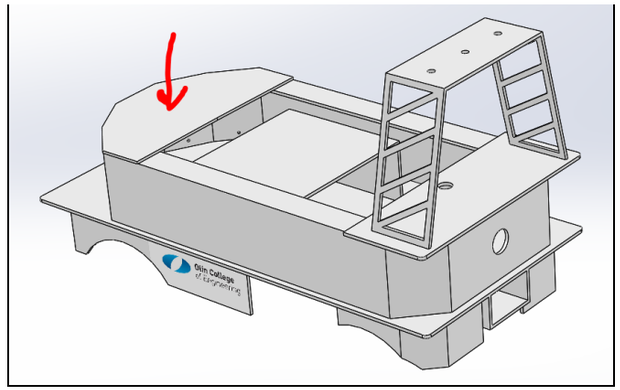
Add in a tower to get GPS antenna up high for a good signal:



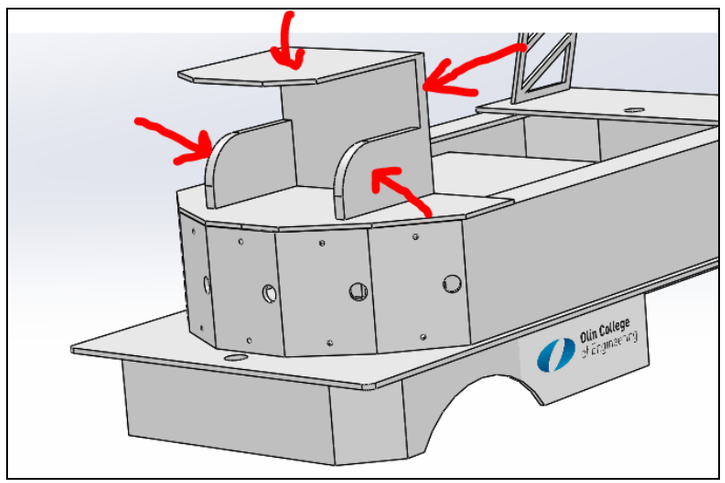
Add some side decks to stiffen hull body up and give top covers a place to rest:



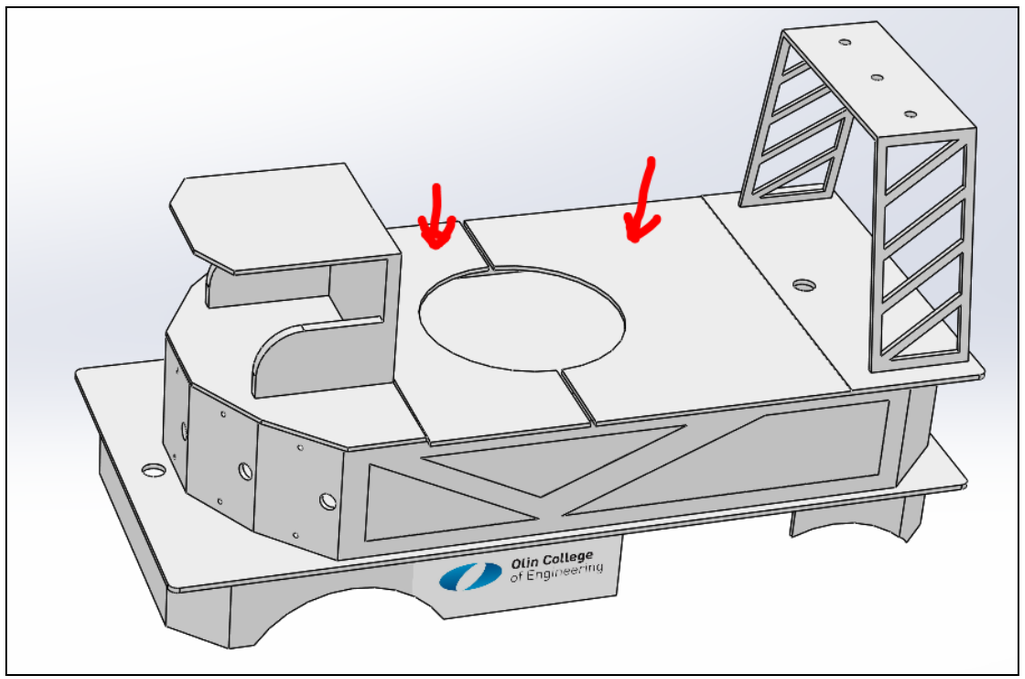
Add a front deck to give Lidar a place to sit:



Build up a Lidar rain and rollover protection structure on this front deck. Remember to left access holes for wires (ME's always forget the wires and drill ugly holes afterwards, DON'T DO THAT!). Protecting the Lidar from damage during a crash or roll-over is critical. They are really expensive and if you break yours, you won't get another….I can't afford it!



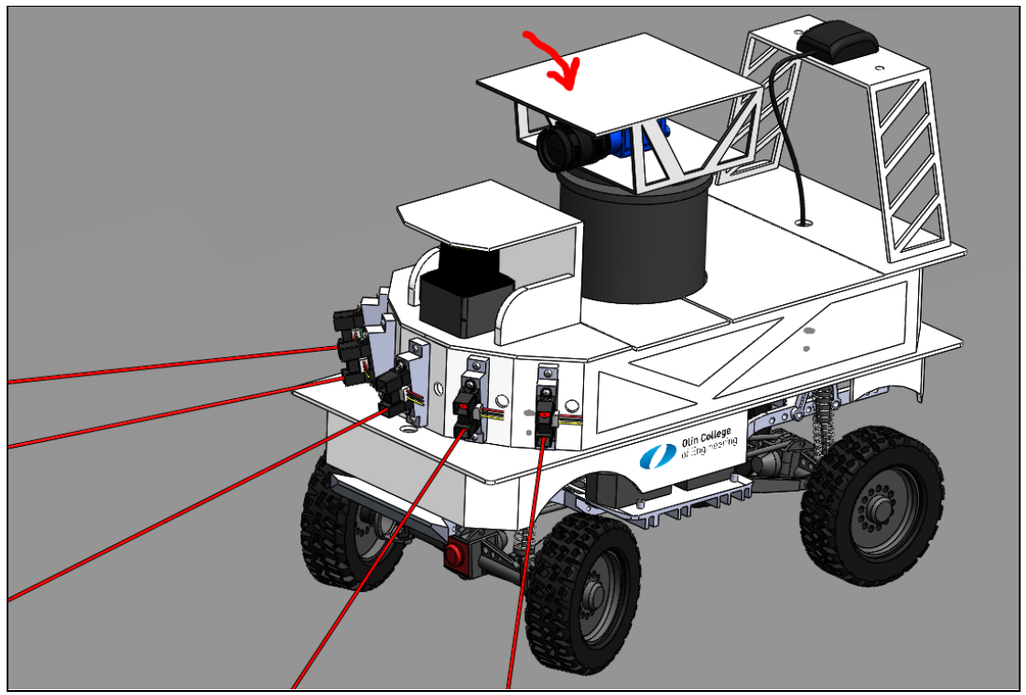
Almost done here. Add a couple of easily removable top covers to go around the pan but provide environmental security for the electronics inside the hull. Ideally your robot could handle a light rain. Overall your hull system should look somewhat like this:



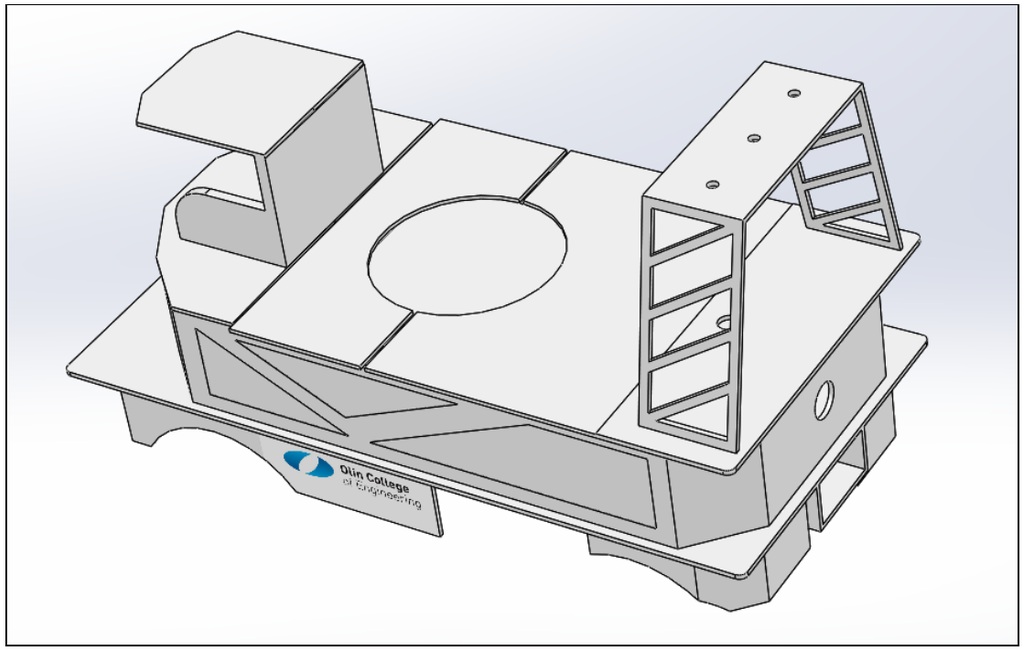
Please feel free to add as many details and style features as you like to make your robot look cool. This robot has two jobs; the first is to win the Ninja's robot race, the second is to help you land an awesome robot job this summer. So go all out, sculpt, bend, paint, decal.

On the technical side, go back through design and add threaded insert, wire pass through holes and wire conduits everywhere they are needed. These features are easy to put in when sheets are still flat stock (either on ShopBot or by hand) but can be a bear to add once the hull is all solvent welded together.

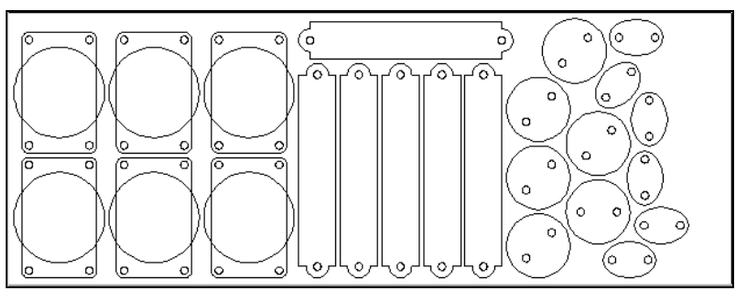
Add a weather-sun hat to pan to protect camera and then when fully assembled your wonderful new rover should look like this:



Make a final pass after design is firmed up to go through and add structural reinforcements (wall thickeners and gussets) wherever anything looks like it might be too flexible or wobbly, like the hull side walls.:



When done, lay out all parts hull parts flat in a single SolidWorks assembly drawing, packed tight, minimizing material waste and then print out full size and tape down to your Sintra sheet and start cutting them out with your Exacto knife:



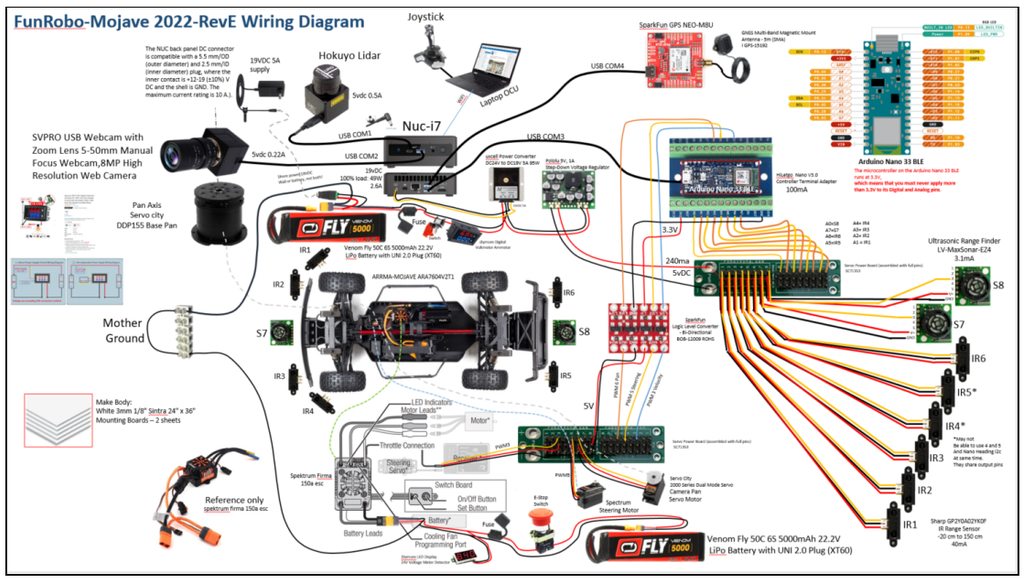
Or save your cut sheets as a **YourRoverName.dxf** file and cut the parts out on a Shop-Bot. We have ⅛" diameter 0-flute Sintra router bits and can cut your sheet if you leave more than ⅛" between parts, round off all internal corners and don't have features below ⅛".

This task is purposely only lightly structured to give your team's MEs and builders something substantial to work on after a semester of software and wiring!

Please feel free to design your team's shells any way you like. It just has to be functional, light and good looking (no ugly robots in Fun-Robo!). If your team has a lot of hard-core ME's, go wild. If not, a simple box hull with a good paint job will work just as well.

Electronic Design

Your planetary rover has a lot of electronic components, fortunately the electrical hook up is pretty straightforward. An overall electrical wiring diagram can be found downloaded from Canvas website and is given below:



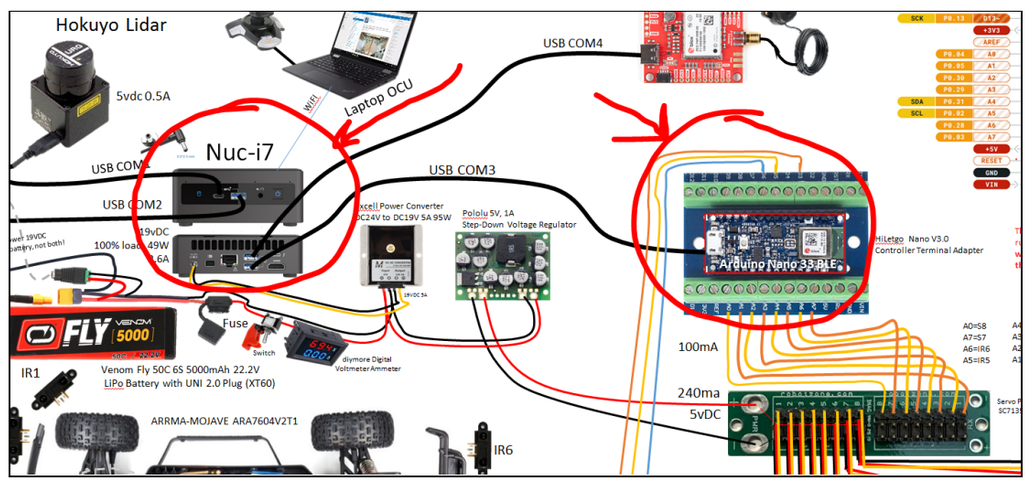
It is recommended that you print out a full size version, when wiring your rover up. Please note: **All electronics are DELICATE, NEVER HOOK UP ANYTHING HOT !!!** This course is way over budget and we don't have spares to bail you out if one of your team smokes a critical part by not being careful. If you do have a careless or sloppy teammate, have them write code! Don't let them touch the electronics.

Unlike many industrial classes of embedded controllers, yoru Arduino uses fairly fragile CMOS logic and as such voltages above 3.3V or static discharge from you wiggling your butt in a chair can easily burn out input pins or the whole board. It is good practice to never put your Arduino, Power or sensors down on anything conductive, never touch it except by the edges and **never ever connect or disconnect anything to it when it is powered**. A really sad way to screw the final demo is to hotplug something in the morning of demo day and burn out your whole board. **Never plug in hot, you have been warned**.

Lets walk through the electronic hook up part by part, so that you can design around each accordingly.

NUC and Arduino Nano BLE

Yor rover will be controlled by a combination of a NUC (Intel's Next Unit of Computing) for laptop level computation power and speed and an Arduino Nano BLE professional embedded microcontroller for its many device connection ports.



Details on the Arduino Nano BLE can be found here:

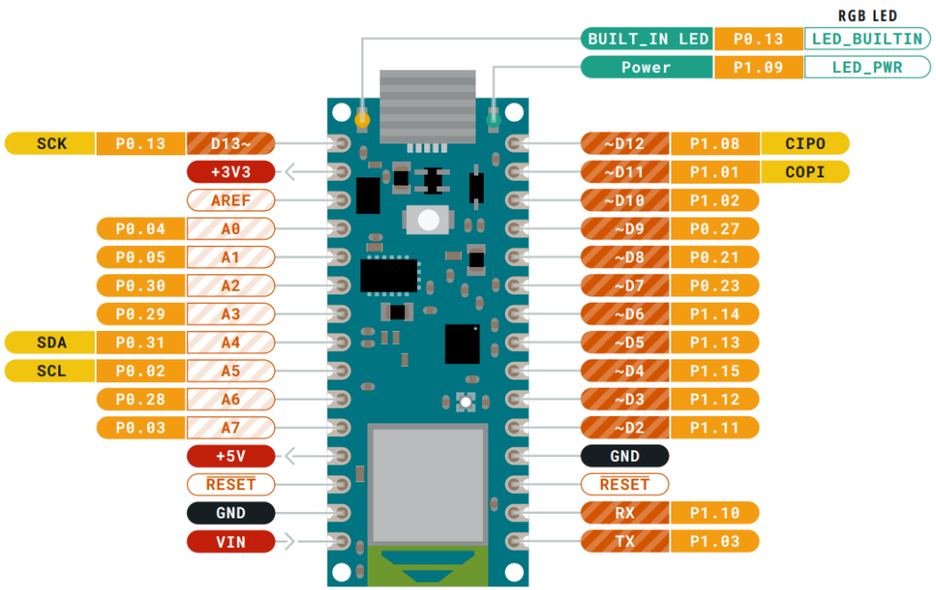
https://store-usa.arduino.cc/products/arduino-nano-33-ble?selectedStore=us

And a full technical manual covering ports, pins, voltages, etc. can be found here:

https://docs.arduino.cc/hardware/nano-33-ble

Its a pretty awesome little robot controller with a full IMU (pitch, roll and yaw) and a magnetometer (heading) built right in. You could use these sensors to prevent a roll-over or to do precise heading based navigation.

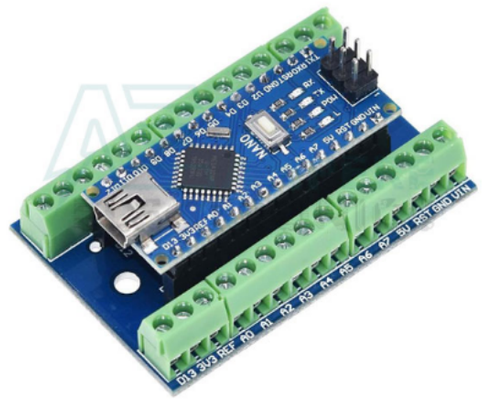
The Arduino Nano BLE has the following ports:



Including 8 analog inputs for your Sharp IR range and Maxbotix Sonar sensors. And 6 PWM capable pins to command your Rovers speed, heading and pan servo motors. Full BLE technical manual can be found here:

https://docs.arduino.cc/static/e9e1861ea882c15a88386afd84925eee/ABX00030-datasheet.pdf

In order to robustly connect to the Arduino Nano's pins we are providing a screw terminal carrier board to both facilitate and robustize connections to and from the Nano:



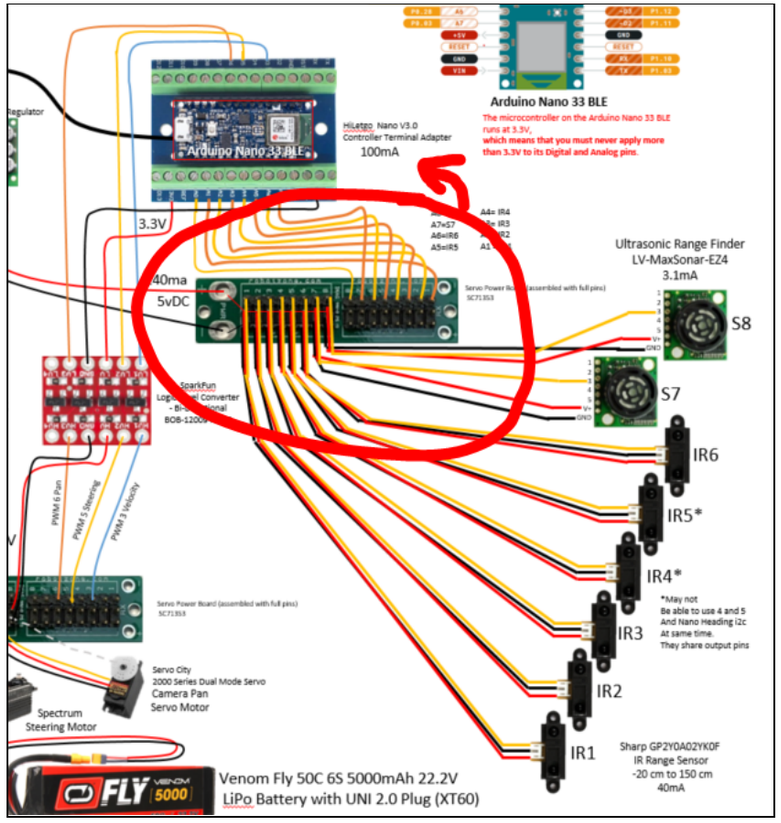
There are several functional components of the carrier board, which are designed to be as foolproof as possible. Although we have attempted to take precautions to guard against the most common user errors, users should still be wary. Most users may already be aware of these common pitfalls. However, if you have forgotten, users primarily need to:

- Be cautious of any loose connections and avoid shorting or bridging 5V and 3.3V on the Nano.
- Be aware of any potential current draw limitations. Only connect devices that draw current to Servo Power Board.

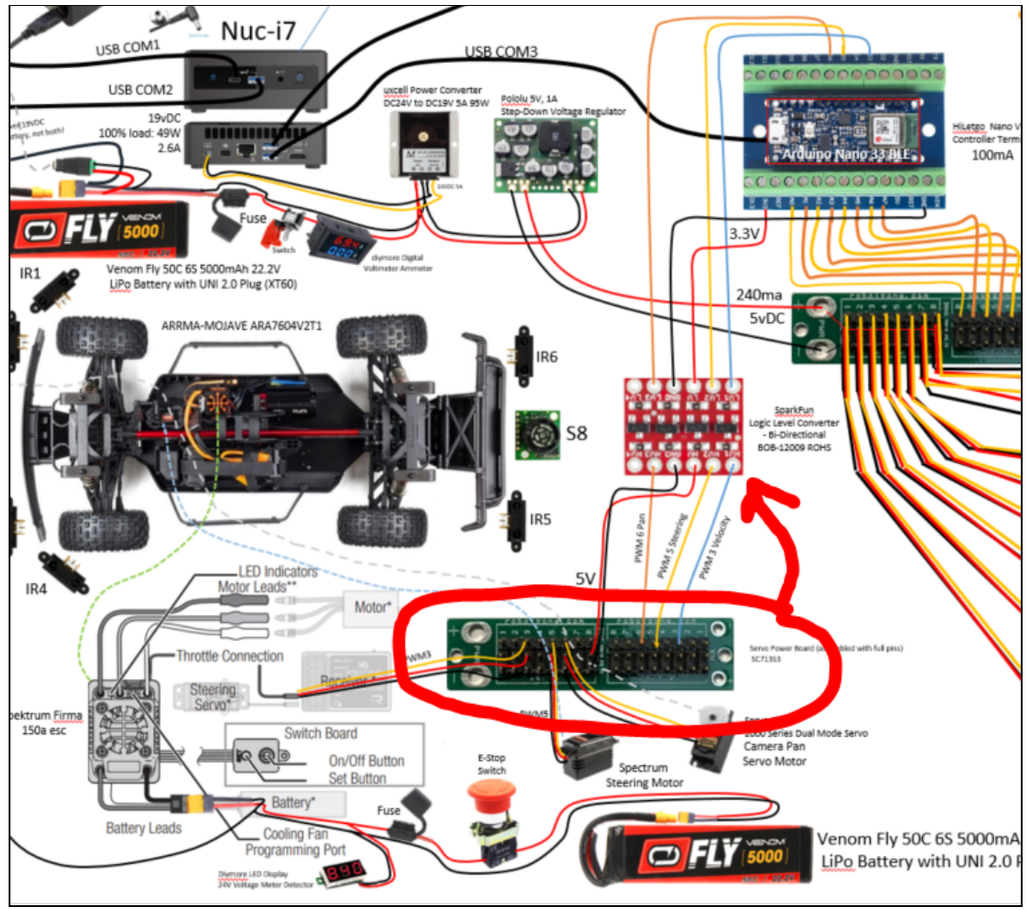
NOTE: Arduino Nano 33 BLE only supports 3.3V I/Os and is **NOT** 5V tolerant so please make sure you are not directly connecting 5V signals to this board or it will be damaged. Also, as opposed to Arduino Nano boards that support 5V operation, the 5V pin does NOT supply voltage but is rather connected, through a jumper, to the USB power input.

ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

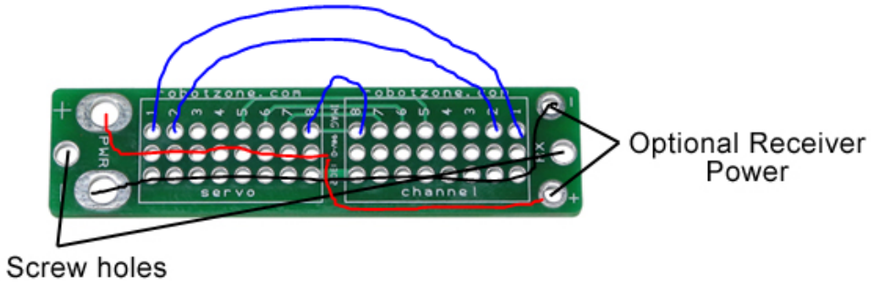
A well designed robot usually has two separate DC power systems; one clean one to support sensors, control signals and processors and one “dirty” (a synonym for electrically noisy) one to supply power to motors. In order to handle this efficiently in your robot and to facilitate connection between these two systems we will be using a set of ServoCity **Servo Power boards**. The first supports the raft of simple 5VDC sensors that your rover uses as the Arduino Nano is running at 3.3 VDC and can’t source much current:



The second allows the 3.3VDC Nano to command via PWM signals the Rover drive motor, steering RC-Servo and the Camera Pan-Servo:

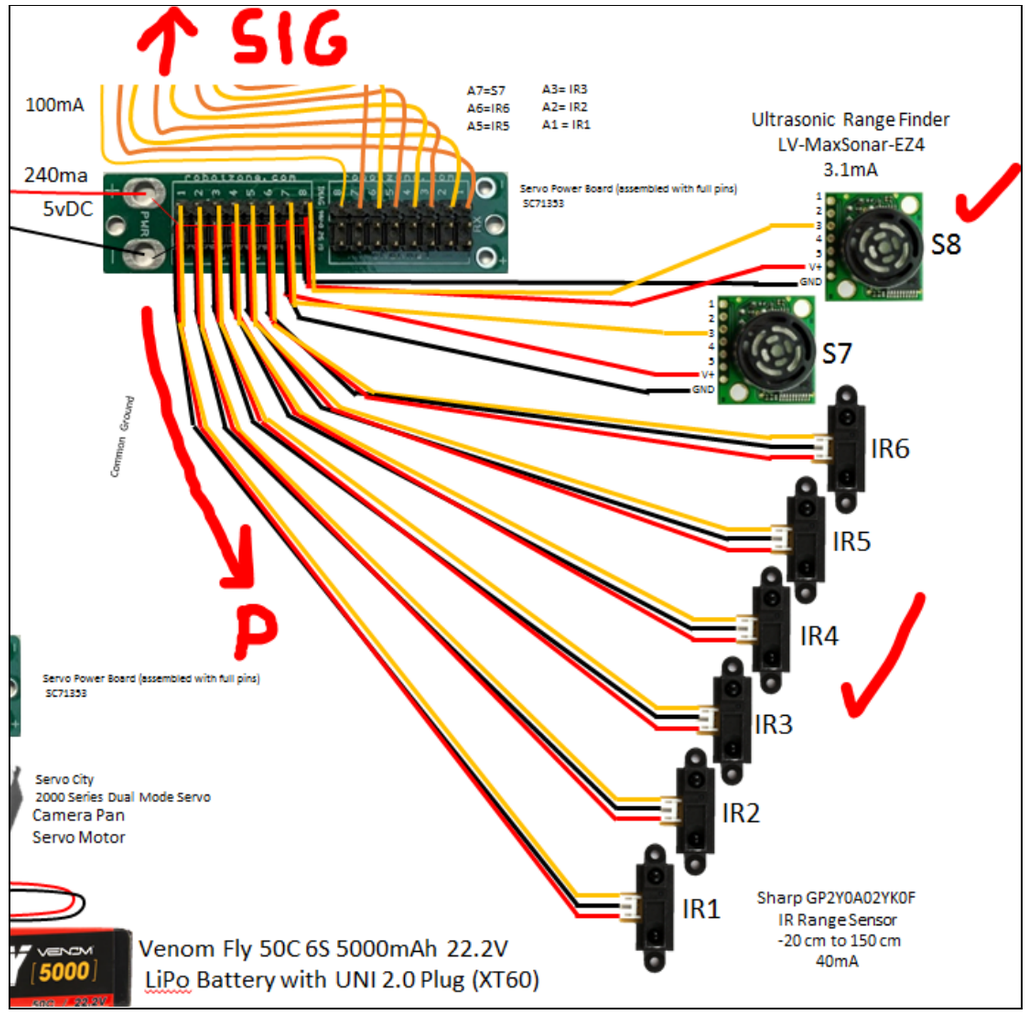


See details at:
https://www.servocity.com/servo-power-board-assembled-with-full-pins/
You may well ask how these **Servo Power** boards work:



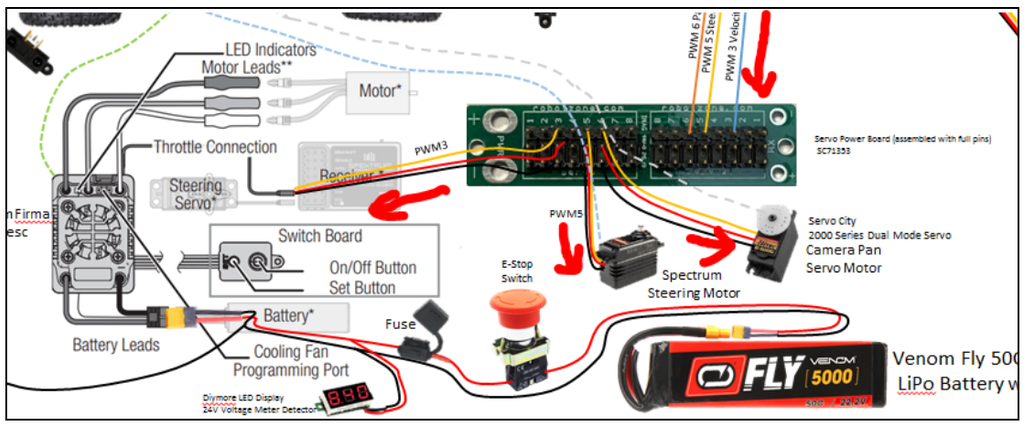
They are elegantly simple. Many simple robots use a three wire connection for sensors and RC-Servo motors, the **red** wire is 5V, the **yellow** or **white** wire carries signal and the **black** wire is ground:

The **Servo Power** boards enable connecting clean signals to a dirty power bus by providing two sets of matching three pin male connectors, side-by-side, where the dirty power is only supplied to the left hand set and the right hand power pins are unconnected. Board traces connect all of the ground pins to ensure a common ground, and individually connect each labeled pin with its cousin on the other side of the board. In our case we can have high amperage, dirty power on the left side of the board and clean 3.3v, low-amp signals on the right all with respect to the same reference ground. This makes it fast and easy to use standard three wire servo cables to hook up both the Sharp IR range sensors and the Maxbotix Sonars as shown below:

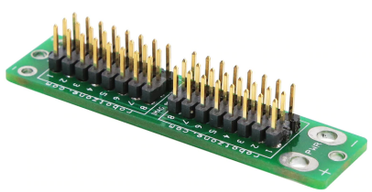


Power flows to sensors from **Servo Power** board over easy to debug servo cables, Analog signal data flows back through those same cables, over the cousin connection of the board to pins connected back to the Arduino.

In a similar fashion 3.3v low power PWM position and velocity commands flow from the Arduino to the second **Servo Power** board which supplies propulsion battery power to the Steering Servo, the Drive Motor and the Pan Servo:



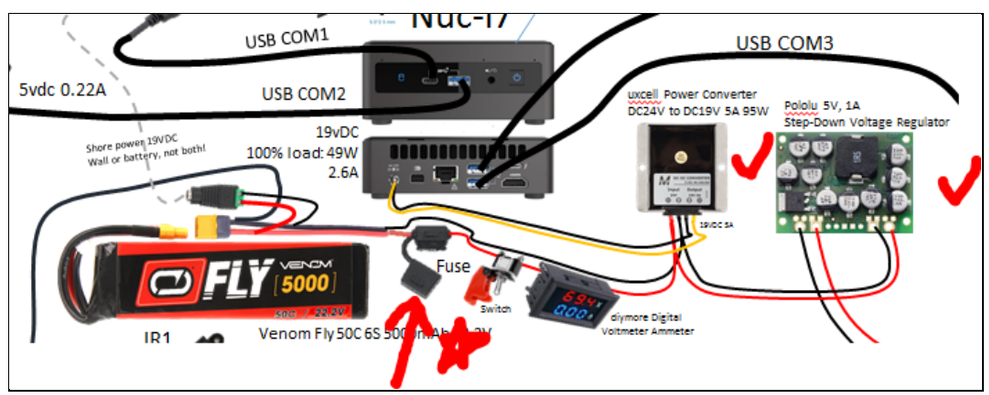
Servo Power Distribution Board



The Servo Power Board allows you to cleanly isolate high current and voltage from your controller to protect the tiny traces within from burning out. One side of the board shares a direct connection with your servo controller while the other allows servos (up to 8) to be plugged in. The positive trace is split between the servo controller side and the servo side so that you can supply power to the solder pads without sending that power on to the servo controller. This is advantageous in setups where the servos require a different voltage than the robot controller. This setup will protect the controller from the current drawn by the servos as the current will travel between the battery and the servos and not through the controller.

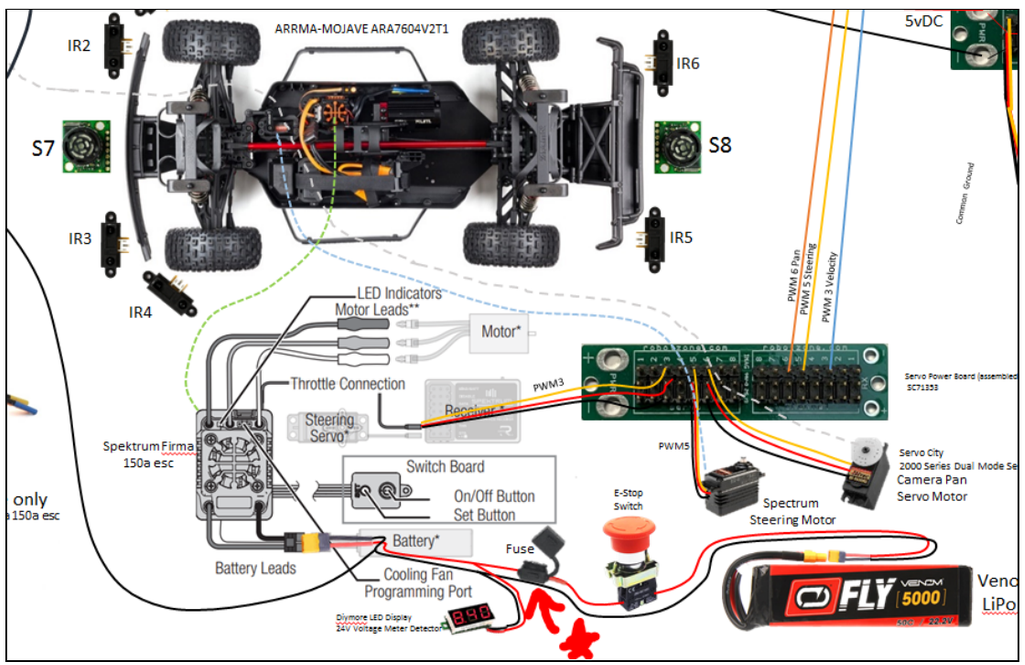
Robot Controller Power System

Because of the choice of controllers used, this rover has a somewhat complex power system. The NUC uses processors designed for 19V laptop computers, the Arduino is designed to run off 5V USB ports on the laptop. To wire all that up in a mobile robot, we start with a 22.2V lithium battery that runs first through a fuse (**always always always have a fuse, right next to the lithium battery on any robot you build in the future. If you don't, a technician will short it, your robot will explode and your company will be asked to move out of Cambridge -true story!**). After the fuse is an on/off switch, a simple Voltage-Current meter (so you can see how much gas is left in the tank and how fast you are burning it) and then the 22.2V power goes to two DC to DC down converters. The left one converts the slowly dropping 22.2V battery power to a stable 19V to power the NUC. The second converter drops the 22.2V down to a stable 5V to power the Sharp IR and Maxbotix Sonar sensors. As previously stated, the **Servo Power Boards** let you safely connect the grounds and signals from these separate rovers power buses..



Rover Actuator Power System

The rover power system heavily uses the existing Mojave power wiring. Here Lithium battery power flows through a fuse (always a fuse!), an E-Stop (on-off) switch, a battery voltage meter (so you can check gas in your other tank), and then to the Mojave's existing Spektrum Firma speed controller-esc. This controller both supplies the 2 quadrant speed control to the main 4WD propulsion motor and it down-converts the 22V to 6V to drive the steering and pan servos.

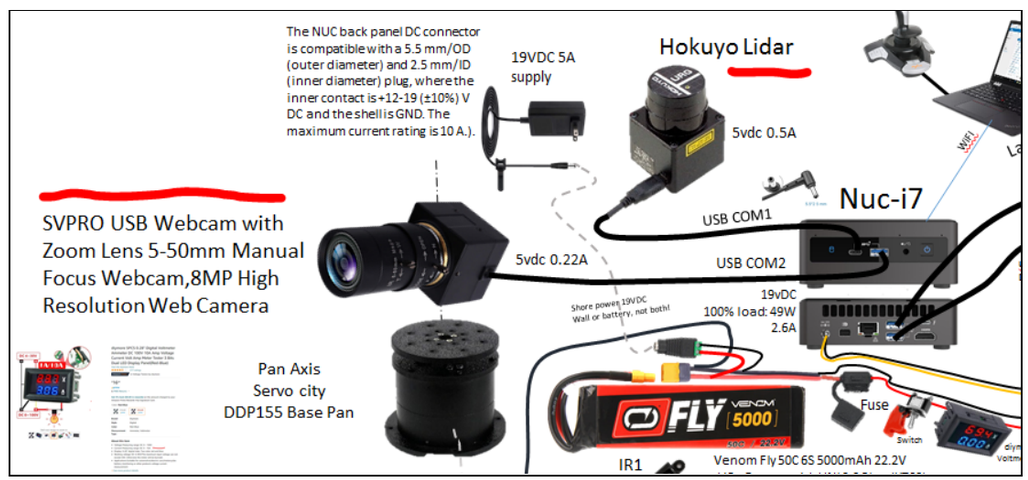


Using separate power buses keeps your Arduino and Nuc safe from over-voltages and power spikes and allows you to run larger motors at higher power levels. Note we will use port 5 for steering servo, port 3 for drive servo and port 6 forthe pan servo.

Advanced Sensor Suite

To provide your Rover with machine vision, precision navigation depth ranging and accurate location sensing; we will add a professional grade USB Web-Camera, a Hokuyo Lidar and a uBlox GPS to it. As the webcam and lidar sensors were well covered in the foundation SENSE lab, no further technical data will be provided here. You can use the best version of your teammates sense lab code as a starting point for your rovers sensing code. Please consult the Sense lab tutorial as needed.

The WebCam and Lidar are connected to NUC as shown::

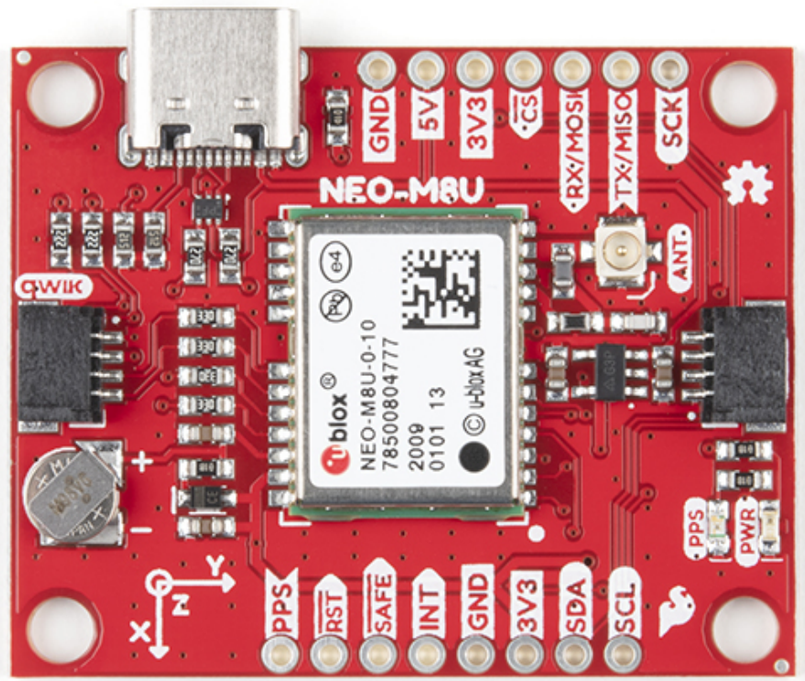


SparkFun Ublox GPS

For your Rover, your team will be supplied with a quite capable SparkFun **GPS Dead Reckoning Breakout - NEO-M8U**

See for details: https://www.sparkfun.com/products/16329

The SparkFun NEO-M8U GPS Breakout is a high quality GPS board with equally impressive configuration options. The NEO-M8U takes advantage of u-blox's Untethered Dead Reckoning (UDR) technology. The module provides continuous navigation without needing to make any electrical connection to a vehicle, thus reducing cost of installation for after-market dead reckoning applications.



The NEO-M8U module is a 72-channel u-blox M8 engine GNSS receiver, meaning it can receive signals from the GPS, GLONASS, Galileo, and BeiDou constellations with ~2.5 meter accuracy. The module supports concurrent reception of three GNSS systems. The combination of GNSS and integrated 3D sensor measurements on the NEO-M8U provide accurate, real-time positioning rates of up to 30Hz.

Compared to other GPS modules, this breakout maximizes position accuracy in dense cities or covered areas. Even under poor signal conditions, continuous positioning is provided in urban environments and is also available during complete signal loss (e.g. short tunnels and parking garages). With UDR, position begins as soon as power is applied to the board even before the first GNSS fix is available! Lock time is further reduced with on-board rechargeable battery; you'll have backup power enabling the GPS to get a hot lock within seconds!

Additionally, this u-blox receiver supports I2C (u-blox calls this Display Data Channel) which made it perfect for the Qwiic compatibility so we don't have to use up our precious UART ports. Utilizing our handy Qwiic system, no soldering is required to connect it to the rest of your system. However, we still have broken out 0.1"-spaced pins in case you prefer to use a breadboard.

U-blox based GPS products are configurable using the popular, but dense, windows program called u-center. Plenty of different functions can be configured on the NEO-M8U: baud rates, update rates, geofencing, spoofing detection, external interrupts, SBAS/D-GPS, etc. All of this can be done within the SparkFun Arduino Library!

The SparkFun NEO-M8U GPS Breakout is also equipped with an on-board rechargeable battery that provides power to the RTC on the NEO-M8U. This reduces the time-to-first fix from a cold start (~26s) to a hot start (~1.5s). The battery will maintain RTC and GNSS orbit data without being connected to power for plenty of time.

The u-blox NEO-M8U is a powerful GPS units that takes advantage of untethered dead reckoning (UDR) technology for navigation. The module provides continuous positioning for vehicles in urban environments and during complete signal loss (e.g. short tunnels and parking garages). We will quickly get you set up using the Qwiic ecosystem and Arduino so that you can start reading the output!

Dead Reckoning is the process of determining current position by combining previously determined positional data with speed and heading. This process can also be applied to determine future positions as well! The NEO-M8U uses what is called Untethered Dead Reckoning (UDR) which calculates speed and heading (amongst many other points of data) through the use of an internal inertial measurement unit (IMU). The addition of an IMU allows the M8U to produce more accurate readings in between GNSS data refreshes!

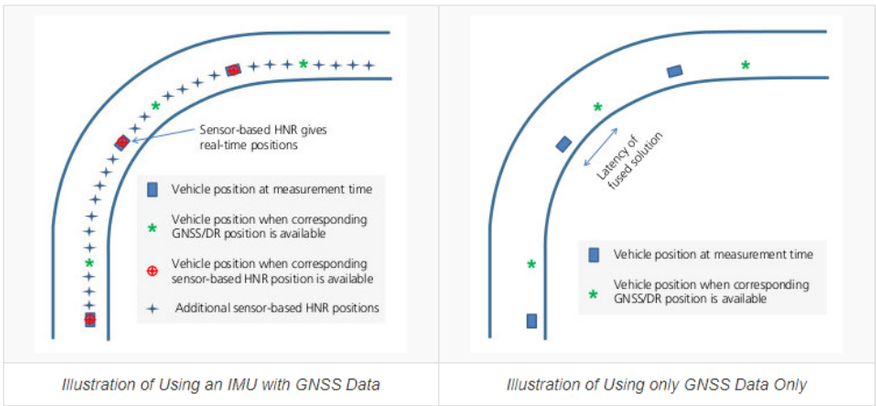
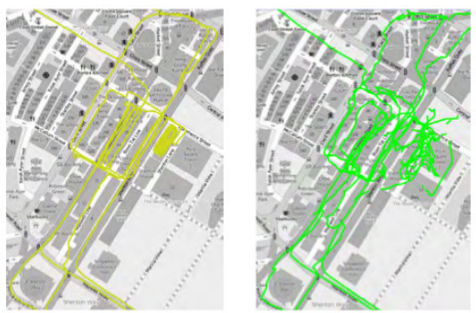


Illustration of Using an IMU with GNSS Data | *Illustration of Using only GNSS Data Only*

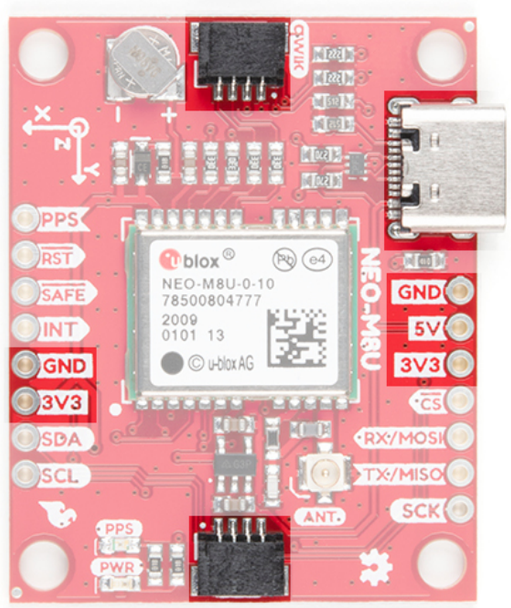
In addition, the module can also give accurate and useful GNSS data in areas where satellite connections are difficult to maintain: areas like the dense urban environments of major cities, long tunnels, parking garages, any large UFO's that may descend from the sky, etc.



UDR | GNSS only

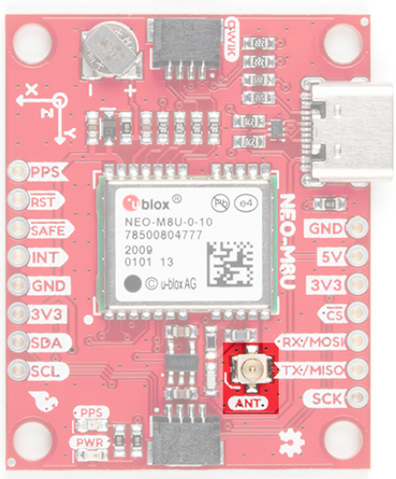
Untethered Dead Reckoning vs GNSS Only Comparison in an Urban Canyon. Image Courtesy of u-blox from the UDR Whitepaper.

Power for this board is 3.3V and we have provided multiple power options. This first and most obvious is the USB-C connector. Secondly, are the Qwiic Connectors on the top and bottom of the board. Thirdly, there is a 5V pin on the PTH header along the side of the board that is regulated down to 3.3V. Make sure that power your provide to this pin does not exceed 6 volts. Finally, just below the 5V pin is a 3.3V pin that should only be provided with a clean 3.3V power signal.

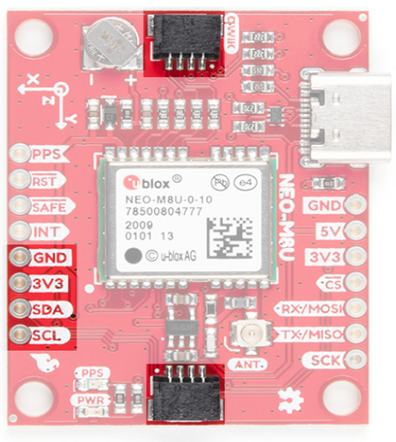


There is a red power LED just to the left of the bottom Qwiic connector and near the board's edge to indicate that the board is powered. There is another LED just above the power LED labeled PPS that is connected to the Pulse Per Second line. When connected to a satellite, this line generates a pulse that is synchronized with a GPS or UTC time grid. By default, you'll see one pulse a second.

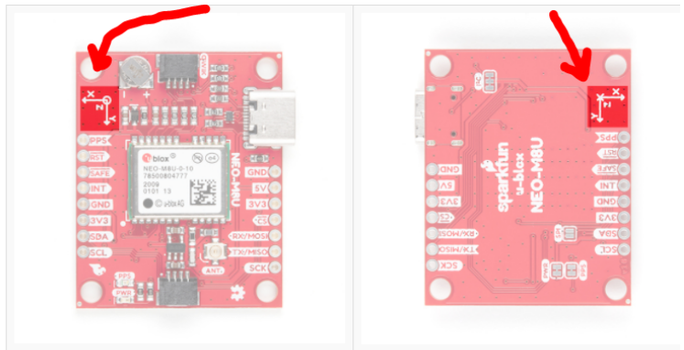
The SparkFun GPS NEO-M8U has a u.FL connector in which you can connect a patch antenna. Be really really careful plugging this in. Can destroy a board if you do it wrong.



There are two pins labeled SDA and SCL which indicates the I2C data lines. Similarly, you can use either of the Qwiic connectors to provide power and utilize I2C. The Qwiic ecosystem is made for fast prototyping by removing the need for soldering. All you need to do is plug a Qwiic cable into the Qwiic connector and voila!



For easy reference, we've documented the IMU's vectors with 3D Cartesian coordinate axes on the top and bottom side of the board. Make sure to orient and mount the board correctly so that the NEO-M8U can accurately calculate navigation information. This is explained in detail in the Dead Reckoning Overview. Remember, it's all relative.



Top View with the the Axis for Reference

Bottom View with the the Axis for Reference

GPS Capabilities

The SparkFun NEO-M8U is able to connect to up to three different GNSS constellations at a time. Below are the listed capabilities of the GPS unit taken from the datasheet when connecting to different GNSS constellations.

Constellations		GPS+GLO	GPS	GLO	BDS	GAL
Horizontal Position Accuracy	Autonomus	2.5m	2.5m	4.0m	3.0m	To Be Confirmed
	with SBAS	1.5m	1.5m			
Max Navigation Update Rate	PVT	25Hz	25Hz	25Hz	25Hz	25Hz
Time-To-First-Fix	Cold Start	24s	25s	26s	28s	29s
	Hot Start	2s	2s	2s	2s	2s
Sensitivity	Tracking and Navigation	-160dBm	-160dBm	-160dBm	-160dBm	-154dBm
	Reacquisition	-160dBm	-159dBm	-156dBm	-155dBm	-152dBm
	Cold Start	-148dBm	-147dBm	-145dBm	-143dBm	-133dBm
	Hot Start	-157dBm	-156dBm	-155dBm	-155dBm	-151dBm
Velocity Accuracy		0.05m/s	0.05m/s	0.05m/s	0.05m/s	0.05m/s
Heading Accuracy		1deg	1deg	1deg	1deg	1deg

Dead Reckoning Overview

As mentioned in the "What is Dead Reckoning?" section, the u-blox M8U module has an internal inertial measurement unit or IMU for short. The IMU calculates position based on the last GNSS refresh and its own movement data points. To use the SparkFun GPS Dead Reckoning Board, there are a few guidelines to orienting and mounting the module to a vehicle outlined in the u-blox ReceiverDescrProtSpec Datasheet.

Orientation for the SparkFun Dead Reckoning

The SparkFun Dead Reckoning adheres to two particular frames of reference: one frame of reference for the car and the second a geodetic frame of reference anchoring it to the globe. The latter, known as the local level frame uses the following as its' axes:

- X-axis points to the North
- Y-axis points to the East
- Z-axis uses the right hand system by pointing down.

This frame will be referred to by its acronym NED (North-East-Down) in the image below.

The second frame of references is the Body-Frame reference and uses the following as its' axes.

- X-axis points to the front of the vehicle
- Y-axis points to the right of the vehicle
- Z-axis uses the right hand system by pointing down.

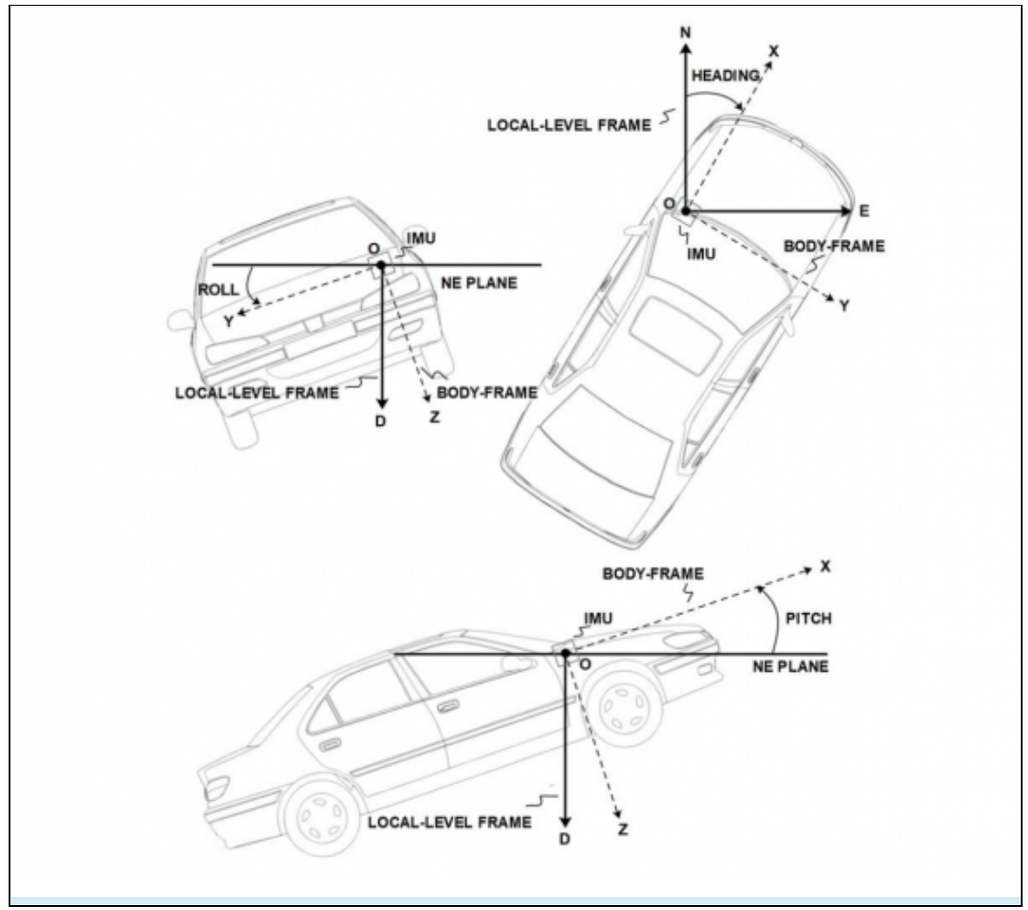
The transformation of the vehicle within these two frames are given as heading, pitch, and roll. In the datasheet these three angles are called the vehicle's attitude. Below is an image that illustrates how all of these elements fit together.

Calibration

After you've mounted the SparkFun Dead Reckoning M8U, there is still a calibration phase to complete that must satisfy the following movements:

- First, the car needs to be stopped with the engine turned on.
- Secondly, the car must do left and right hand turns.
- Lastly, the car must reach a speed over 30 km/h.

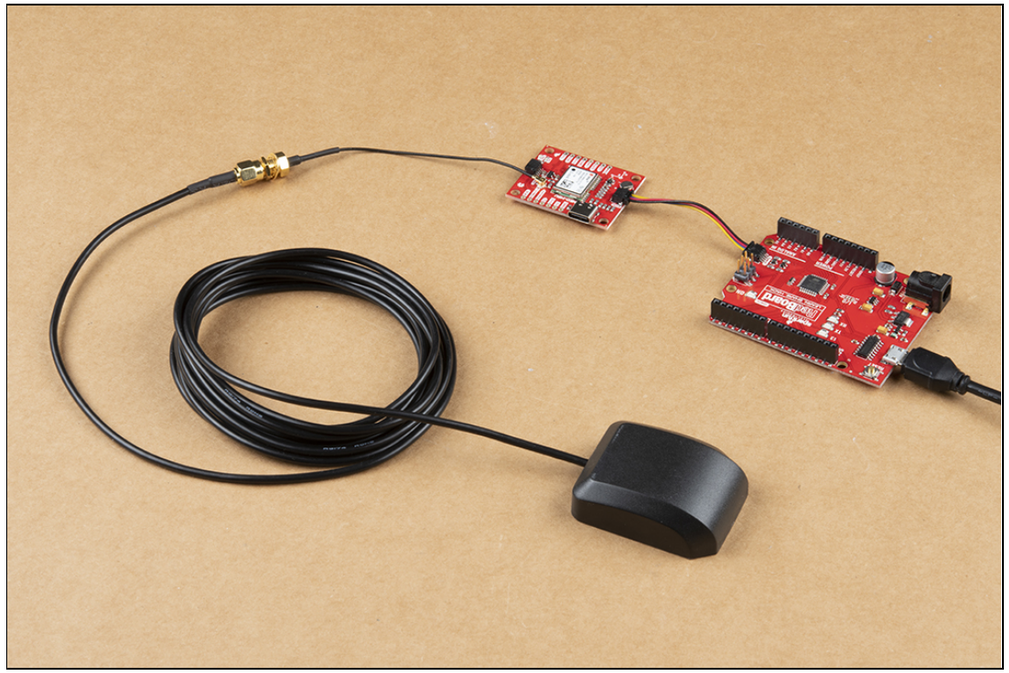
In SparkFun's u-blox Arduino library, SparkFun has included an Example (shown below), that prints out the module's calibration status.



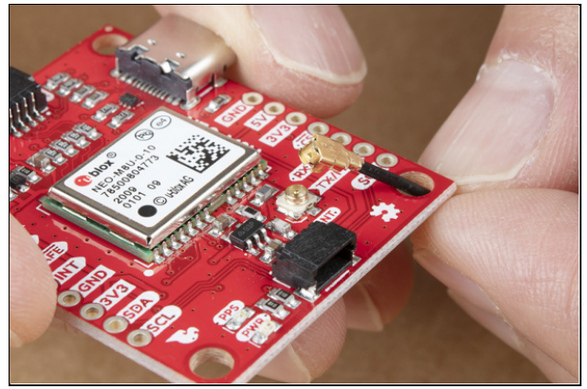
The only guideline here is that the SparkFun Dead Reckoning is stable within 2 degrees, and of course that the X-axis points towards the front of the car as mentioned above.

Hardware Assembly

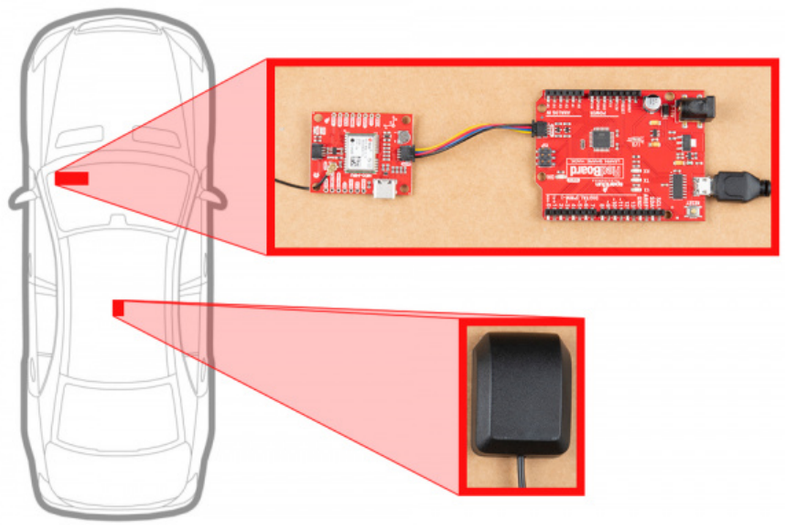
For this example, we used a RedBoard Qwiic and associated USB cable. Connecting the boards with Qwiic cable, the assembly is very simple. Plug a Qwiic cable between the RedBoard and SparkFun NEO-M9U. Then plugged in one of our patch antennas to the u.FL connector. If you need tips on plugging in the U.FL connector, then check out our U.FL tutorial. If you're going to be soldering to the through hole pins for I2C functionality, then just attach lines to power, ground, and the I2C data lines to a microcontroller of your choice. Your setup should look similar to the image below.



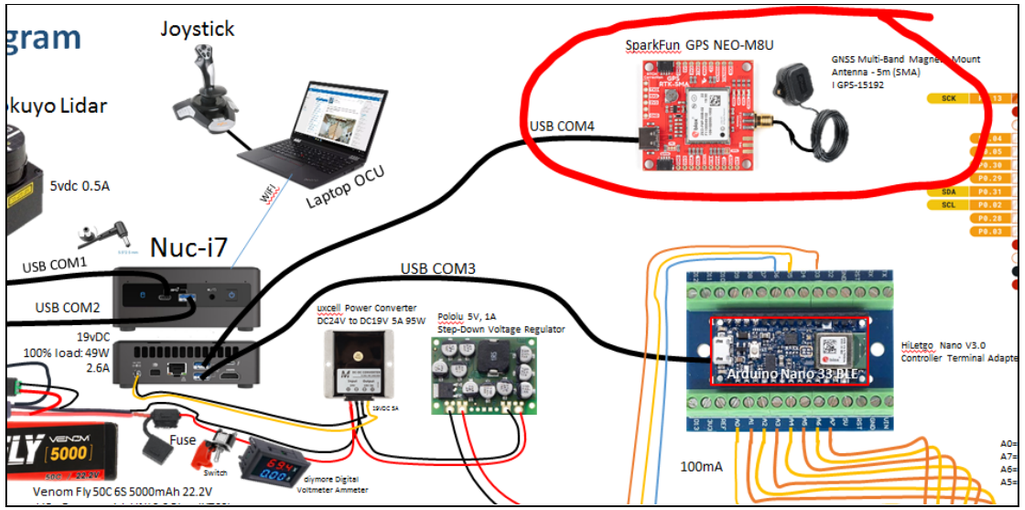
For secure connections, you may want to thread the U.FL cable through a mounting hole before connecting. Adding tape or some hot glue will provide some strain relief to prevent the cable from disconnecting.

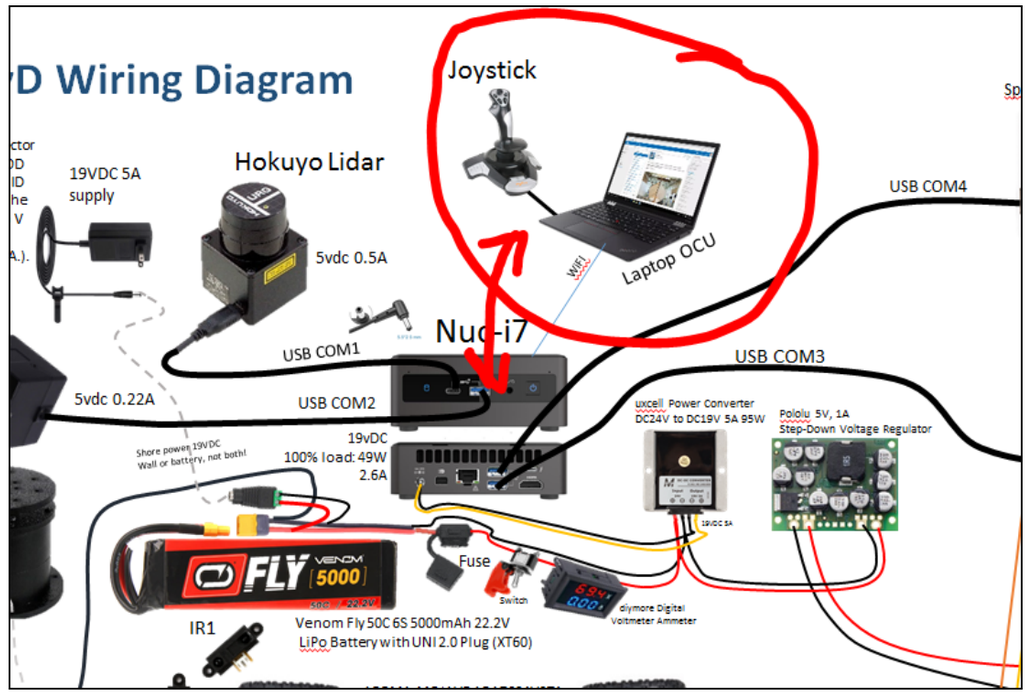


Make sure to secure the board above a vehicle's dashboard using some tape or sticky tack when prototyping and testing. For best signal reception, it is suggested to guide the antenna from the inside of the car and through a window before attaching the GPS on top of a car.



Finally, to add this GPS to your rover, plug it into the last port on your NUC as shown:





Student Laptop Operator Control Unit (OCU)

Finally, you will both develop software for and command your rover from a suitably configured student laptop operator control unit (commonly called an OCU) over Olin's dedicated Robot WiFi Network.

To facilitate testing and just for the sheer fun of it, your team will also be equipped with a multifunctional joystick to enable both manual operation for testing and perhaps an emergency backup if things go terribly wrong during testing. Setting up a joystick to remotely E-Stop rover, should be one of the very first things your team does after completing the wiring.

Software Design

On first pass, it might look like there is an overwhelming amount of code to be written for this final project, but take heart, you have already written most of what you will need in the previous tutorials and labs and can just refactor your code and reuse it here in your actual rover. In your very first Matlab for Robots tutorial, you simulated a rover driving between waypoints in the Olin oval, you have already used the Web-Cam, Lidar, Sharps and Sonars to find targets or holes, written whale tracking behaviors and you have already mastered Pan-Tilt servos and driven drive wheels. This final project code will have your team take pieces from your existing work and then integrate all of them together into one working robot controller for a smoothly functioning robot planetary rover.

Overall Code Architecture

To maximize your team's code development efficiency, you will want to think seriously about reusing the first tutorial's **Oval-Rover code** as your team's main architectural structure, add the **Think lab's** behavior engine to it and then break down all of the remaining code into well defined functions that individual students can write. Good team code is modular. Good team code uses one small main program and lots of modular individual functions that can be developed independently.

If you put a simple main robot control structure in place first, then break up the remaining function writing evenly among your team, you will succeed. **If you do almost anything else, your team will fail**. *Specifically, if you listen to, blindly follow the loudest E:C super-programming wizard on your team, your project will almost surely go down in flames.* Professional code creation is team creation; simple, clean, written by all and understandable by all.

Let's talk a little about the robot control structure your team should consider using. You have in fact been using it for the whole first part of this course. **The first part of your structure is a documentation paragraph** that describes what your main code does, in enough detail that a cogent fellow programmer can follow it. To make field editing code easy, it should also include a wiring diagram of your robot and a clear indication of what external Matlab code parts (functions, files, objects, etc.) need to be in the same directory folder and on the same path as your robot's control code.

Next should be a clearly delineated section that contains all of **the code that is run, just once**, to set up your Rover's, data-structures, variables, etc. This would also be a good place to download the current mission parameters and set up the main mission data (like waypoints, active behaviors, etc).

Following the code that runs once, comes **the main Sense-Think-Act control loop that runs over and over.** This is where the behavior engine consisting of a set of mission specific behaviors and an arbiter for the vehicle and perhaps a separate one for the pan-tilt head are situated.

After the body of text that holds the main code in the loop that runs over and over, your team should have at least **three function repositories**, one for Sense functions (i.e. ReadSharpIR, etc.), one for Think functions (GoToWaypoint etc.) and one for Act functions (PanTilt(angle, angle).

Please note, your team can lay out the full structure of your robot rovers control code in a single meeting and then put in place a set of empty Sense, Think and Act functions with jointly agreed on input and output arguments. Having done this you can break into subteams and asynchronously write, debug and test those functions using a common team Matlab-Drive code repository. Avoid what happened during the Think lab; organize in modules first, code second.

By design, you and your programming partners have been doing this process all semester. There is a common robot control template beneath all of the labs and you can now go back and harvest the Sense, Think, and Act functions you created during those labs for reuse here in your final project rover.

In addition to the code you have generated in your Sense-Think-Act labs this semester, there are a few other code modules in the Sense space that will really help your Rover succeed in the final project. We will describe them in detail in the following sections.

Sense: Better Sharp IR Range Code

Your Rover will have a suite of Sharp IR Range sensors on it. Starting with your existing Sense-Lab code, you will want to add a bit more raw data and collection code to deal with the fact that the Sharp IR output is actually a somewhat noisy analog waveform:

As you can see on the above scope trace, when a target distance changes, there is an initial high spike followed by some clear second order ringing. If you sample during this transient, your actual ranges could be way off. In practice to get clean and reliable range data with a moving robot constantly changing the ranges, you might want to collect multiple range voltages, quickly and sequentially and average them before using your calibration code to convert that voltage to a usable range. More sophisticated filtering and processing will give you even better range data.

Sense: Red Circle Dock Detection

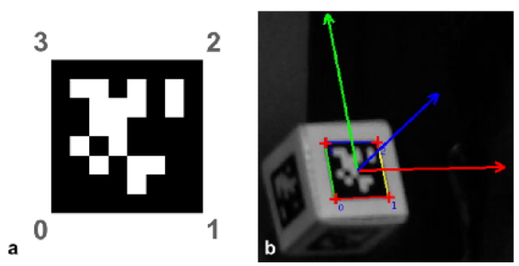
Your Rover will have a quite nice USB camera with a very nice telephoto lens on it, most probably mounted on its Pan head. The USBCam you are using is similar to, but much better than the one on the Sense Lab test stand. It can be used to find the big red circles used to designate locations on the race track. You will probably need to frequently calibrate the color mask used to do so (based on lighting just before an experimental run). You could even put a small red circle on your rover's GPS tower and build in an auto-calibration procedure to save yourself some work. Just have your Rover auto-calibrate red circles before each run. Once you have solid calibration code in place, you can write a single function that the main control program can call, once each loop cycle, to scan around rover to see if a circle is within sight of Rover.

The function can find heading (or bearing to) to the circle using the pan angle of the camera and can estimate distance from the diameter of the circle (little circle, far away, big circle close, linear function based on measured circle vertical diameter gives elegant range data). Your team can design this function any way you'd like, but a classic way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the red circles location to the grid, then have function return the updated occupancy grid to the main program. The more computation you can do locally in a function, the more reliable your main code will be. Your Find Circle function could look something like this:

```
function [roverOcupancyGrid, circleBearing, circleRange] = redCricleFind (roverOcupancyGrid)
% redCircle scans pan space for red circle. Finds circle and adds the current circle
% location to  Rover centric occupancy grid. 0=open space 1-10= obstacles
% 20 = dock
% written by Mary and Beth 3-30-21 Rev A
    Body of code
end
```

Sense: April Tage Range and Direction

Your Rover will have a professional USB Camera, most probably mounted on the Pan-Unit. Its main function is to find the big red circles at track navigation points. But you can also use it to get more precise range and bearing data by looking for April Tags:



With red circles you will need to frequently calibrate the color mask used to do color blob tracking outdoors in highly variable real-world lighting; this can pose a significant challenge. To overcome this, April tags were invented to provide a relatively lighting invariant way of getting accurate range and bearing to a visual fiducial target. April tags are a little complex, but fortunately Matlab has built in April tag support in its Vision toolbox. We will provide you with some April tag starter code (see Canvas) and strongly recommend that your team explore using them.

You can download the April tags used in this project here:

https://www.dotproduct3d.com/uploads/8/5/1/1/85115558/apriltags1-20.pdf

And full documentation for Matlab's built in April Tag support can be found here:

https://www.mathworks.com/help/vision/ref/readapriltag.html

Once you have a solid tag finding code in place you can write a single function that the main control program can call, once each loop cycle, to accurately see if a specific April Tag is within sight of the rover. Your team can design this function any way you'd like, but a simple way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the april tag to that grid, then have function return the updated occupancy grid to the main program. The more computation you can do locally in a function, the more reliable your main code will be. Your findAprilTag function could look something like this:

```
function [roverOcupancyGrid, tagBearing, tagRange] = findAprilTag (roverOcupancyGrid)
% findAprilTag scans the space around the rover and  adds any April Tag found to
%the rover  centric occupancy grid. 0=open space 1-10= obstacles
% 20 = dock, 40 = April Tag
% written by Agash and Fred 3-30-21 Rev A
    Body of code
end
```

Sense: Hokuyo Lidar Obstacle Detection

Your SharpIR range sensor will give you very sparse range data at 5 discrete points off the front of your Rover. The two Sonars will give you longer range, but fuzzier coverage 360 around Rover if you mount them on a pan-tilt head. But neither will give you really detailed data about what is immediately in front of the rover, which may be critical to both finding a clear path and not hitting any obstacles..

The Hokuyo Lidar can do a very precise distance ranging across its full field of view depending on how you mount it on your rover. If it's tipped down at a slight angle while mounted on the front of the rover it can sweep the entire drivable space as you move forward and that can be used to make a very detailed terrain map to add in local navigation.

You can use this to get a precise 2d distance map of what is in front of your rover across the camera's full field of view. It will give you much more detailed and much more useful information than the array of fixed IRs can. It would give your rover a decided advantage in the final race, especially in the area of precise positioning.
.

Once you have a solid set of lidar code you can write a single function that the main control program can call, once each loop cycle, to get really detailed data on what is in front of the rover. Your team can design this function any way you'd like, but a simple way to do this is to pass a vehicle centric simple occupancy grid (matrix) to the function as an input parameter, use the body of the function to add the laser range data to that grid, then have function return the updated occupancy grid to the main program. The more computation you can do locally in a function, the more reliable your main code will be. Your scanLaser function could look something like this:

```
function [roverOcupancyGrid] = scanLaser (roverOcupancyGrid)
% scanLaser adds PiCam lidar range data to
% loop cycle's Rover centric occupancy grid. 0=open space 1-10= obstacles
% 1-3 drive over, 4-10 drive around
% 20 = dock
% written by Mary and Beth 3-30-21 Rev A
    Body of code
end
```

Sense: GPS Position and Heading

Your Rover will have a very sophisticated SparkFun-Ublox GPS sensor to give you both the Rover's location and orientation. Setting it up is a little complicated, so we will provide you with that function, and use instructions that you can download from the Canvas course website.

Sense: IMU Position and Orientation

Your Rover's Arduino Nano 33 BLE will have a very sophisticated internal Inertial Measurement Unit sensor to give you both the Rover's pitch-roll-yaw and magnetic orientation (heading with respect to magnetic North). The heading part alone could be used with some simple time based dead reckoning to get your rover safely around the course. It also can give you a ground truth rover heading to use as a baseline for object recognition with the pan unit. Simply put, if you know your heading and you can take two fixes on two april tags, you can back calculate your exact position in the Oval. This would be quite useful for bridge crossing. Fortunately, Matlab natively supports its functionality, please see:

https://www.mathworks.com/help/supportpkg/arduinoio/ug/Nano33BLEsense_sensorconnections.html

This example shows how to use the MATLAB® Support Package for Arduino® Hardware to connect an Arduino Nano 33 BLE Sense board over Bluetooth® with an LSM9DS1.

The sensor is connected to the Arduino board over an I2C interface.

Overview

The `lsm9ds1` sensor is a nine degrees of freedom (DOF) inertial measurement unit (IMU) used to read acceleration, angular velocity, and magnetic field in all three dimensions.

ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B

The sensor object represents a connection to the device on the Arduino hardware I2C bus. The LSM9DS1 sensor is present on the Nano 33 BLE Sense board. You can read the data from your sensor in MATLAB using the object functions.

You can also use other sensors like `hts221`, `lps22hb`, or `apds9960` on the Arduino board to connect to MATLAB over Bluetooth.

To connect the Arduino board to MATLAB over Bluetooth, type `arduinosetup` in the MATLAB command line and set the connection type as Bluetooth. After you select the board and the port, in the **Upload Arduino Server** window, select **I2C**. For the complete workflow, see Connection over Bluetooth.

Connect to Sensor and Read Sensor Output

Create an Arduino object.

```
a = arduino
```

```
a =
  arduino with properties:

               Address: 'DEBD293F7CDC'
                  Name: 'LSMOverBLE'
             Connected: 1
                 Board: 'Nano33BLE'
        AvailablePins: {'D0-D13', 'A0-A7'}
  AvailableDigitalPins: {'D0-D13', 'A0-A7'}
      AvailablePWMPins: {'D0-D13'}
   AvailableAnalogPins: {'A0-A3', 'A6-A7'}
    AvailableI2CBusIDs: [0, 1]
AvailableSerialPortIDs: [1]
             Libraries: {'I2C', 'SPI', 'Servo'}
Show all properties
```

Create the sensor object. The LSM9DS1 sensor is connected to the I2C bus `1` of the Nano 33 BLE Sense board.

```
lsmObj = lsm9ds1(a,"Bus",1)
```

```
lsmObj =
  lsm9ds1 with properties:

                          I2CAddress: 107 ("0x6B")
                                    : 30 ("0x1E")
                                 Bus: 1
                             SCLPin: "SCL1"
                             SDAPin: "SDA1"
Show all properties all functions
```

Read acceleration.

```
acceleration = readAcceleration(lsmObj)
```

```
acceleration = 1×3

    2.6315   -7.8341    5.6433
```

This function returns one sample of acceleration data. For more information, see `readAcceleration`.

Read angular velocity.

```
angularVelocity = readAngularVelocity(lsmObj)
```

```
angularVelocity = 1×3

    0.0053    0.0057    0.0078
```

This function returns one sample of angular velocity data. For more information, see readAngularVelocity.

Read magnetic field.

```
magneticField = readMagneticField(lsmObj)
```

```
magneticField = 1×3

   26.1100   27.0620   14.0140
```

This function returns one sample of magnetic field data. For more information, see readMagneticField.

Clean up

Once the sensor connection is no longer needed, clear the associated object.

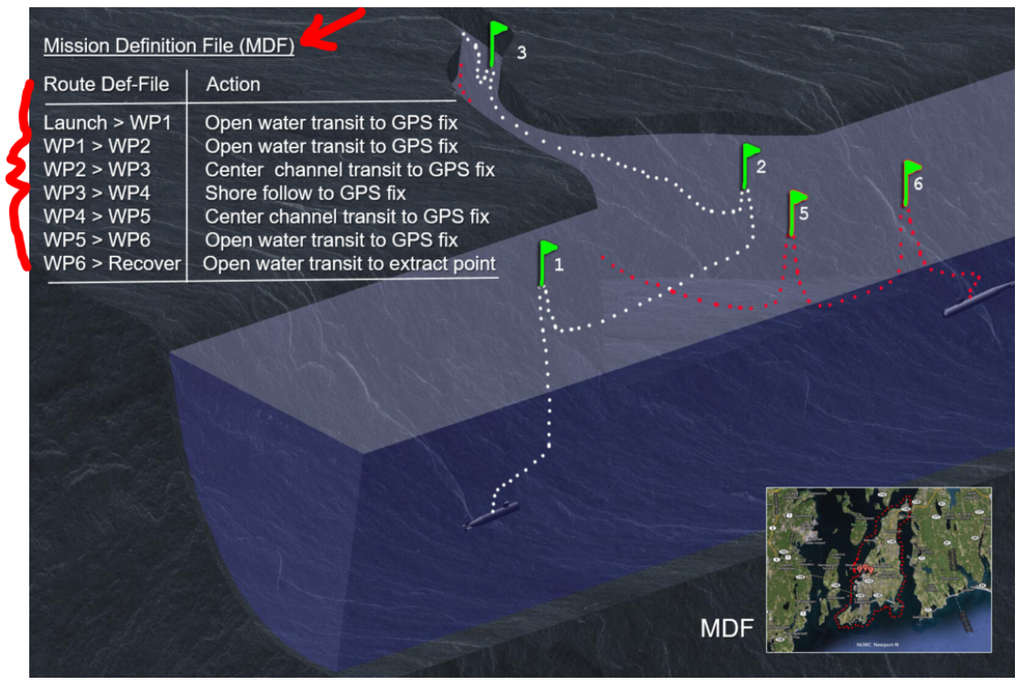
```
clear lsmObj a
```

Sense code wrap up

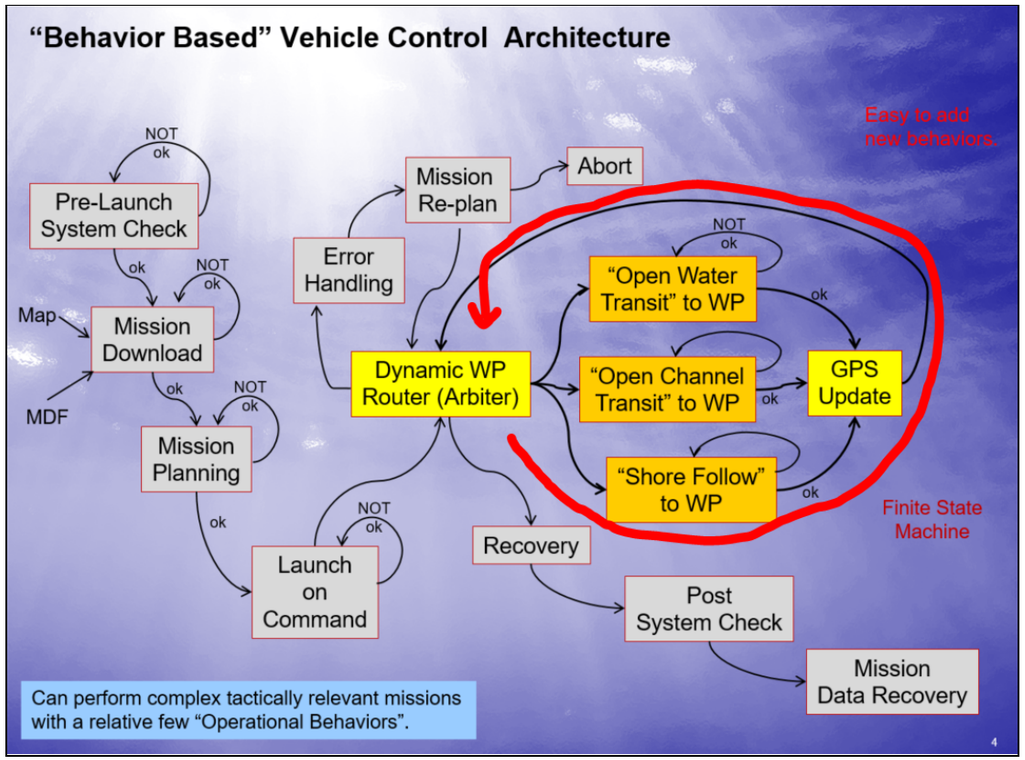
In addition to enhancing your robots Sense code, it is also useful to take a second, more organized pass through your Team's think code generation process. Part of the reason you do the Tugboat lab is to both experience what coding is like on a big team firsthand and to realize that you probably don't want to do what you did on the Tug as far as group code generation twice (i.e. "it's only a mistake if you do it twice!"! Please take a look at the following sections to help develop a more organized and more robust set of Think functions.

Think: Mission Planner and Executor

Your final project Rover is going to need a slightly more sophisticated controller than the one you built in your **Think lab**. Your Think tugboat was largely reactive and driven by external sensor readings as to heading and what behavior to do. Your Rover will still need that medium level of control, but you will also need to incorporate a more procedural higher level of control to cycle your robot through a complete real-world mission. Going back to the basics, your team will want to create an extended finite state machine to carefully cycle through all of the components of a complete mission; consider oceanographic sampling mission to swim up a river and collect water samples as shown below:

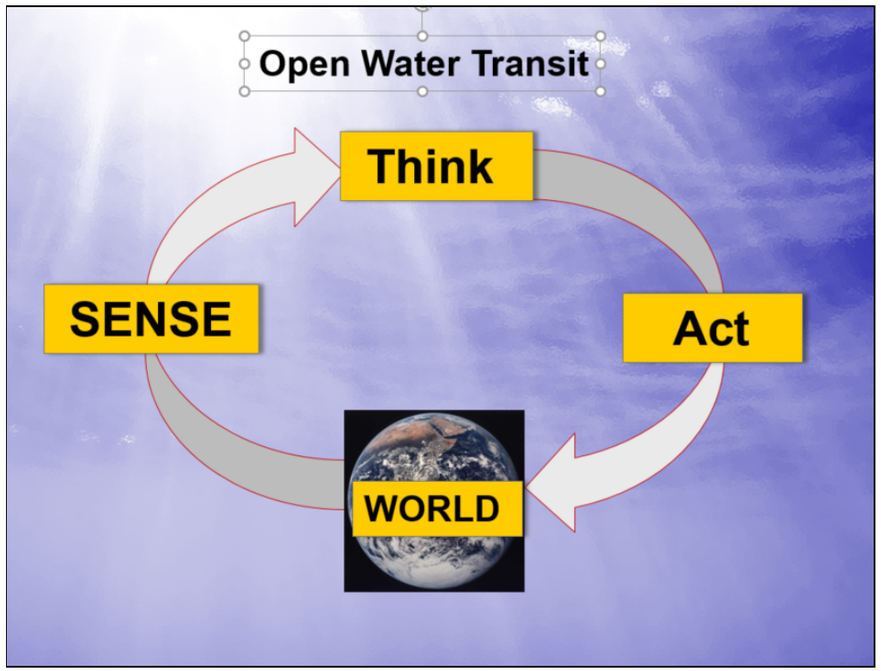


In order to command this mission a **Mission Definition File (MDF)** must be created that describes what the robot is doing on each leg of the mission and what behaviors it would express during each leg (Hint: Matlab tables make great Mission Definition Files). For example, if sampling water quality up a river, the robot would switch between Open Water Transit and Channel Transit. And there is no point in turning on the sensors until you get to the test site. A Mission Definition File can be turned into a Finite State Machine Graph (which can in turn be converted into Matlab code with a **switch** statement) that looks like this:



The gray states in the above diagram are **code that runs once,** the colored states are meta-behaviors that run over and over, or **code that loops**, until a waypoint or time limit is achieved. The first gray code sets the robot up, the yellow code runs the mission, the last gray code shuts it all down cleanly.

Each yellow meta-behavior is really just a **Sense-Think-Act** loop that runs until it gets your robot close enough to its desired waypoint that it can cycle to the next waypoint and its corresponding meta-behavior (i.e. sub-mission) in the **Mission Definition File**:

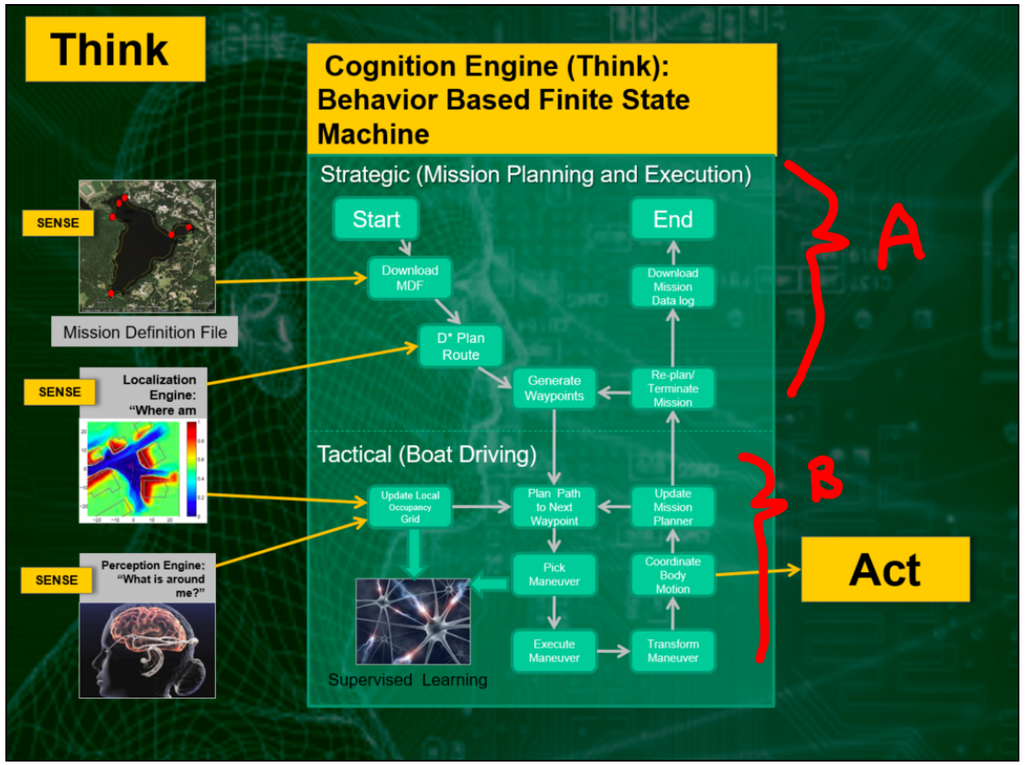


It's important to note that each meta-behavior, like **Open Water Transit** or **Open Oval Transit** is actually potentially made up of several discrete smaller robot behaviors like **Go to Waypoint, Obstacle Avoidance,** and **Look for BigRedDot.**

Each of these discrete behaviors will typically output a **brain wave** of desired robot headings and speeds that will feed into some form of **weighted polling arbiter** to fuse them into a single robot heading and speed that can be passed by ACT functions to drive the robot.

In some very real sense, your team now needs to write a two level controller, the top level is a big **Matlab switch based Finite State Machine** cycling through the desired legs (table waypoint entry) of a **Mission Definition File**. The lower level is a **behavior engine based controller** that will run the two to four discrete behaviors

your robot will need to successfully execute each leg of the mision. Graphically this type of multi-level robot controller looks like this:



We strongly recommend that your full team sits down with a whiteboard and plans out your full Rover mission in the form of a **mutually agreed on Finite State Machine Diagram**, two folks co-write big switch statement that drives mission execution in this diagram and then you break into sub coding teams to create the behaviors, functions and code that goes in each finite state block. Please note, pre-flight mission checking and on mission error handling are just as critical as the big main behaviors like **going to a waypoint** to be able to create a robust Rover that can deal with a dynamic real world environment. A robust controller would be able to deal with sensor malfunctions, sun-blinding, cloudy skies and **annoying civilians walking on track**!

Before proceeding please note; it is perfectly ok to stop your Rover dead in its tracks, take a detailed sensor scan by pivoting camera-sonar head 360 degrees and then running some optimization path planning algorithm to plot out what to do next. Just like the Mars rovers, there is no a priori need to keep your Rover moving at high speed continuously. It can stop, look and think. It probably should. Finally a bit of advice:

Carpenters have a solid guideline; **Measure twice , cut once**. The entire robotics industry could benefit from the coding guideline, **Diagram twice, code once** !

As a suggestion, not command direction, your team will need to create a set of Think code blocks (or functions) that fall into the following areas:

Think: Behavior Engine and Waypoint Arbiter

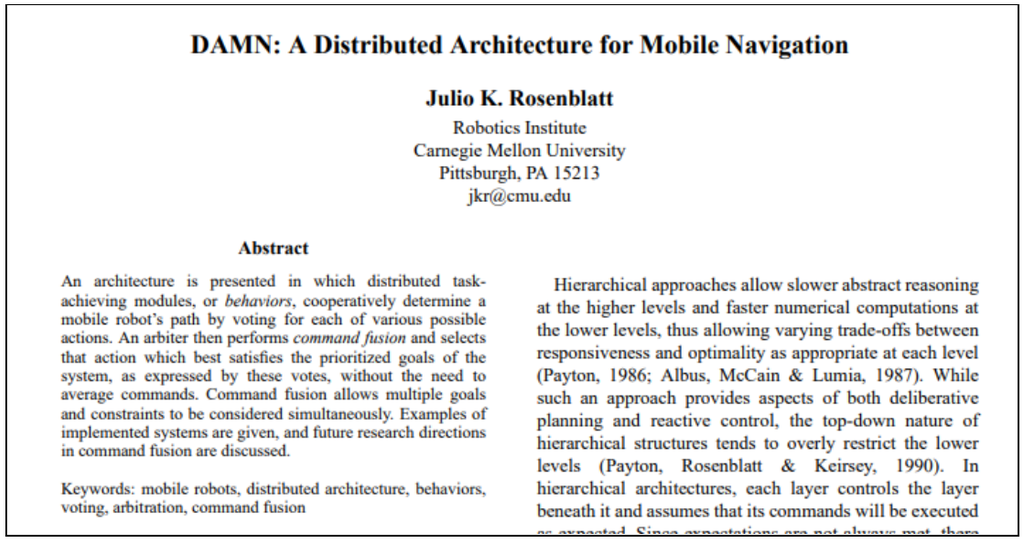
At the core of your Rover controller, you will need a good behavior engine and a behavior arbiter. You have already created a simple version of this via your **ThinkLab Tugboat** experience, a behavior engine usually contains 2 or more competing robot behaviors that all seek to command your robot's actuators.

A complex robot will have more than one arbiter, for example your Rover could have one arbiter merging competing behaviors to command the steering angle and the drive motor, while another arbiter collects and arbitrates behavior commands for where to point the pan-tilt sensor head.

Before diving into your behavior engine arbiter design, it would be good to absorb some seminal background material on one of the first ones.

A classic is the Rosenblatt DAMN arbiter presented in:

ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B



DAMN: A Distributed Architecture for Mobile Navigation

Julio K. Rosenblatt
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
jkr@cmu.edu

Abstract

An architecture is presented in which distributed task-achieving modules, or *behaviors*, cooperatively determine a mobile robot's path by voting for each of various possible actions. An arbiter then performs *command fusion* and selects that action which best satisfies the prioritized goals of the system, as expressed by these votes, without the need to average commands. Command fusion allows multiple goals and constraints to be considered simultaneously. Examples of implemented systems are given, and future research directions in command fusion are discussed.

Keywords: mobile robots, distributed architecture, behaviors, voting, arbitration, command fusion

Hierarchical approaches allow slower abstract reasoning at the higher levels and faster numerical computations at the lower levels, thus allowing varying trade-offs between responsiveness and optimality as appropriate at each level (Payton, 1986; Albus, McCain & Lumia, 1987). While such an approach provides aspects of both deliberative planning and reactive control, the top-down nature of hierarchical structures tends to overly restrict the lower levels (Payton, Rosenblatt & Keirsey, 1990). In hierarchical architectures, each layer controls the layer beneath it and assumes that its commands will be executed

Please download from Canvas and read through. It is a ground floor view of what goes into a robust arbiter:

Abstract: *An architecture is presented in which distributed task achieving modules, or behaviors, cooperatively determine a mobile robot's path by voting for each of various possible actions. An arbiter then performs command fusion and selects that action which best satisfies the prioritized goals of the system, as expressed by these votes, without the need to average commands. Command fusion allows multiple goals and constraints to be considered simultaneously. Examples of implemented systems are given, and future research directions in command fusion are discussed.*

The Distributed Architecture for Mobile Navigation: Deliberative planning and reactive control are equally important for mobile robot navigation; when used appropriately, each complements the other and compensates for the other's deficiencies. Reactive components provide the basic capabilities which enable the robot to achieve low-level tasks without injury to itself DAMN: A Distributed Architecture for Mobile Navigation or its environment, while deliberative components provide the ability to achieve higher-level goals and to avoid mistakes which could lead to inefficiencies or even mission failure. But rather than imposing an hierarchical structure to achieve this symbiosis, the Distributed Architecture for Mobile Navigation (DAMN) takes an approach where multiple modules concurrently share control of the robot. In order to achieve this, a scheme is used where each module votes for or against various alternatives in the command space based on geometric reasoning, without regard for the level of planning involved.

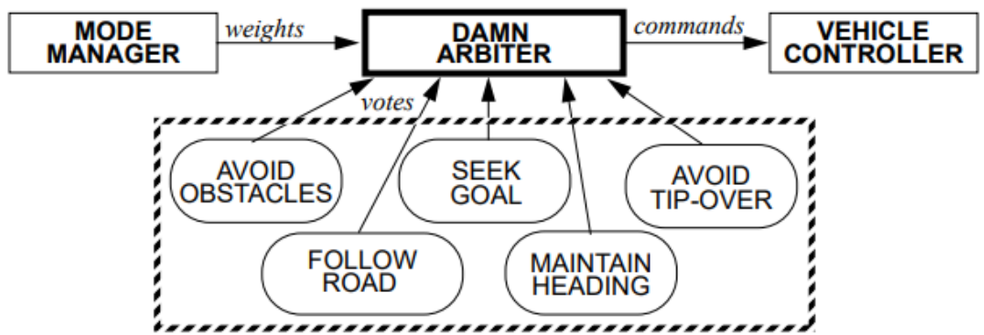


Figure 1: Overall structure of DAMN.

The arbitration process used in DAMN is illustrated in Figure 2, where: (a & b) the votes from behaviors are received, (c) a weighted sum of those votes is computed, nd (d), the summed votes are smoothed and interpolated to produce the resulting command sent to the vehicle controller.

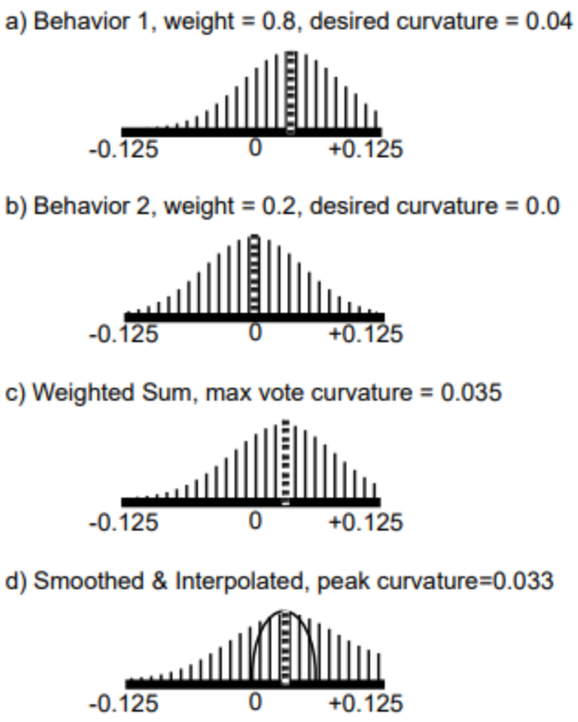
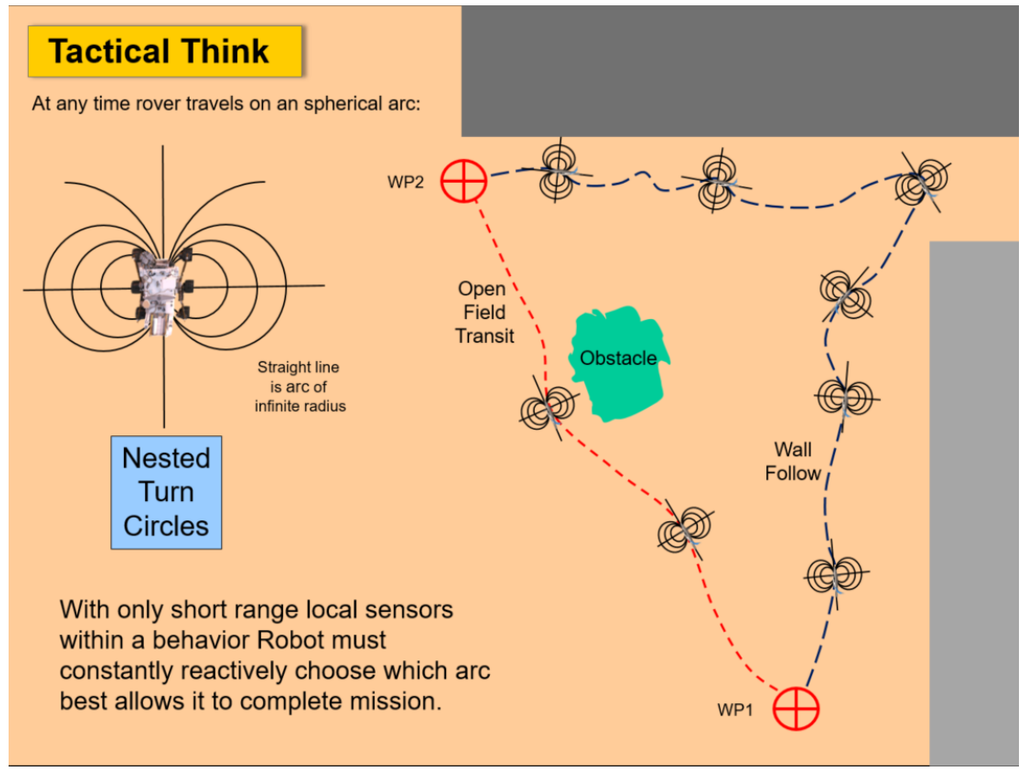


Figure 2: Command fusion process

You have already used a simplified version of this on your ThinkLab tugboat.

Your behavior engine's task is simple. At any instant of time a wheeled ground vehicle in motion is traveling on one of a nested set of **bumble-bee circles** (see diagram). The purpose of your behavior engine and arbiter is to pick the best instantaneous circle to be on to achieve its current main behaviors mission. The DAMN arbiter is a tried and true way to pick this circle, but it is not the only way. If you search behavior arbiters you will find many other equally interesting options.



Once your team has chosen an arbitration path, it is time to move on to creating discrete Rover behaviors. Your team's arbiter could look like this:

```
function [direction, speed] = roverArbiter (behavior1, behavior2, behavior3, behavior4)
% roverArbiter takes the weighted steering vectors from multiple behaviors and
% merges them down into a single Rover steering angle and speed.
% written by Vision and Wanda 3-30-21 Rev A
    Body of code
end
```

Think: Go to Waypoint Behavior

One powerful concept that allows a human operator to define and control a complex robot mission is the idea of a geographically located waypoint. A waypoint is a physical location in the mission area that has some form of significance. It might be a dock, the edge of a building, a precise GPS latitude and longitude or the location of a scientific payload.

There is a whole sub-genre of people that run through the woods from GPS waypoint to waypoint (often called hikers) and the skill to do so at speed is called orienteering. One really useful behavior (that you already used in your second week Robot Toolbox training) that you could create is to use Matlab's built in **Pure-Pursuit** functions to create a path stored as physical waypoints in a **Mission Definition File**.

Depending on how your team structures your code, a GotToWaypoint behavior function could look something like this:

```
function [bumbleBeeSteeringVector] =GoToWayPoint(roverOcupancyGrid,
targetWaypoint)
% GoToWayPoint takes the current occupancy grid and next target waypoint and %
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover. Rover centric occupancy grid. 0=open space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

Think: Obstacle (and Alien) Avoidance Behavior

A well designed Rover control system doesn't drive into either stationary or moving obstacles. Using the combined data from Sharp IRs, Sonar and your camera system; fused into a **Rover-centric Occupancy Grid** your rovers obstacle avoid behavior should produce a weighted steering vector that minimizes the potential for collison. It could look something like this:

```
function [bumbleBeeSteeringVector] =ObstacleAvoid(roverOcupancyGrid)
% Obstacle Avoid takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer avoid Rover hitting obstacles. Rover centric occupancy grid. 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

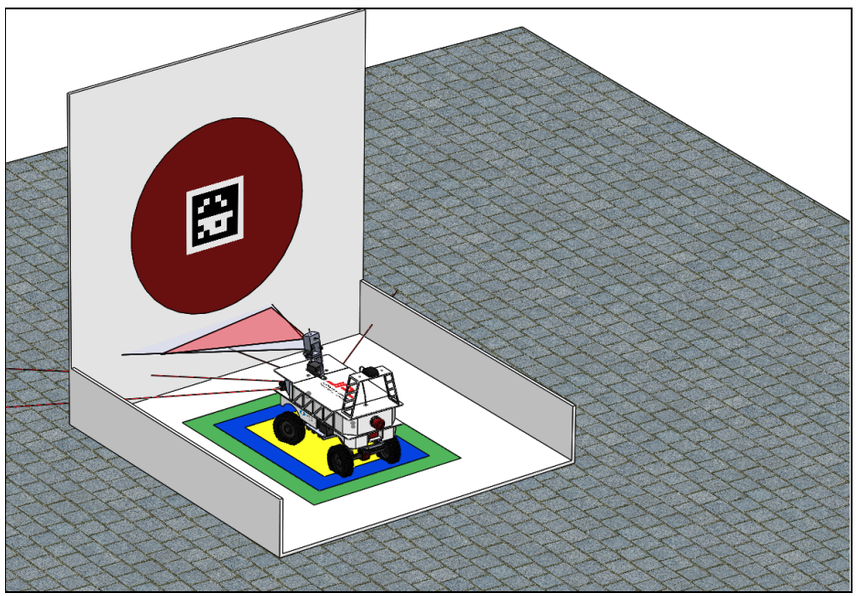
Think: Wall or Curb Follow Behavior

GPS has its advantages and disadvantages. First off, it can only tell you where your Rover is within a meter or two. Second, it tends to drop out near buildings at the most awkward of times. You can compensate for this by using fixed observable features in your environment, like walls to follow, that in conjunction with **a good digital map** (ground based occupancy grid) will let your Rover smoothly get from waypoint A to waypoint B. Luckily the Rover test track in the Oval has lots and lots of curbs, walls and features your Rover can follow to get around in the absence of reliable GPS. It's highly recommended that you consider creating a right and left-hand wall following behavior. A prototype for this would look like:

```
function [bumbleBeeSteeringVector] =WallFollow(roverOcupancyGrid, right-left)
% WallFollow takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer to follow the wall at a fixed offset. Rover centric occupancy grid. 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

Think: Dock Rover Behavior

At the start and potentially end of each Rover mission, your robot has to leave a well-marked dock and find and park at another well marked dock.



Although your team will be provided with the GPS Latitude and Longitude of the docks physical location, that position fix will only be good to 1-2 m and you will need to use your vision system, pan-tilt sensor head, Red-dot finder and potentially April Tag finder to precisely navigate onto the docks parking station. In addition to the Dock Rover Behavior, you might also want to create a Find Dock behavior that stops the rover and slowly scans a 360 degree horizon to look for the dock.

Once you have a good idea of where it is and are close enough to it to have a good visual lock on the big red dot and April Tag, then you can visually steer your rover onto the landing pad. You might need a behavior function that looks like this:

```
function [bumbleBeeSteeringVector] =DockRover(roverOcupancyGrid,
RedDotCoords)
% DockRover takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover toward dock. Rover centric occupancy grid. 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Tom and Wanda 3-30-21 Rev A
    Body of code
end
```

One very interesting wrinkle is how to best stop a rover on a landing pad a fixed distance from Big Red Dot.

Think: Drive Over Bridge Behavior

As part of the overall mission goals, your rover may need to cross over a bridge. You can find the bridge with a combination of April tags and GPS location, but you will need your lidar and Sharp IR array to successfully get across it. DON'T ROLL YOUR ROVER AND BREAK YOUR LIDAR!:

```
function [bumbleBeeSteeringVector] =CrossBridge (roverOcupancyGrid, Bridge
Coords)
% Cross Bridge takes the current occupancy grid , the obstacles near it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover over bridge on. Rover centric occupancy grid.
% 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Sam and Jackie 3-30-21 Rev A
    Body of code
end
```

Think: Drive Under Arch Behavior

As part of the overall mission goals, your rover may need to drive under an arch. You can find the arch with a combination of April tags and GPS location, but you will need your lidar and Sharp IR array to successfully get under it.

```
function [bumbleBeeSteeringVector] =UnderArch (roverOcupancyGrid, Bridge Coords)
% Cross Bridge takes the current occupancy grid , the obstacles near it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover over bridge on. Rover centric occupancy grid.
% 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Sam and Jackie 3-30-21 Rev A
    Body of code
end
```

Think: Go Home Behavior

If and when something goes wrong, your Rover gets lost, it starts to rain or you complete the mission successfully, you just don't know what else to do; it will always be useful to have a **GoHome** behavior (just like your cars GPS), that will safely and autonomously bring your Rover back to its starting point and home dock.

```
function [bumbleBeeSteeringVector] =GoHome(roverOcupancyGrid, HomeCoords)
% GoHome takes the current occupancy grid , the obstacles on it and
% generates a weighted steering angle brainwave that can be used by an arbiter
% to steer Rover towards its home dock. Rover centric occupancy grid.
% 0=open
% space 1-10= obstacles
% 20 = dock, 30 = April Tag
% written by Sam and Jackie 3-30-21 Rev A
    Body of code
end
```

Act: Drive Rover on Curved Path R at Speed S

Compared with the potential complexity of both the Sense and Think functions needed by your Rover, the Act functions should be relatively straight forward. Your team may want to create three; one for the Rover, one for its Pan-Tilt Head and one for the payload deployment system. You have already successfully controlled RC type servo motors in the **ACTLAB,** here the one new wrinkle for the final project is that we will use real motors on a real rover.

```
function [ ] =DriveRover(steerAngle, speed)
% DriveRover takes in the desired steering angle and speed for the Rover and
% commands the steering Servo and embedded drive motor speed control to those
% setpoints.
% written by Brady and Amy 3-30-21 Rev A
    Body of code
end
```

Act: Point Pan at Angles {P}

If you are focused on conserving energy on your Rover mission (and you should be!) It makes a lot more sense to scan the horizon with sensors mounted on a movable Pan Head than to rotate the entire vehicle to collect data. For this and a host of other good reasons, you will probably need a simple Act Pan command that will let Think behaviors point the head sensors where they should be pointed. You probably need something like this:

```
function [ ] =DrivePan(panAngle)
% DrivePanTilt takes in the desired pan angle for the sensor head and
```

```
% commands the pan and tilt Servos to those setpoints.
% written by Brady and Amy 3-30-21 Rev A
    Body of code
end .
```

Remote Desktop Operator Control Unit

Last but not least, you will need to communicate with and control your rover over the Olin Robotics Network from a stationary student laptop, nominally called an Operator Control Unit (OCU). Typically this will be done via Windows 10 remote desktop and perhaps with a clean Matlab OCU APP.

Suggested Team Work Breakdown

There is a reasonable, but significant amount of work to do between the final project kickoff and the final (wildly successful) demo. In order to complete it in time, we recommend that you organize your student team to use individual strengths to the maximum possible level, while still giving all teammates a balanced and fair access to contributing to the overall design. Your team can structure themselves any way you are comfortable, but please consider the guideline below while doing so. It may make sense to break up into four sub-teams at the start and end with one team where everyone is coding and field testing at the end for the final demo. In discrete terms your team will need to cover the following tasks

Project Management

Your team needs a **Keeper of the Main Finite State control diagram**, who will both constantly update it and will keep it aligned with the project mission. Your team will also need a **Work in Progress task manager** whose goal is to make sure all tasks are getting done and will be completed in time. Please see the final section of this document for a brief overview of creating a work in progress tracking system.

ME

This course has been pretty heavily skewed to code generations and robot component wiring up to this point. To give our hard-core MEs something substantial to do, your team will need to quickly design, manufacture and assemble your Rover's Sintra Hull as well as design and print its sensor mounts and sensor head. Probably only need 2-3 students on this, preferably with previous strong ME design, fabrication and assembly skills.

ECE

Using the provided wiring diagram as a guide and promising to never hook up any hardware hot, the EE work to be done includes making a quick plug-it-all-together and get it to work on a table top phase, followed quickly by a careful and robustly install all components into the Rover's hull phase. Bearing in mind that we've overspent the course budget at this point and there are no spares, the teammates selected to do this work should be patient, methodical, careful and have reasonably good EE assembly skills and **NEVER EVER PLUG IN PARTS HOT !** Please note: we don't have spares, if the ME's break an ME part, they can just rebuild it. If your EE team burns out, shorts, frys or loses a critical piece, I can't replace it and you might have to pass the hat around the team to fund ($$) a new replacement part.

CODE

The initial code work would be in the form of extending the course robot code template to include all of the new mission functionality required and then once that is in place, generating, field testing and debugging the many functions required to do the full mission. It may be advisable to have a few teammates work on the main structure of the code first,while the ME's are design-building the hull and the EE's are hooking up the table top wiring, then have the entire team shift into writing functions.

This all being said, please strive for inclusion and don't shift teammates from tasks they passionately want to do , just because they are relatively inexperienced at them. **This is Olin, we learn by doing.**

Suggested Project Timeline

Your team will have just about 5 weeks plus a few days to design, build, test, debug, test, improve, test ,debug and demo your Rover. This is a tight, but realistic schedule. You Rover is, by design, a fusion of all of the basic fundamental robotic components you have already gained technical skills with in the Sense-Think-Act labs. To stay on track and on schedule, you will want to consider meeting these project milestones:

Project kickoff: Friday April 8

Design Done and Design Review: Tuesday April 12

Rover built and ready for code: Friday April 15 (a week from kickoff)

Achieve stable waypoint navigation: Friday April 22 (2 weeks from kickoff)

Perform full mission (unoptimized): Friday April 29 (3 weeks from kickoff)

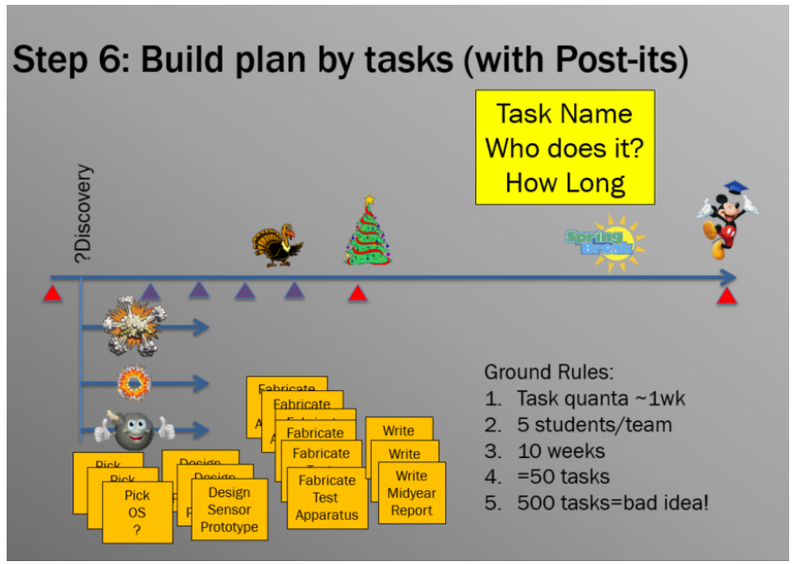
Optimize mission for final demo and Final demo: Wednesday May 4 !

Required Work Management System

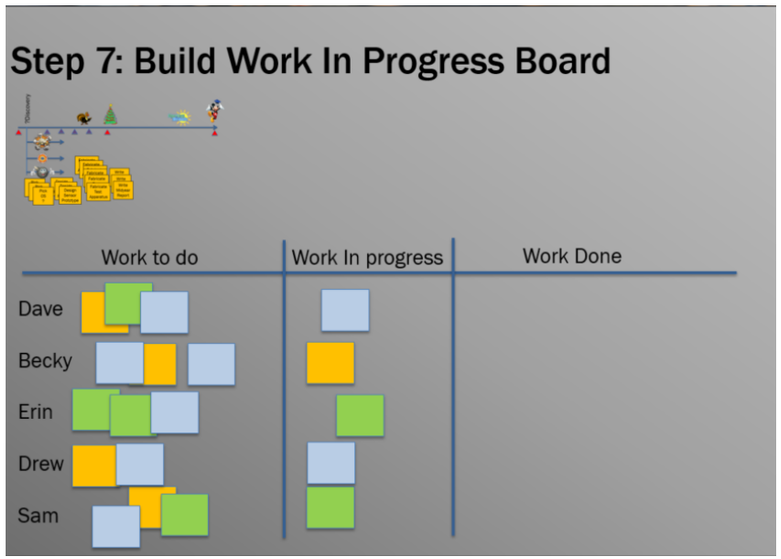
By design and by intention this final project is not as heavily scaffolded with respect to workflow as the three preceding Robot labs were. This is to give your team substantial experience in self-organizing your own workflow to optimize project results with a small team within the given time frame.

You are free to self-organize in any way you like, but that being said, to insure fairness to all, to achieve individual student contribution transparency and to give you some exposure to a low-overhead small-team self-management system; we will use a commonly used, lightweight post-it based **Work In Progress** (WIP) workflow guidance system.

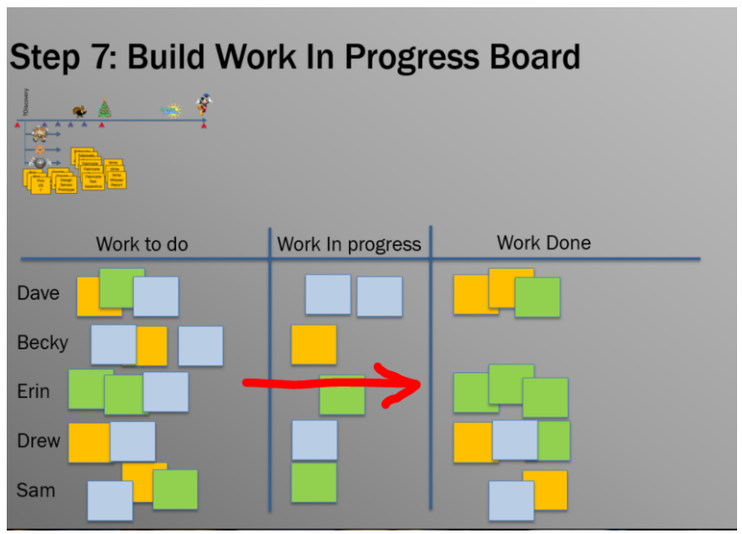
To set up and use this system, first create your team's finite State Machine Diagram to facilitate the planning conversation. Then sit down as a team (we will do this in class) with a set of post-it notes and write out each half week long task needed to get to the final demo. Each student gets roughly 8 post-it notes and half week tasks.



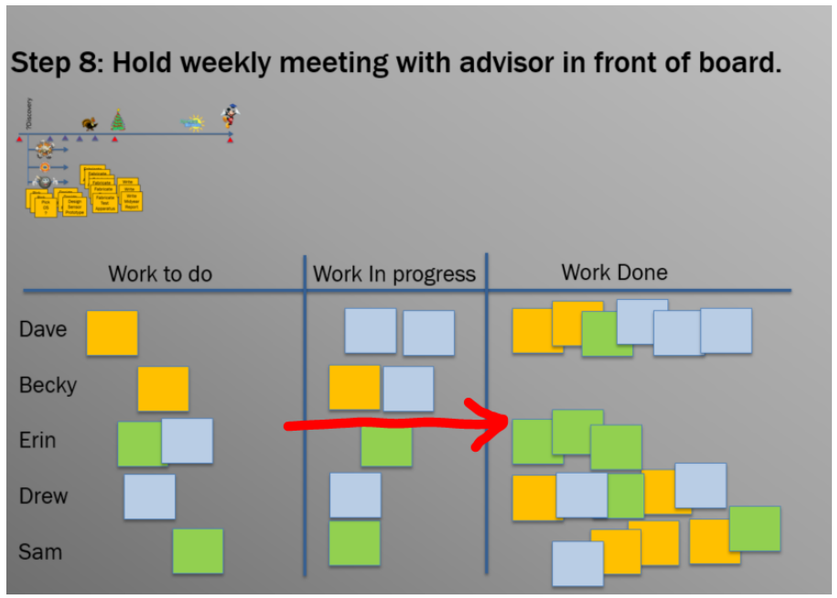
In your team space create a three column WIP board and place the tasks to be done by each student next to their names:



As work progresses, move the post-it notes from the left to the right:



On a bi-weekly basis update the WIP board:



A good common WIP board lets all team members see progress, see what work remains to be done and allows the team to quickly update their supervisor (and course instructors and Ninjas) on progress and setbacks. Most importantly it lets your team work in a highly efficient, asynchronous manner, see where you stand and see what needs to get done next. WIP boards like this are very common in start-up companies, aerospace companies and consumer product companies, really any place with a fast project cycle time. Get good at this and your team will succeed.

Conclusion

You have a great team, a full supply of hardware and a well-defined mission. Your team has all of the pieces needed to succeed in this final Planetary Rover contest. Don't forget to seek out help from the course instructor and Ninjas as needed and now dive right in and go for it !

May the best Rover win.

ENGR3390: Fundamentals of Robotics Final Project 2022 Rev B