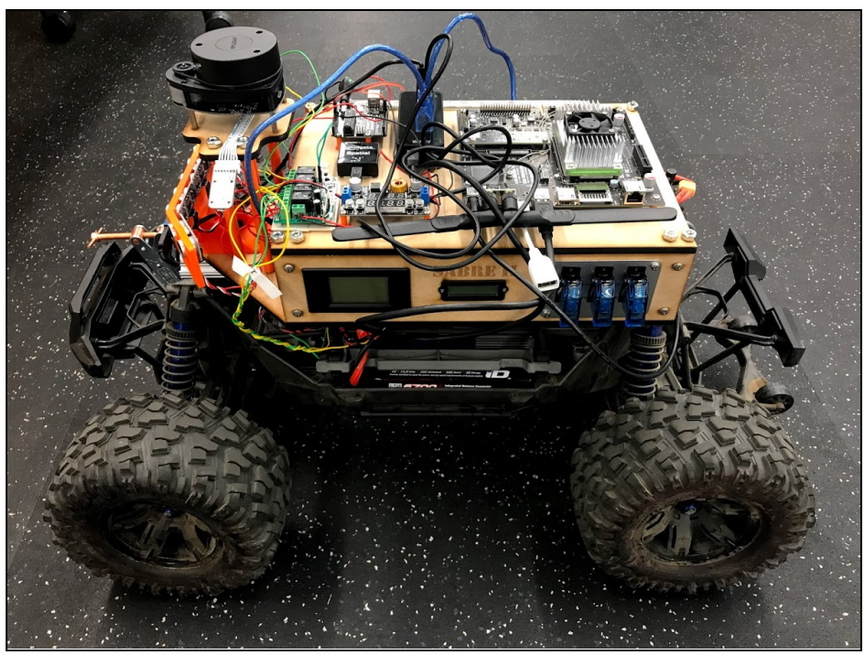
ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

In the next six weeks, you will move on from the MATLAB skill building tutorials you have just completed to a 6 week set of hands-on-hardware, team-based tutorials (called Robot-labs). In these tutorials, the primary goal will be to both master the underlying technology beneath the SENSE-THINK-ACT robotics components they contain and also generate your own toolbox of MATLAB robotics functions to efficiently work with those same components. You will use your new robot toolbox on a big, multi-week final project where your team will build a fully operational autonomous robot to do a representative real-world mission. This year we will have teams build and lightly compete in an autonomous rover race around the Olin Oval.



Pretty much all robot control software shares a SENSE-THINK-ACT data flow in some manner or another. In fancier academic language, “perception” feeds into “cognition” which commands “actuation.” You will find deep resources in background material; technical papers, books, videos, journal articles, in all three of these areas as you move into the robotics technical space. A robotics-engineer can build a whole career investigating and building new technology in just one of these areas.

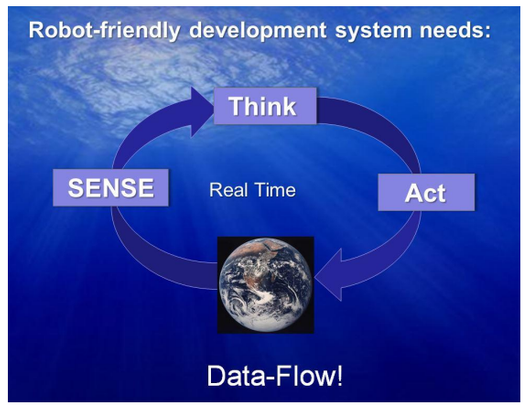


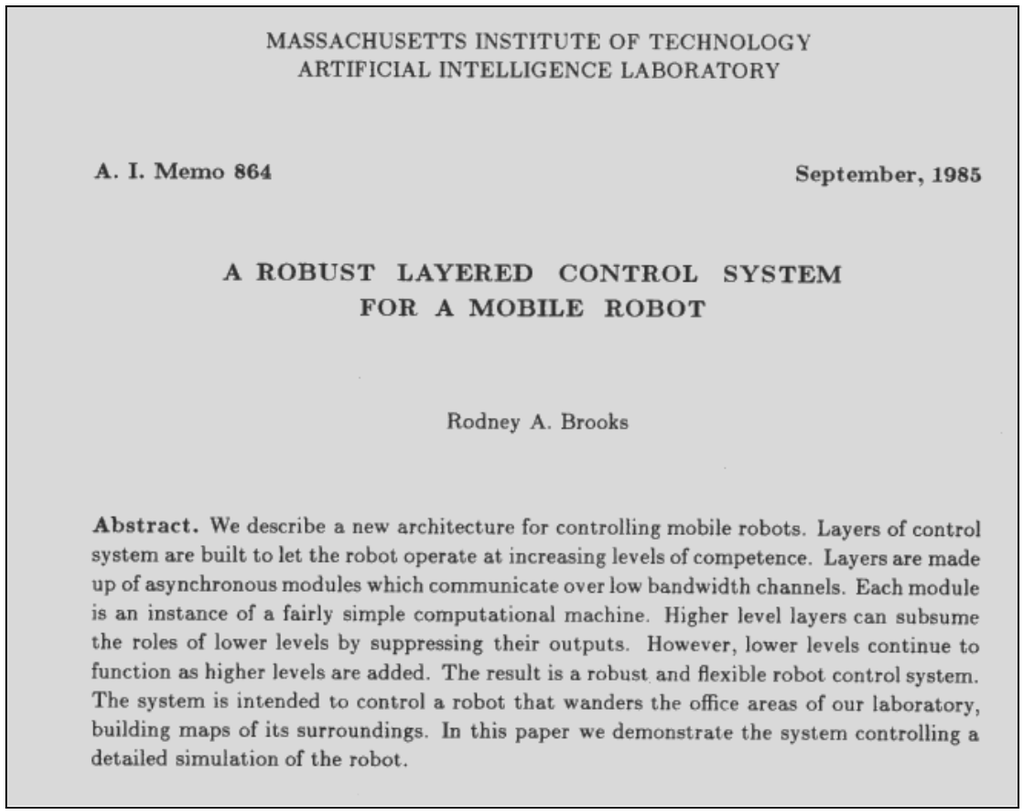
Figure : Sense-Think-Act Control Loop

THINK: involves designing robot behaviors {follow wall, avoid obstacle, move toward target, pick up container, follow lane, stop for pedestrian, weld on joint line, etc.} that receive data from the robot sensor suite processing code {SENSE: “Where am I?”, “What is around me?” and “How am I?”}, perform useful mission based cognition on this input data stream {based on sensor data, have robot do something useful}, and then runs the outputs from the THINK behaviors into a set of arbiters which finally command the robots actuators {ACT} to send setpoints to the actuator code to execute.

That's a pretty big wordy sentence. It's much simpler in practice. For this lab and with the Arduino tools you have already used, you will be given a set of SENSE Matlab functions, you will create both a handful of THINK behavior functions, and one or two THINK arbiter functions (a arbiter for each main actuation system; wheels, arms, pan/tilt, etc. is pretty standard) and will use a small set of pre-written Matlab ACT actuator functions. You can then just add those new functions to the MATLAB SENSE-THINK-ACT robot control template that you developed in the second tutorial for this course, reusing code to the maximum extent possible.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Let us look at the theory behind all of this a bit before diving right into the mechanics of how to code it. It's important to understand why you are doing something, before you master how you do it. Many of today's robotics engineers feel that a large part of modern robotics started with Prof. Rod Brook's seminal 1985 paper on behavior based robotics:



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo 864 September, 1985

A ROBUST LAYERED CONTROL SYSTEM
FOR A MOBILE ROBOT

Rodney A. Brooks

Abstract. We describe a new architecture for controlling mobile robots. Layers of control system are built to let the robot operate at increasing levels of competence. Layers are made up of asynchronous modules which communicate over low bandwidth channels. Each module is an instance of a fairly simple computational machine. Higher level layers can subsume the roles of lower levels by suppressing their outputs. However, lower levels continue to function as higher levels are added. The result is a robust and flexible robot control system. The system is intended to control a robot that wanders the office areas of our laboratory, building maps of its surroundings. In this paper we demonstrate the system controlling a detailed simulation of the robot.

I was lucky enough to work for Rod at that time at the MIT Artificial Intelligence Lab where this work started. Please see the Canvas THINK folder for full text of paper and give it a read through. This paper (and the steady stream of papers out of his MIT Mobot lab) changed the course of modern robotics. Before Brooks, robots were slow, dumb and required enormous rooms full of computers to laboriously build fragile simulation 'toy block' world models of the real world in order to make even the smallest decision. After Brooks, robot cars autonomously drive at high speed through the streets of Boston on a daily basis.

The original Brooks robots were of about the same level of complexity as those we seek to build in this Fun-Robot class. Following in the footsteps of a successful strategy, we will be building a simplified version of his **Subsumption** architecture, using multiple robot **Behaviors** simultaneously feeding commands into a set of simple **Arbiters** to let your Olin robots perform complex multi-modal missions, in real-time, in a complex real world. In a brief summary, before Brooks, robot control software was written as a linear sequential procedure, one, long, block of dense robot control code:

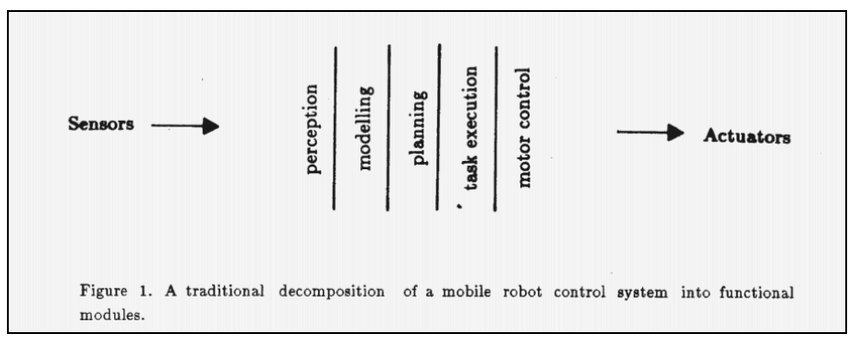


Figure 1. A traditional decomposition of a mobile robot control system into functional modules.

Brook's key insight was that biological systems don't run that way. Instead their control is a complex tapestry of many behaviors, running in parallel, where higher order behaviors can "subsume" control over actuators if needed, but the lower level behaviors will always robustly control the actuators unless/until a higher behavior steps in. Your breath control is a great example. Take a deep breath and hold it until you finish reading the next two paragraphs. While you were reading the initial part of this lab, your lower level lung control behavior was steadily commanding your chest muscles to expand to pull fresh air in, and then to contract to push your exhaust out.

This behavior chugs along, nicely keeping you alive, without your brain thinking too much about it. Having this awesome lower level behavior frees your brain up to think about more demanding things, like reading this paragraph.

Then bam! You fall out of a canoe and a higher level water-survival behavior (diving reflex) rapidly closes off your air intake to keep you from pulling dirty river water into your lungs. It subsumed control over your lung actuators. Subsumption based bio-control and just saved your life! The ability to have both behaviors (and in practice many behaviors) running at the same time, gives people and robots the ability to robustly survive and adapt to a dynamically changing world. You can stop holding your breath now, relax your subsumption override, and let the lower level behavior take over and breathe for you while you read the rest of this lab.

Brooks proposed to run many robot-behaviors, from low to high, all in parallel, all able to see the same sensors and drive the same actuators:

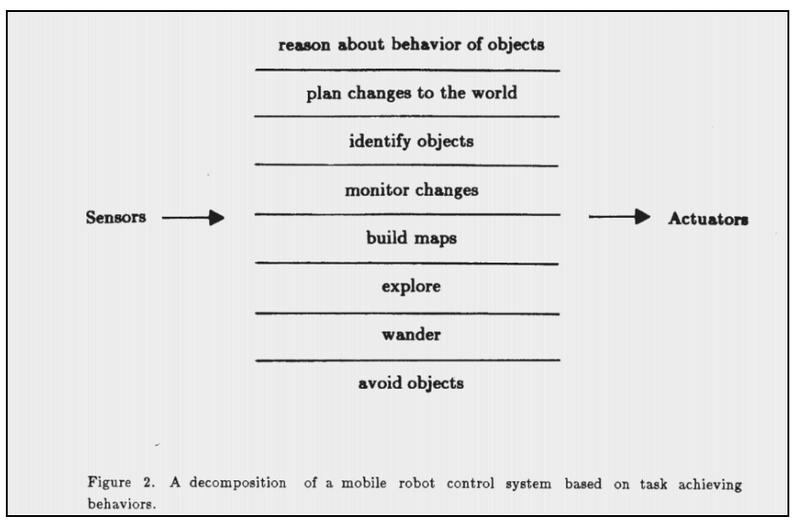


Figure 2. A decomposition of a mobile robot control system based on task achieving behaviors.

This one bold step allowed robots, for the first time, to drive around and interact with a dynamically changing world, in real-time, using non-room size computers. Specifically, not just in a computer simulation sped up, but in actual real-time.

Within this form of control system, you will write a set of simple behaviors as functions and then weave them together into a structure like the one in Brook's original paper:

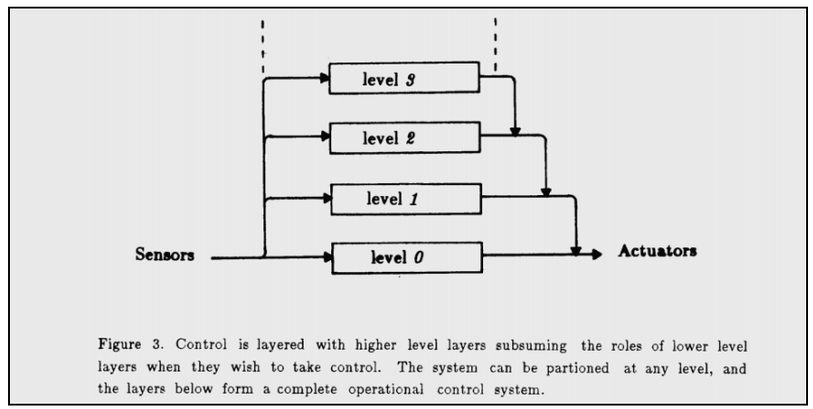


Figure 3. Control is layered with higher level layers subsuming the roles of lower level layers when they wish to take control. The system can be partioned at any level, and the layers below form a complete operational control system.

We will use finite state machines (FSM) encoded into Matlab functions to hold each behavior:

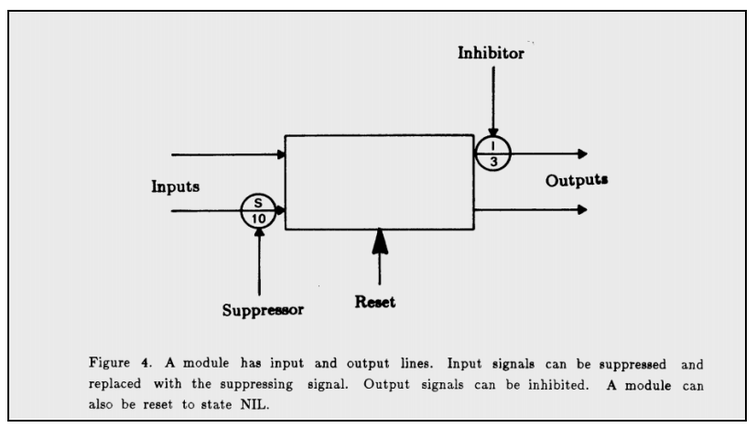


Figure 4. A module has input and output lines. Input signals can be suppressed and replaced with the suppressing signal. Output signals can be inhibited. A module can also be reset to state NIL.

Eventually ending up with a system of about the complexity shown below to drive your THINK lab-cart's robot RoverRover.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

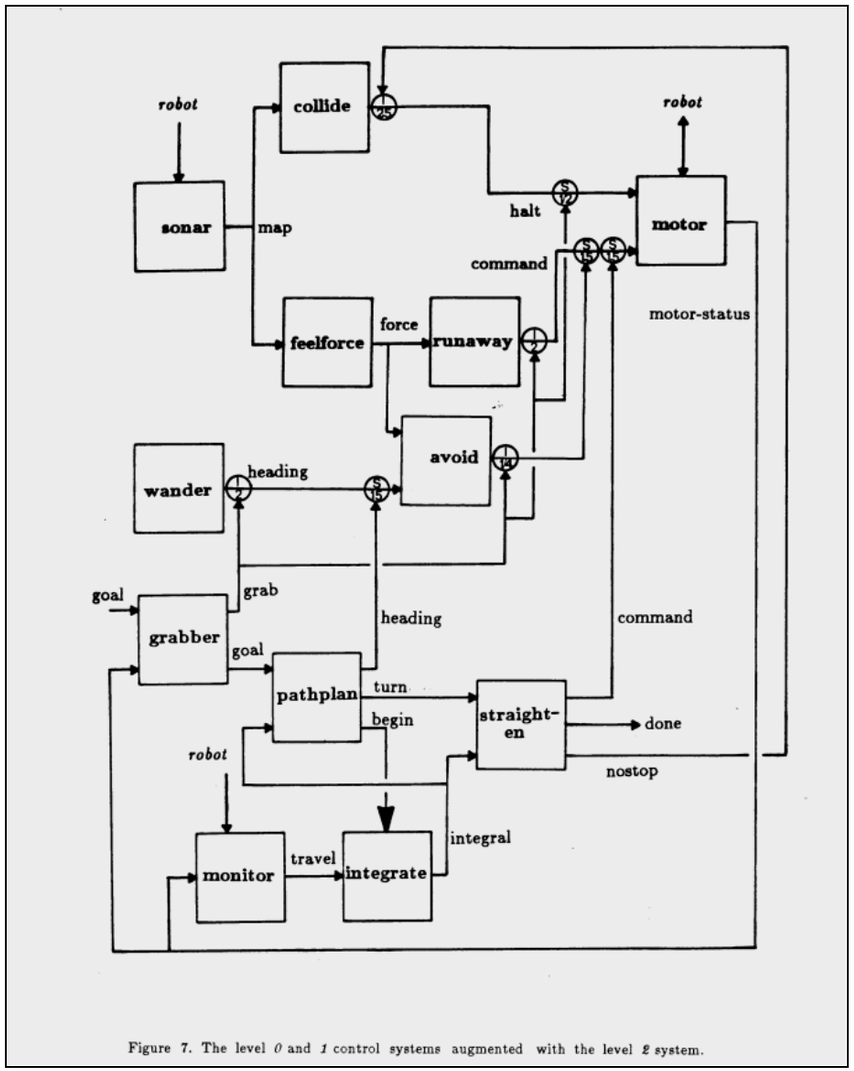


Figure 7. The level 0 and 1 control systems augmented with the level 2 system.

In this THINK lab, we will give you the SENSE and ACT functions for the rover code and work with your team to develop and give you hands-on experience with writing the THINK part, namely, the following set of simple robot-Rover behaviors:

1. Explore: Head out on a random, time switching, heading to see what is there
2. Avoid: Head away from obstacles in front of Rover
3. Follow: If you see a sought after target (a brightly colored orange traffic cone), follow it.

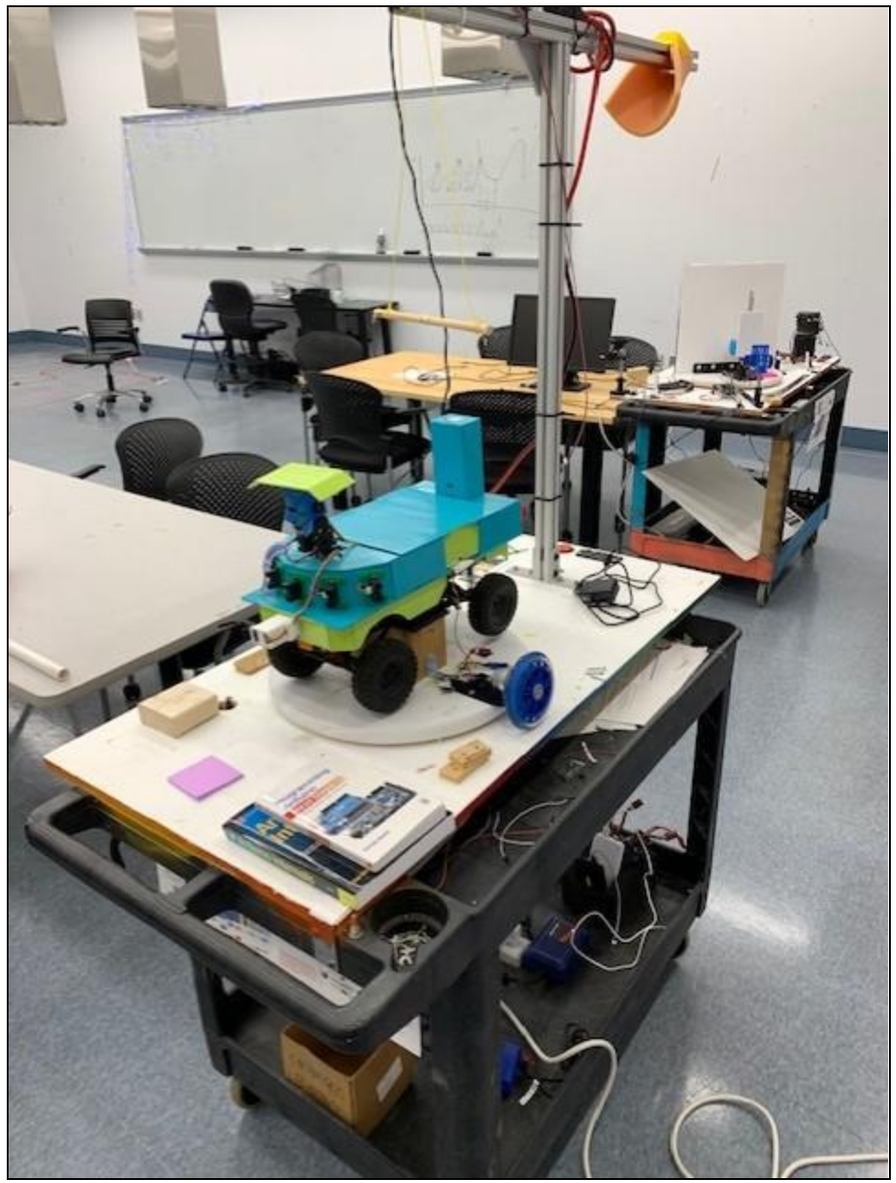
And help your team write two simple arbiters;
-Heading: one to control the commanded steering angle of the Rover
-Speed: and a separate one to control the speed of the drive wheels.

We will use a hardware-in-the-loop simulation model to drive your physical Rover (on the Think lab test cart turntable) in a virtual 30' diameter mini-Olin Oval..

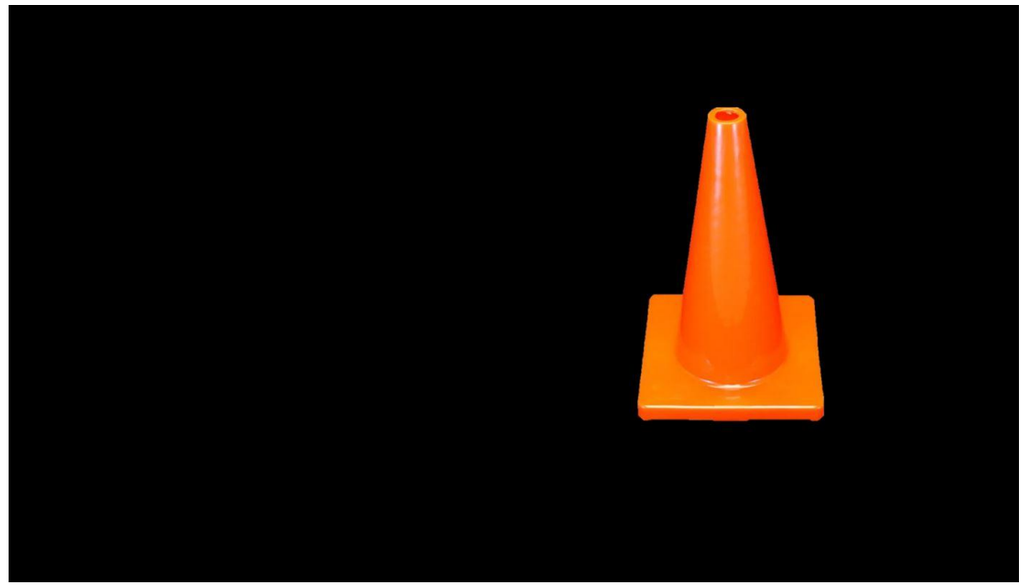
Finally, as in all 4 labs, we will ask you to, in parallel with your lab work (in part 2 of this assignment), to keep advancing your MATLAB programming skills by carrying out an additional set of simple MATLAB tutorials. Please spread them out over the next two weeks, in whatever way you are comfortable.

Think Lab

This lab contains a fully functional robotic RoverRover mounted on a driven turntable. The RoverRover has an Arduino Robot controller on board, as its brain, and a forward looking WebCam and an array of Sharp Infrared range sensors as its main perception suite, and a set of embedded servo controls for the front steering wheels and rear drive wheels. You will be given working functions for the sensors and actuators, your team's mission is to create all of the THINK code that goes in between:



Think Lab Robot Rover Hardware

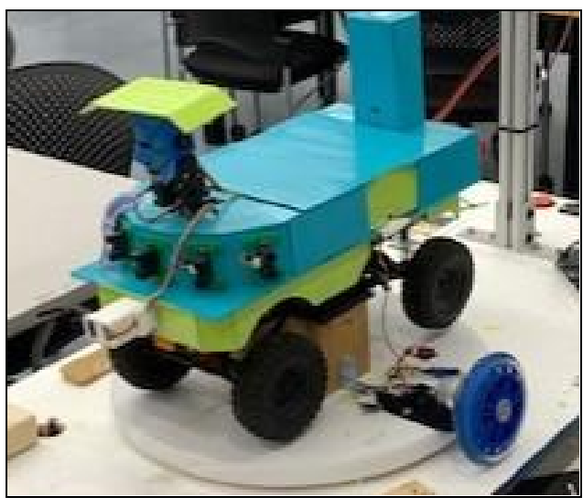


Traffic Cone Target

Description of Robot Rover SENSE code functions

The THINK lab robot RoverRover has three types of sensors on it, an IR-Range suite to see and help avoid obstacles, a web Camera to find the orange cone and a turntable mounted Potentiometer to stand in for the Rovers compass to provide a Rover heading. Each sensor system is described below:

1. There is an array of 6 SHARP Infra-red range sensors on the bow.



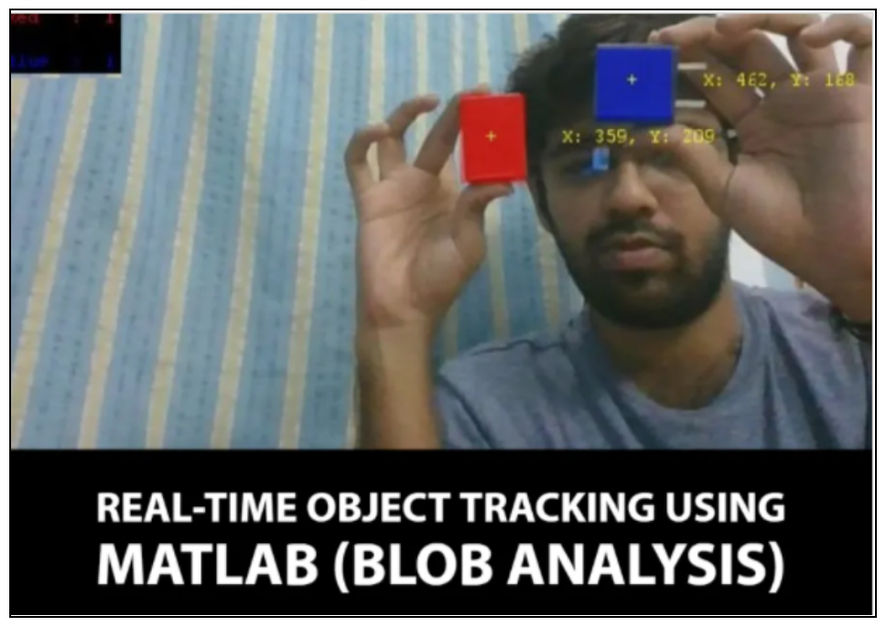
Sharp IR Range Sensor Array and WebCam

They can be used for obstacle avoidance to give you a set of ranges to whatever is in front of your Rover. We will also be using virtual Sharp IRs in the simulated tank environment to give you range data for what is in front of the Rover. You will get to work with real Sharp-IRs in depth in the SENSE lab and again on your final project Oval Rover.

2. There is a webCam to let you find and track the orange cone. The webCam is a quite high quality computer vision sensor that works seamlessly with your laptop. A quick orientation video from the MathWorks can be found here:

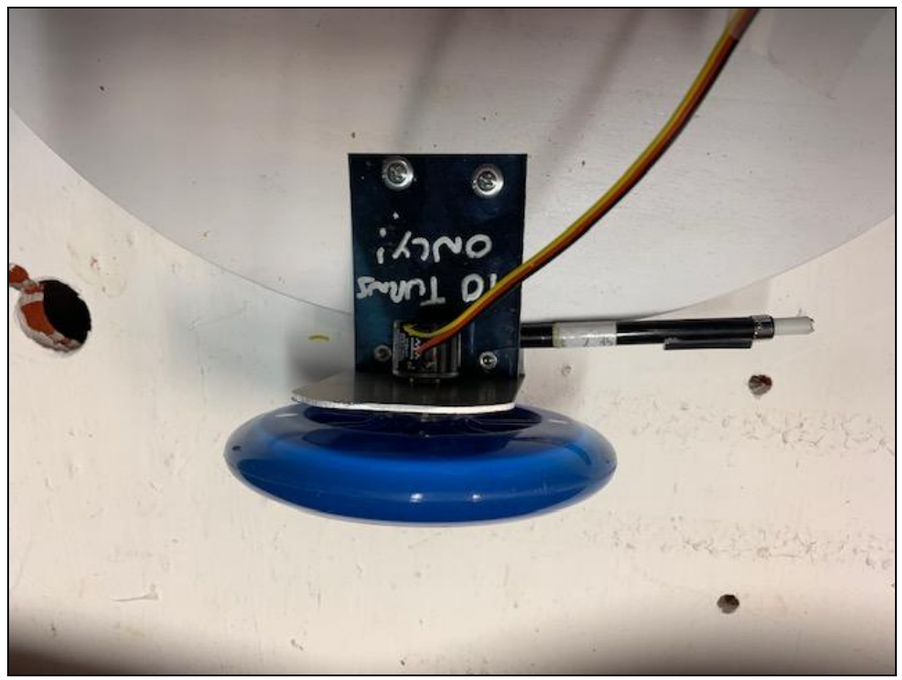
   https://www.mathworks.com/videos/introduction-to-matlab-with-image-processing-toolbox-90409.html

   A typical color tracking code output of webCam shown here:

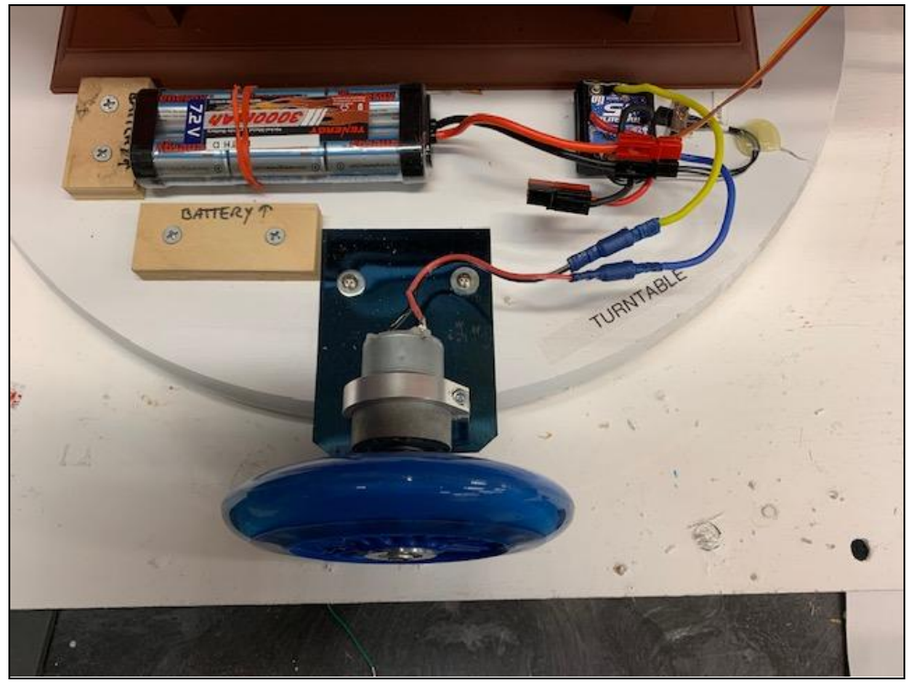


   You will work the basics of computer vision, color blob tracking and with the webCam extensively in one of the SENSE lab's test stations. So that you can focus on THINK for this lab, we will supply you a set of completed functions to do so here.

3. There is a multi-turn potentiometer to tell you the Rover's current heading with respect to the test cart. We will provide you a function to read the heading pot.The heading turntable Potentiometer shown here:



4. There is a Marine Viper servo speed control to drive the turntable drive motor. It is basically an open loop DC motor speed control. It's pre-wired to the turntable drive wheel motor. You will get to work extensively with all types of position and velocity servo motors in the ACT lab. Here, to let your team focus on THINK, we will provide a function to drive this turntable motor. The battery for this motor sits in a cradle near the hull on the turntable.



Turntable drive motor and MarineViper speed control

5. There is a standard RC servo motor inside the Rover connected to the steering wheels. We will provide a function to let you position control them.

6. The rear drive wheels are driven by an internal ESC speed control unit located inside the Rovers hull. Its propulsion battery sits inside the hull next to the speed control. We will provide you with a function to control the wheel's speed forward and back.

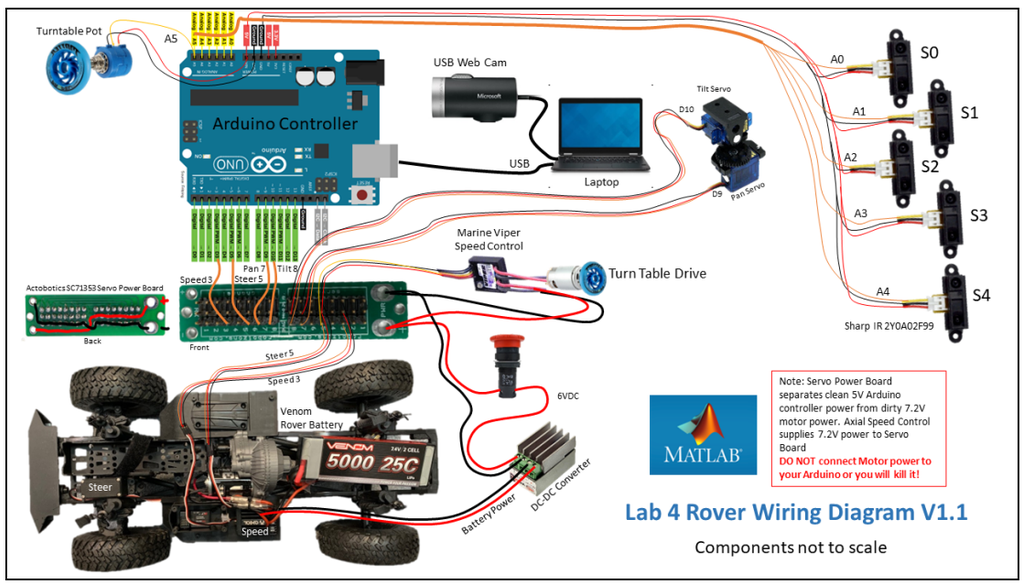
7. There is also a set of RGB super bright LEDs inside the Rover. As a stretch goal, you can hook these up to your Arduino and change the colors based on what behavior you Rover is currently doing.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Description of Robot Rover Control Wiring

The THINK lab consists of a fully operational small robot control system for an autonomous robot Rover Rover. It is built around a single Arduino and shown below:

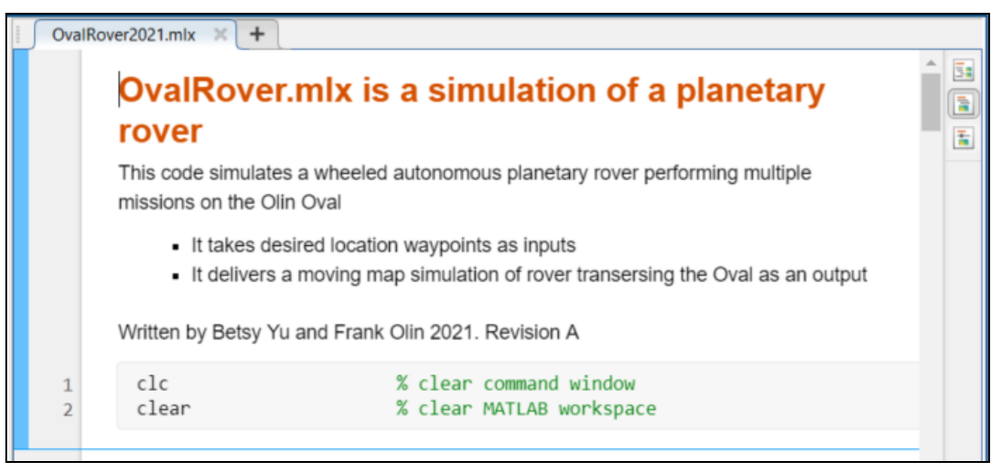
A downloadable larger pdf copy can be found on the ThinkLab canvas site as well as taped to the Think lab test car deck.



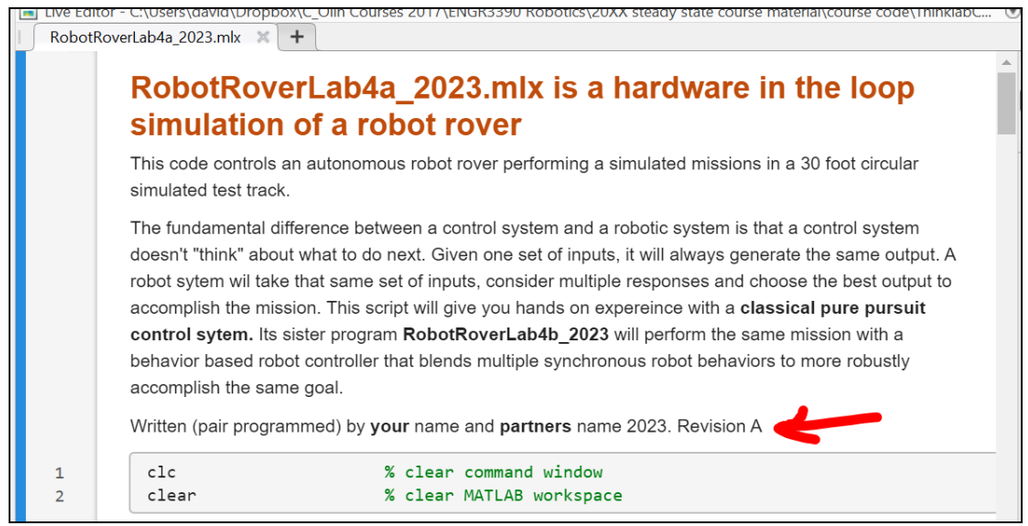
Create ThinkLab Robot Rover control code

The main **difference between a control system and a robot controller** is that in a control system *the connection between the sensor input and the actuator output is hardwired.* They tend to be brittle when dealing with unforeseen changes in the surrounding environment or unforeseen disturbances. In a robot, the sensor input is evaluated by a family of basic robot behaviors and their collective output is arbitrated to produce an optimal actuator output. **Control systems don't think, Robots do.** In order to get a first hand experience in the differences between a control system and a robot, for this hands-on tutorial, you will write two twin Matlab Scripts, the first will be a straight RoverRover pure pursuit way point controller. You will use it to crash into unplanned obstacles in a simulated oval test track . The second will copy the first code, make a few light changes to add a behavior based robot brain, and (hopefully) your Robot Rover will avoid those obstacles with ease. Shoot to complete one each week.

You can start by opening your copy of the Matlab robot code you wrote in week 2 (**OvalRover.mlx)** and save a copy of it (in your joint team MATLAB DRIVE depository) called **RobotRoverLab4a.mlx.** Just open up your old **Oval Rover**:



And save as **RobotRoverLab4a.mlx**. Working in pair programming teams of 2, please make code changes to first text section of the new code, as shown, to document its new functionality:



Reminder: **Most robot control code isn't written from scratch**. It is created by modifying existing code to meet a new need. We will do that here (and save you a lot of rewriting) by reusing most of your existing Oval Rover code structure for your hardware-in-the-loop RoverRover controller and the robot brain work to follow.

You will find in most of your future robot work, you will be either reusing code you have downloaded from an on-line repository, like GitHub, or reusing and rewriting code your research group or company has given you for the robot system you are developing on. In all cases robust documentation is key. **You don't write code for yourself, you write it for all of those robot engineers that follow in your footsteps!**

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

For this hands-on tutorial, we will have each team of pair programmers produce their own two person version of this first classical control code and informally demo it by week's end. By week 2 will then reform you into one bigger team and have each sub-team take on writing and deploying one of the Rover's new robot behaviors in one big set of demo code. More to come on that in the second section. For now, please modify the **code that runs once** section to look like this:

Set up Rover control system (code that runs once)

Set up your Arduino to control the rover. See function code for details, can take 5-60 seconds, be patient!

```
3  disp('Connecting to Rover Arduino over USB, note comport number must be set right');
4  [roverArduino, driveServo, steerServo, spinServo, ...
5      panServo, tiltServo, blinkLED, cam] = ArduinoSetup();
```

The second line calls a function to set up your Arduino hardware to act as the Rover's main robot controller. It's always good software design practice to encapsulate the hardware specifics in a **SetUp** function, so that if you upgrade the hardware, you can just change this function and the rest of the flight code is unaltered. Please code out the **ArduinoSetup** function, in the first section of the **Functions** repository at the end of your robot code template. See function code below:

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot control code should be as brief as possible and the bulk of the work should be done by functions. Functions also well support team programming as each sub-teamcan develop their functions independantly from the other sub-teams.

General Code Functions

```
130  function [roverArduino, driveServo, steerServo, spinServo, ...
131      panServo, tiltServo, blinkLED, cam] = ArduinoSetup()
132  % ArduinoSetup creates and configures rover Arduino to be a simple robot
133  % controller. It requires no inputs and returns Servo objects and parameters
134  % D. Barrett 2023 Rev A
135
136  % Create a global arduino object so that it can be used in functions
137  % NOTE: you must set COM# to match your specific hardware
138     roverArduino = arduino('COM4','Uno', 'Libraries', 'Servo');
139
140  % Set Blinky Alive light port
141      blinkLED = "D13";
```

After a bit of good function documentation, this code creates and configures the Arduino to support robot control operations. Next add this code for WebCam:

```
    blinkLED = "D13";

% Set up external webcamera on rover
    webcam = webcamlist;
    cam = webcam(2);
```

Then complete the setup function by creating the five servo objects; one for steering, one for drive wheels, one to make the lab cart table spin, one for camera pan and one for camera tilt. Then add a final bit of code to set them all to a neutral center position.

```
    cam = webcam(2);

% create servo objects driving PWM GPIO pin 3, pin 5, pin 6
% MinPulseDuration: 1120 (microseconds)  center 1520 (microseconds)
% MaxPulseDuration: 1920 (microseconds)
% s = servo(arduino, 'D3') creates a servo motor object
% with additional options specified by one or more Name, Value pair arg
    driveServo = servo(roverArduino, 'D3');
    writePosition(driveServo, 0.50);          % establish drive ESC center as zero speed
    steerServo = servo(roverArduino, 'D5');
    writePosition(steerServo, 0.42);          % always leave rotary servo at center
    spinServo = servo(roverArduino, 'D6');
    writePosition(spinServo, 0.50);           % establish drive ESC center as zero speed
    panServo = servo(roverArduino, 'D9');
    tiltServo = servo(roverArduino, 'D10');
```

Having created the servo objects attaching your code to the actual servo ports, it's always good practice, during controller setup, to move each actuator through its full range and then leave it in the centered position, when the code first runs. Checking the actuators at this point, will save you hours of debugging problems like; E-Stop button engaged, a loose connector, servo-connectors plugged in backwards, etc. Nominally, if an actuator works during setup, you can count on it working while you are writing and debugging the rest of your controller code.

The next bit of function code (see right) does that, sequentially verifying by moving, each of the primary actuators on the Rover.

Please note the final **end** completing the **ArduinoSetup** function. Then pause and appreciate that putting all of this complex Arduino setup detail down here in a modular function, both makes the main body of the control text much easier to read and also make the code much more portable, in that if we change control hardware, to get higher performance, from an Arduino Uno to a Teensy, we just need to rewrite this setup function and the rest of the robot code will work as written.

Heavily using functional modularity is the hallmark of professional code design. Run this section and make sure your Arduino is set up. When you run this first code section line by line you may notice that it may take about 10-12 seconds to connect and download code to the Arduino the first time (that is normal for Matlab downloading a server onto an embedded type controller in this class). It will download quicker the next time you update your code.

Please see course's instructors or TA's for the safe use of the lab equipment, specifically the Rover's E-Stop and battery charging infrastructure.

```
% R-C Mode expects to start up with joystick centered
% In the Arduino MATLAB function servo position is 0-1 so 0.5 is centered
% Move each axis through its full travel and then leave it centered

  writePosition(steerServo, 0.25);    % hard left
  pause(1.0);
  writePosition(steerServo, 0.65);    % hard right
  pause(1.0);
  writePosition(steerServo, 0.42);    % always leave servo-command at center
  pause(1.0);

  writePosition(driveServo, 0.55);    % slow reverse
  pause(1.0);
  writePosition(driveServo, 0.45 );   % slow forward
  pause(1.0);
  writePosition(driveServo, 0.43 );   % fast forward
  pause(1.0);
  writePosition(driveServo, 0.50);    % off  leave ESC off
  pause(1.0);

  writePosition(spinServo, 0.490);    % hard left
  pause(1.0);
  writePosition(spinServo, 0.510);    % hard right
  pause(1.0);
  writePosition(spinServo, 0.50);    % always leave ESC off
  pause(1.0);

  writePosition(panServo, 0.00);      % hard left
  pause(1.0);
  writePosition(panServo, 1.00);      % hard right
  pause(1.0);
  writePosition(panServo, 0.50);      % always leave servo-command at center
  pause(1.0);

  writePosition(tiltServo, 0.00);     % hard left
  pause(1.0);
  writePosition(tiltServo, 1.00);     % hard right
  pause(1.0);
  writePosition(tiltServo, 0.50);     % always leave servo-command at center
  pause(1.0);

  disp('Rover Arduino set up!');

end
```

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Having gotten your Arduino well setup to support the rest of your robot control code, continue to set up the matlab workspace to support developing rover control by creating a set of pop-up figures to display complex data, simply and graphically in the next code section. Please go back to the code that runs once section and modify/add the following:

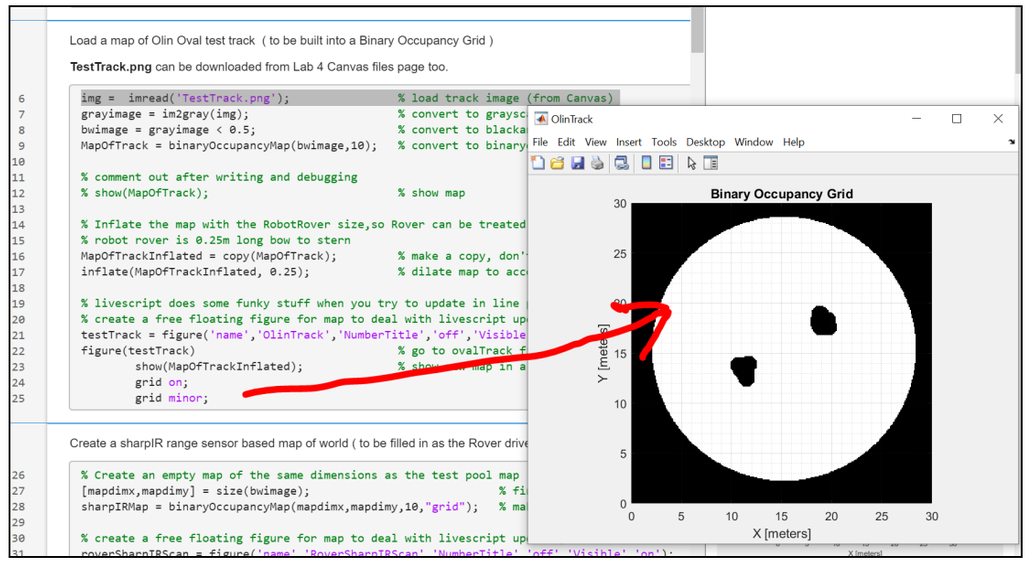
Load a map of Olin Oval test track (to be built into a Binary Occupancy Grid)

TestTrack.png can be downloaded from Lab 4 Canvas files page too.

```
6   img =  imread('TestTrack.png');                % load track image (from Canvas)
7   grayimage = im2gray(img);                      % convert to grayscale
8   bwimage = grayimage < 0.5;                     % convert to blackand white
9   MapOfTrack = binaryOccupancyMap(bwimage,10);   % convert to binaryoccupancymap in meters
10
11  % comment out after writing and debugging
12  % show(MapOfTrack);                            % show map
13
14  % Inflate the map with the RobotRover size,so Rover can be treated as a point. Assume
15  % robot rover is 0.25m long bow to stern
16  MapOfTrackInflated = copy(MapOfTrack);         % make a copy, don't lose original data
17  inflate(MapOfTrackInflated, 0.25);             % dilate map to accomidate robot body size
18
19  % livescript does some funky stuff when you try to update in line plots quickly, so
20  % create a free floating figure for map to deal with livescript update problem
21  testTrack = figure('name','OlinTrack','NumberTitle','off','Visible','on');
22  figure(testTrack)                              % go to ovalTrack figure for next plot
23          show(MapOfTrackInflated);              % show new map in a new figure window
24          grid on;
25          grid minor;
```

Note: You can download the round **TestTrack.png** file from the Lab 4 Canvas files folder and make sure to add it to the working directory your pair-team's MATLAB code is in. Run this section and get positive feedback that both your Arduino is hooked up and that there is a popup window figure of the simulated test track:

In this last code section, you modify your code to use a premade map of the simulated circular test track with two obstacles located in the middle of it. And it will generate a pretty, floating, stand alone figure of the rover test track, that you can position somewhere, not in the way, for future use.

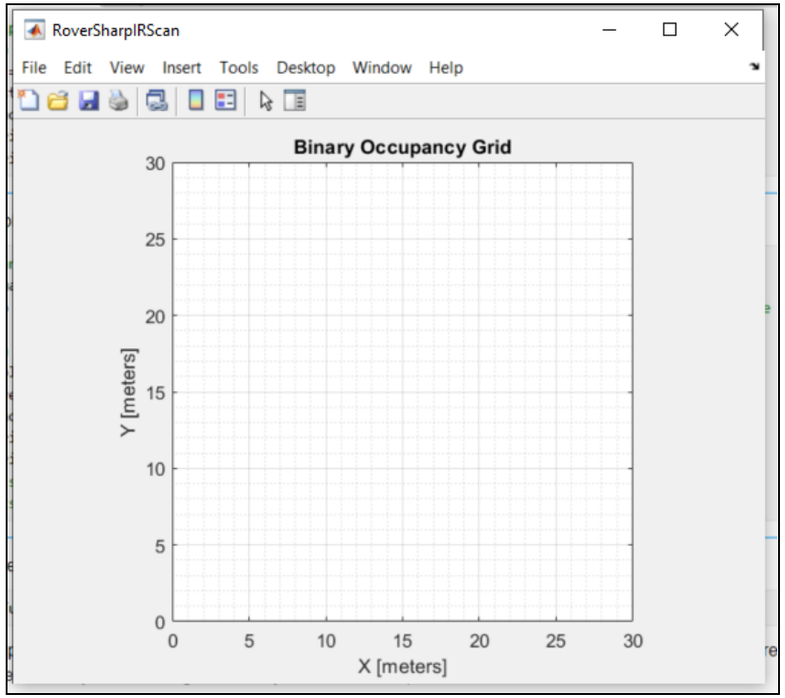


Next you will modify code to switch from a 360 degree scanning LIdar to the 6 discrete SharpIR range sensors (currently on the rover), so change your existing code to set up a new IR data figure to see this new data in the next section:

Create a sharpIR range sensor based map of world (to be filled in as the Rover drives around the Track)

```
25  % Create an empty map of the same dimensions as the test pool map
26  [mapdimx,mapdimy] = size(bwimage);                          % find size of map image
27  sharpIRMap = binaryOccupancyMap(mapdimx,mapdimy,10,"grid");   % make empty map the same size
28
29  % create a free floating figure for map to deal with livescript update problem
30  roverSharpIRScan = figure('name','RoverSharpIRScan','NumberTitle','off','Visible','on');
31  figure(roverSharpIRScan)
32          show(sharpIRMap);
33          grid on;
34          grid minor;
35  %  yes it should start out blank, it will get filled in as the rover drives and
36  %  collects data
```

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d



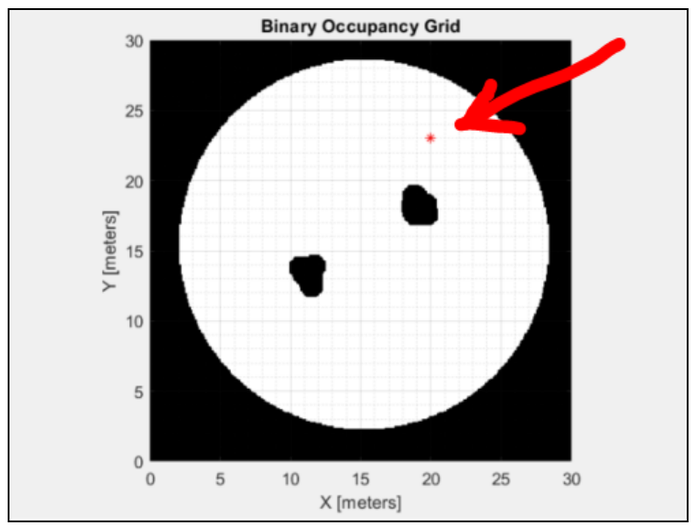
Moving on please modify the original rover control code to reflect the new Rover robots dynamic model as shown below:

```
Create a differential drive rover (this model takes robot vehicle speed and heading for inputs)
38  diffDriveRover = differentialDriveKinematics("VehicleInputs","VehicleSpeedHeadingRate");
Create a pure pursuit Rover controller (this controller generates the robots speed and heading needed to follow
a desired path, set desired linear velocity and max angular velocity in m/s and rad/s)
39  RoverController = controllerPurePursuit('DesiredLinearVelocity',2,'MaxAngularVelocity',3);
```

Next, please do a pretty heavy rewrite of the existing lidar code to remake **rangeSensor** shift from a 180 deg Lidar to a cluster of 6 discrete long range Sharp IR distance sensors. In the upcoming or previous SENSE lab, you will get (or got) hands-on expertease in wiring up, calibrating and using these very common low-cost range sensors. For this lab we are going to use a simulated version of them in the simulated test track. Please change code as is shown below:

```
Create 180 degree sharp IR range sensor suite for your rover.
  ▪ Create a sensor with a max range of 10 meters. This sensor simulates
  ▪ range readings based on a given pose and map. The Test Track map is used
  ▪ with this range sensor to simulate collecting sensor readings in an unknown environment
  ▪ Its a little hard to find documentation on rangeSensor function, so right click on it and choose "help" or f1

39  sharpIRPod = rangeSensor;                % creates a rangeSensor system object see help for info
40  sharpIRPod.Range = [0.1,10];             % sets minimum and maximum range of sensor
41  sharpIRPod.HorizontalAngle =[-pi/2, pi/2];   % sets max min detection angle
42  sharpIRPod.HorizontalAngleResolution = 0.628318;  % sets sharp pod resoultion to 6 sensors
43  testPose = [20 23 pi/2];                 % set an inital test pose for lidar
44
45  % Plot the test spot for lidar scan on the reference Test Track Map
46  figure(testTrack)                        % designate free floating poolTrackMap to draw on
47  hold on
48  plot(20,23, 'r*');                       % plot robot initial position in pool
49  hold off
```

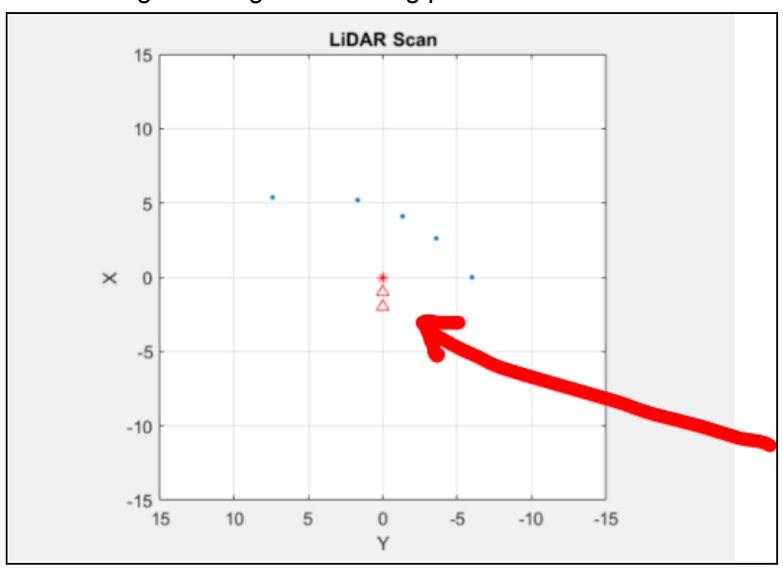
The unchanged part of code places Rover on the circular test track to give frame of reference for sharp lidar scan to come:

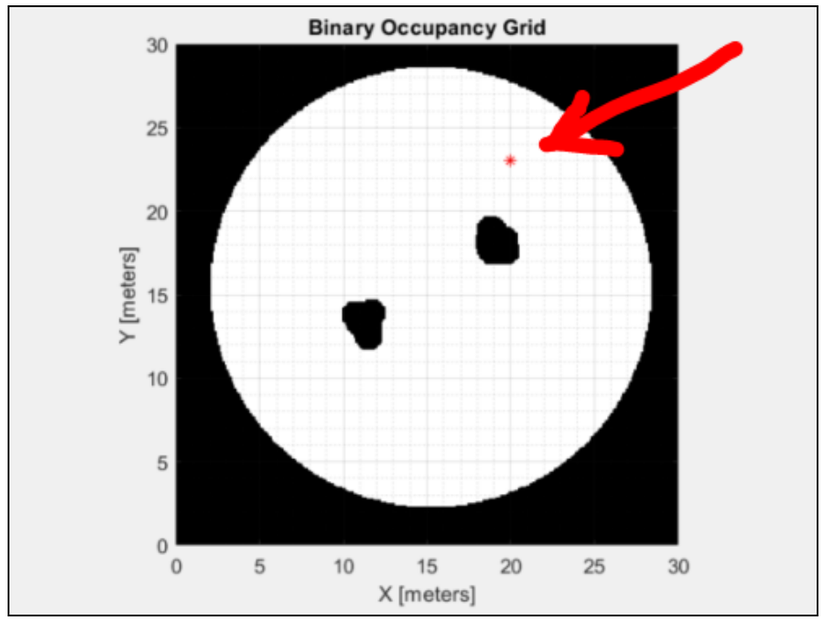


Then update the remaining code as shown to take a first test scan of the track around Rover:

```
49      hold off
50
51      % Generate a SharpIR test scan from test position.
52          [ranges, angles] = sharpIRPod(testPose, MapOfTrackInflated);
53          scan = lidarScan(ranges, angles);
54
55      % Visualize the test lidar scan data in robot coordinate system.
56      % create a new free-standing figure to display the local sharpIR range data,
57      % rover at center of plot
58          localSharpIRPlot=figure('Name','localSharpIRMap','NumberTitle','off','Visible','on');
59          figure(localSharpIRPlot)
60          plot(scan)                  % positive X forward, positive Y left
61          axis([-15 15 -15 15])
62          hold on;
63          plot(0,0, 'r*');            % plot robot position
64          plot(-1,-0, 'r^');          % plot robot position
65          plot(-2,-0, 'r^');          % plot robot position
66          hold off;
```

Run this section and generating the following plot:





Where you can clearly see 5 of the six SharpIR range beams getting valid range data from the nearest test track wall. Your Rover is designated by red icons at the center of the plot, heading to the top of the plot is what is directly in front of Rover. +X is forward of the Rover bow, -X to its stern. The red asterisk on the following plot shows you where your rover is currently physically located on the test track.

Next we will have you create two lists (vectors) waypoint paths coordinates for your rover to follow on this track. Many mobile robots use a list of sequential waypoints to drive to as the backbone of their mission planner. Such a list, moved into a table format with the behaviors that the robot should be co-executing while driving from waypoint to waypoint, is often called a **Mission Definition File**.

We will hardcode the waypoint paths here for speed, but you can foresee the utility of placing them in an external data structure (or file) so that you won't need to edit the rovers flight code for every unique mission.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

The first list of waypoints is a clear circumnavigation of the track (i.e. when looking for an orange cone). The second will take you smack through an uncharted obstacle. Please comment the second out for now, get your whole system working and then go back and run the second path to see what happens. Change your team's code to :

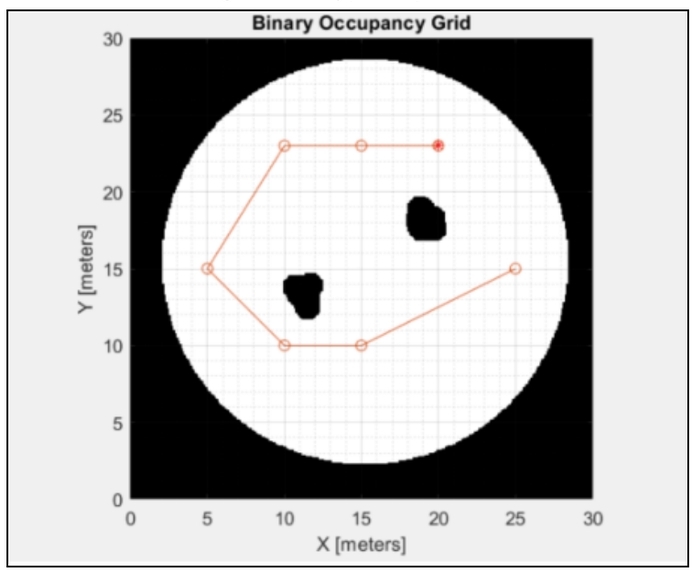
Create a test path of waypoints to sail through the pool for gathering range sensor readings.

```
68  path = [ 20 23; 15 23; 10 23; 5 15; 10 10; 15 10; 25 15];    % clear path waypoints
69  % path = [ 20 23; 15 23; 10 23; 5 15; 15 12; 20 10; 25 15];  % crash path waypoints
70
71  % Plot the path on the reference map figure.
72  figure(testTrack)                                  % select testPoolMap
73  hold on
74  plot(path(:,1),path(:,2), 'o-');                   % plot path on it
75  hold off
```

Use this test path as the set of waypoints the pure pursuit controller will follow:

```
76  RoverController.Waypoints = path;                % set Rover waypoints
```

Generating the clear circumnavigation waypoint path:



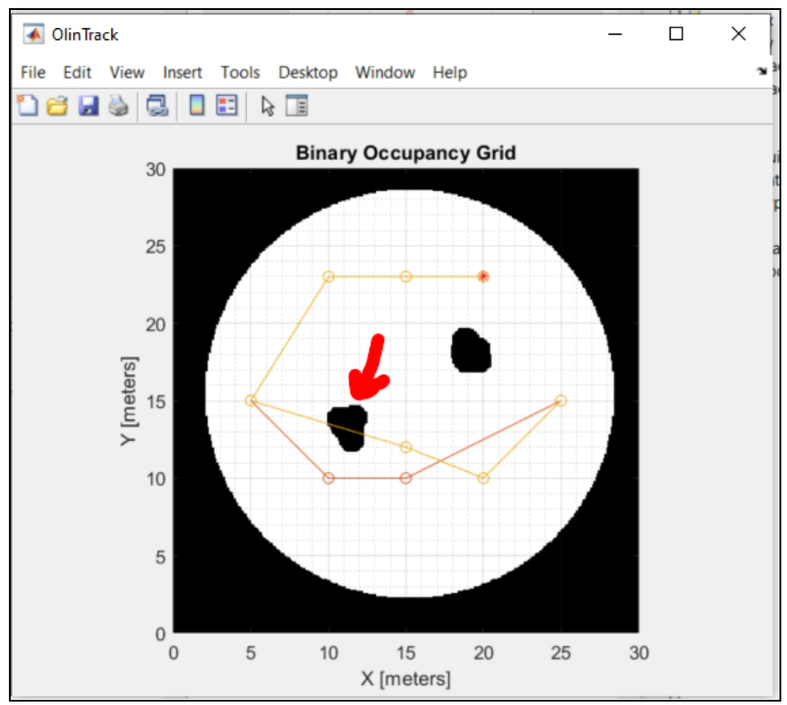
Swap the two waypoint paths and rerun:

Create a test path of waypoints to sail through the pool for gathering range sensor readings.

```
67  % path = [ 20 23; 15 23; 10 23; 5 15; 10 10; 15 10; 25 15];    % clesr path waypoints
68  path = [ 20 23; 15 23; 10 23; 5 15; 15 12; 20 10; 25 15];  % crash path waypoints
69
70  % Plot the path on the reference map figure.
71  figure(testTrack)                                  % select testPoolMap
72  hold on
73  plot(path(:,1),path(:,2), 'o-');                   % plot path on it
74  hold off
```

Use this test path as the set of waypoints the pure pursuit controller will follow:

```
75  RoverController.Waypoints = path;                % set Tug waypoints
```



Ouch, that second path is going to hurt! Please switch back to the safe path.

Add a final line to this section that loads these waypoints into the Rovers controller:

```
74  hold off

    Use this test path as the set of waypoints the pure pursuit controller will follow:

75  RoverController.Waypoints = path;                % set Tug waypoints
```

Make some light naming text changes to the next section (go code reuse!):

Set the initial pose and final goalWayPoint location based on the path. Create global variables for storing the current pose and an index for tracking the iterations.

```
initRoverPose = [path(1,1) path(1,2), pi/2]; % store intial loaction and orientation of rover
goalWayPoint = [path(end,1) path(end,2)]';   % path end waypoint
sampleTime = 0.10;                           % Sample time [s]
t = 0:sampleTime:100;                        % Time array
robotPoses = zeros(3,numel(t));              % Pose matrix
robotPoses(:,1) = initRoverPose';            % store robot robotPoses
r = rateControl(1/sampleTime);               % reset control loop  rate
reset(r);                                    % reset loop time to zero
aliveLED = true;                             % create a blinky LED variable to toggle
```

Ready to run robot, ask operator if they are set to run rover and suggest they move plots to a visable position

```
runRover = input('Ready to run Rover? Arrange your plots for visability, then type 1 and hit Enter ');
clc                                          % clean up cmmand window
```

And you have finished revising **your code that runs once. Woo-Hoo, well done!**

Pause to reflect on this just a bit. By reusing your previous well-structured, well-documented code (originally created for a whole different robot), you have just saved yourself and the team about a week or so of hard, heavy new code generation and debugging.

This is why it is so critically important to write well documented code for future use (by yourself and others), employing the twin tools of a **clean understandable common control structure** and heavy **clear embedded-documentation**. To make your code usable to others, you should shoot for a 50-50 ratio of code to documentation.

Imagine the sucking swamp you would be in now, if you were doing an archaeological dig on messy spaghetti code you got from the previous team, in order to harvest some of it to build this new code on! ***It's really easy to write bad undocumented code***. Everyone does for a while. **Bad code is pain.**

Please choose to not subject your awesome teammates to bad code pain.
Moving on to the center **Rover Sense-Think-Act control loop**, first take a look at the following rewrite of the code that runs continually:

Next move on to the part of the Rover code that loops:

Run robot control loop (code that runs over and over)

Please note how compact, clear and readable this code is. Yours should be too!

```
controlIndex = 1;                                   % create a robot loop index
while (controlIndex < numel(t))                     % loop for number of elements in t
    writeDigitalPin(roverArduino,blinkLED,aliveLED);   % blink alive LED
    position = robotPoses(:,controlIndex)';         % rovers current position
    roverLocation = position(1:2);                  % current rover X and Y from position

    % End loop if rover has reached goal waypoint within tolerance of 1.0m
        dist = norm(goalWayPoint'-roverLocation);
        if (dist < 1.0)                             % robot reaches end goal
            disp("Goal position reached")
            break;                                  % stop control loop
        end

    % SENSE: collect data from robot lidar
    [ranges, angles] = SENSE(position, MapOfTrackInflated, sharpIRMap, ...
        roverSharpIRScan, localSharpIRPlot, ...
        sharpIRPod);

    % THINK: compute what robot should do next
    [roverX,roverY,vRef, wRef,robotPoses] = THINK(robotPoses,diffDriveRover, ...
        RoverController,sampleTime, controlIndex);

    % Check to see if Rover is hitting a physical obstacle
        crash = checkOccupancy(MapOfTrackInflated,[roverX roverY]);
        if (crash == 1)
           disp('Rover crashed, all stop!');
           break;
        end

    % ACT: command robot actuators
    ACT(roverX, roverY, vRef, wRef, driveServo, steerServo, spinServo, testTrack);

    controlIndex = controlIndex + 1;                % increment control loop index
    waitfor(r);                                     % wait for loop cycle to complete
    aliveLED = ~aliveLED;                           % toggle blinky alive LED

end
beep                                % give audio indication waypoint is reached
```

First notice that the main **Sense**, **Think** and **Act** functions got a lot longer. We are explicitly passing all the individual variables into and out of functions here for clarity while you are just getting started with learning to program. There are more elegant ways to bundle these variables into data structures, or arrays that will make the code more compact (one argument in, one argument out) but that compactness of expression comes with the downside of much less clarity. It adds a step where you need to decode what variables each structure carries and how to work with them. As you complete the MATLAB software tutorials at the end of this write up, your skill with those tools will increase and we can start using data structures to make this type of code a bit less wordy.

Let's go through this control loop code step by step and explore what each function does.

First let's take a look at the Sense function:

Sense Functions (store all Sense related local functions here)

```
function  [ranges, angles] = SENSE(position, mapOfTrack, blankSharpMap, fig_SharpMap, fig_localSharpPlot, lidar)
% SENSE scans the reference map using the range sensor and the current pose.
% This simulates normal range readings for driving in an unknown environment.
% Update the sharp map with the range readings.
% inputs are  rover position, mapOfTrack, blank sharpMap, figure to plot global sharp IR data,
% figure to plot local sharp IR, sensor name
% outputs are N sharp IR array ranges and angles in local coordinate system
% Betsy Yu 2021  Rev C

    [ranges, angles] = lidar(position, mapOfTrack);
    scan = lidarScan(ranges,angles);
    validScan = removeInvalidData(scan,'RangeLimits',[0,lidar.Range(2)]);
    insertRay(blankSharpMap,position,validScan,lidar.Range(2));

    figure(fig_SharpMap);
        hold on;
        show(blankSharpMap);
        grid on;
        grid minor;
        hold off;

    figure(fig_localSharpPlot);    % positive X is forward, positive Y left on graph
    plot(scan);
    axis([-15 15 -15 15])
    hold on;
    plot(0,0, 'r*');               % plot robot position
    plot(-1,-0, 'r^');             % plot robot position
    plot(-2,-0, 'r^');             % plot robot position
    hold off;

end
```

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

The **SENSE** function takes in the map of the pool, the Rovers position and the Sharp IR range sensor array configuration. It gives back a set of 6 sharp IR ranges at the 6 angles the sensors are mounted at, in the array on the Rover's bow. It also plots that data both on the global map of all recorded sharp range data and a local map of what those ranges look like from the Rover itself. You will need the second map to design behaviors for your Rover in the next part of this tutorial work. Please enter and modify your code to look like the code shown above.

Next up, let us look at the **THINK** function. In all candor it's not really doing any real thinking. There is no deciding between options or path optimizations, it's just a straightforward control system running a standard **pure pursuit algorithm** to produce Rover actuation commands for the Act function to carry out. If you command Rover into a wall, by mixing up a few waypoints, it will happily steam full speed into that wall and crash.

Please modify your existing **Think** function to look like this:

Think Functions (store all Think related local functions here)

```
function [roverX,roverY,vRef,wRef, poses] = THINK(poses,diffDrive, ppControl,sampleTime, loopIndex)
% THINK gets control commands from pure pursuit controller
% to drive to next waypoint. Calculates derivative of robot motion
% based on control commands.
% inputs are  rover poses, diffDrive dynamics, loop sampleTime, loop index
% outputs are roverX, roverY, robot poses (position of robot)
% Frank Olin 2021  Rev B

        % Run the Pure Pursuit controller and
        % convert output to wheel speeds
        [vRef,wRef] = ppControl(poses(:,loopIndex));

        % Perform forward discrete integration step
        vel = derivative(diffDrive, poses(:,loopIndex), [vRef wRef]);
        poses(:,loopIndex+1) = poses(:,loopIndex) + vel*sampleTime;

        % Update rover location and pose
        roverX = poses(1,loopIndex+1);
        roverY = poses(2,loopIndex+1);

end
```

This new **Think** function will both calculate the new position and orientation of the Rover, but it now also calculates the **vRef,** the desired commanded linear velocity and the **wRef**, the desired commanded angular velocity needed to get to the next waypoint and returns them to the main control program. Both will be used as inputs to the Act function to control the Rover's speed and its steering angle.

Finally take a look at the new **ACT** function. The original function did a good job of plotting the Rover's position on the map of the track. Because we are going to do hardware in the loop simulation, you will need to add a few more operations to command the Arduino controller to drive the rover's speed controller and move its wheels to the correct angle. Note: Your Rover has a single RC type servo that is connected to the Rovers steering wheels via a linkage. It can move them from ~ -45 to 45 degrees with 0 degrees being straight on. The Rover also has an Axial ESC speed control that commands the motor driving all the wheels from full forward or reverse speed to full stop. In practice you'd develop calibration curves for both actuators that mapped them into the 0-1 control that the Matlab Arduino's servo function takes as an input. You will have either had or will get to have hands-on experience with doing both in the ACT lab test stand. For this lab, we will give you those functions to use without further background development.

Taking a look at the new **Act** function:

Act Functions (store all Act related local functions here)

```
function ACT(roverX, roverY, vRef, wRef, driveServo, steerServo, spinServo, fig_testTrack)
% ACT Increment the robot pose based on the derivative and drives Rover
% servos appropriately to simulate actual motion on a virtual test track.
% inputs are curretn X and Y position of rover and figure to plot rover in.
%
% outputs are no  outputs
% D Barrett 2023 Rev A

        % Plot Rover location on track
        figure(fig_testTrack)
        hold on
        plot(roverX,roverY, 'b*');              % plot robot position on track
        hold off
```

It now takes in the Rover's **vRef**, and **wRef** as commanded inputs as well as the objects naming the steerServo, driveServo and spinServo. The first part of the code draws the Rovers position on the test track map, the second uses the provided **servo objects** to drive the actual Rovers steering a drive wheel actuators so as the Rover goes about on its virtual mission on the simulated tank, the actual Rover will turn steering wheels, turntable and drive wheels to follow along in real life.

This form of **hardware in the loop simulation** is a really powerful way to both speed up code development and a safe way to try out new robot behaviors without needing to put on cold weather gear and chase your Rover through snow banks when a small bit of code goes wrong. Add this code to finish Act function:

```
        % Drive Rover hardware servo motors
        steerScale=[-3 wRef 3];             % build sim to real world scaling vector wRef -3 to 3 rad/sec
        driveScale=[0 vRef 2];              % build sim to real world scaling vector vRef 0 to 2 m/sec
        steerCom = rescale(steerScale);     % convert angular rate wRef to steer servo command from 0->1
        driveCom = rescale(driveScale);     % converts vRef to drive servo command from 0->1
        if (vRef > 0.5)
         turnTableCom= rescale(steerScale); % converts steer angle into turntable servo command 0->1
        else
          turnTableCom = 0.5;               % stop turntable servo if drive servo is off
        end

        % scale simcomand based on real world limits [low speed,comspeed, highspeed]
        % servo command in center of xCom vector [ 0 , comRate, 1 ]
        writePosition(steerServo, steerCom(2));         % command steer servo to run 0-1
        writePosition(driveServo, driveCom(2));         % command drive servo to run 0-1
        writePosition(spinServo, turnTableCom(2));      % command turntable servo to run 0-1

end
```

Don't stress about this code now, just use it. We will cover it, and how it works in the ACT lab. If you've taken the ACT lab, you can go in and modify this code, at will, to improve your rover's performance.

Hardware in-the-loop control system testing was pioneered by the space and aircraft industry, where the cost of small failures is incredibly high, but it is now becoming more and more common both throughout robot development and in general industrial control as well.

Returning to the main control loop:

Run robot control loop (code that runs over and over)

Please note how compact, clear and readable this code is. Yours should be too!

```
controlIndex = 1;                                   % create a robot loop index
while (controlIndex < numel(t))                     % loop for number of elements in t
    writeDigitalPin(roverArduino,blinkLED,aliveLED);% blink alive LED
    position = robotPoses(:,controlIndex)';         % rovers current position
    roverLocation = position(1:2);                  % current rover X and Y from position

    % End loop if rover has reached goal waypoint within tolerance of 1.0m
        dist = norm(goalWayPoint'-roverLocation);
        if (dist < 1.0)                             % robot reaches end goal
            disp("Goal position reached")
            break;                                  % stop control loop
        end

    % SENSE: collect data from robot lidar
    [ranges, angles] = SENSE(position, MapOfTrackInflated, sharpIRMap, ...
        roverSharpIRScan, localSharpIRPlot, ...
        sharpIRPod);

    % THINK: compute what robot should do next
    [roverX,roverY,vRef, wRef,robotPoses] = THINK(robotPoses,diffDriveRover, ...
        RoverController,sampleTime, controlIndex);

    % Check to see if Rover is hitting a physical obstacle
        crash = checkOccupancy(MapOfTrackInflated,[roverX roverY]);
        if (crash == 1)
           disp('Rover crashed, all stop!');
           break;
        end

    % ACT: command robot actuators
    ACT(roverX, roverY, vRef, wRef, driveServo, steerServo, spinServo, testTrack);

    controlIndex = controlIndex + 1;                % increment control loop index
    waitfor(r);                                     % wait for loop cycle to complet
    aliveLED = ~aliveLED;                           % toggle blinky alive LED

end
beep                                % give audio indication waypoint is reached
```

This control loop consists of a Matlab **while** loop that will run continuously until the Rover gets close to its final waypoint and then the code in the **if end** statement * will **break** the loop and let the robot move through to a clean shutdown. **SENSE** uses the sharp Irs to see what is around Rover, **THINK** applies pure pursuit control law to

determine rudder and prop setting to best hit the next waypoint along the path and **ACT** drives rudder and propeller speed to get Rover there in a controlled optimized way.

Before running this to see what happens, let's write the final section of clean shutdown code, so that you clear memory and leave your laptop in a good state each time you do a code development run. Please modify your existing shutdown code to look like this.

Clean shut down

finally, with most embeded robot controllers, its good practice to put

all actualtors into a safe position and then release all control objects and shut down all

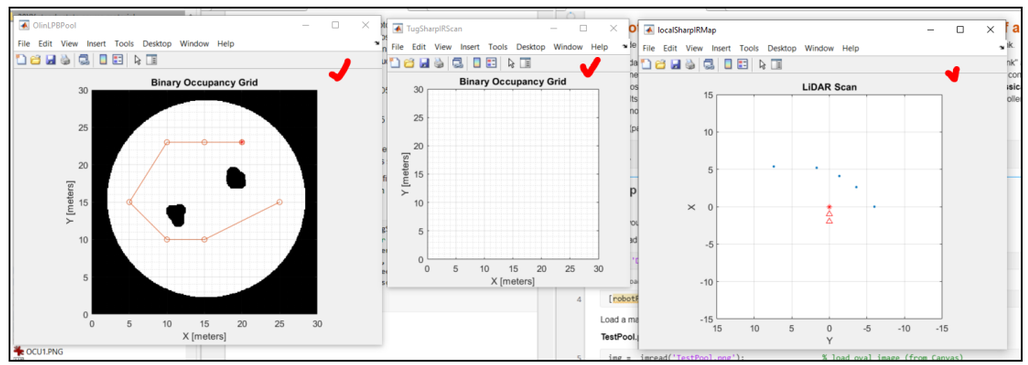
communication paths. This keeps systems form jamming when you want to run again.

```
% Stop program and clean up the connection to Arduino
% when no longer needed
  writePosition(driveServo, 0.5);      % always end servo at 0.5
  writePosition(steerServo, 0.5);      % always end servo at 0.5
  writePosition(spinServo, 0.5);       % always end servo at 0.5
  clc
  disp('Rover control program has ended');
  clear robotArduino

  wrapUP = input('Type 1 to delete figures and exit');
  close(testTrack);          % clear figure
  close(roverSharpIRScan);   % clear figure
  close(localSharpIRPlot);   % clear figure

  beep                % play sytem sound to let user know program is ended
  disp('All done!');
```

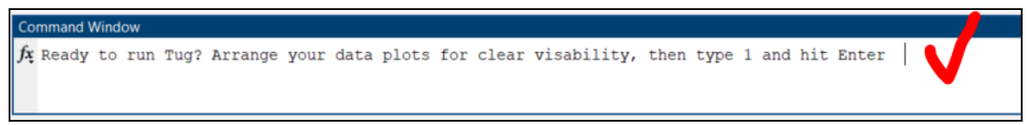
Robot Functions (store this codes local functions here)

Specifically, you always want to turn off or center all servos on a robot and you will want to clear up and free memory for all MATLAB figures created.

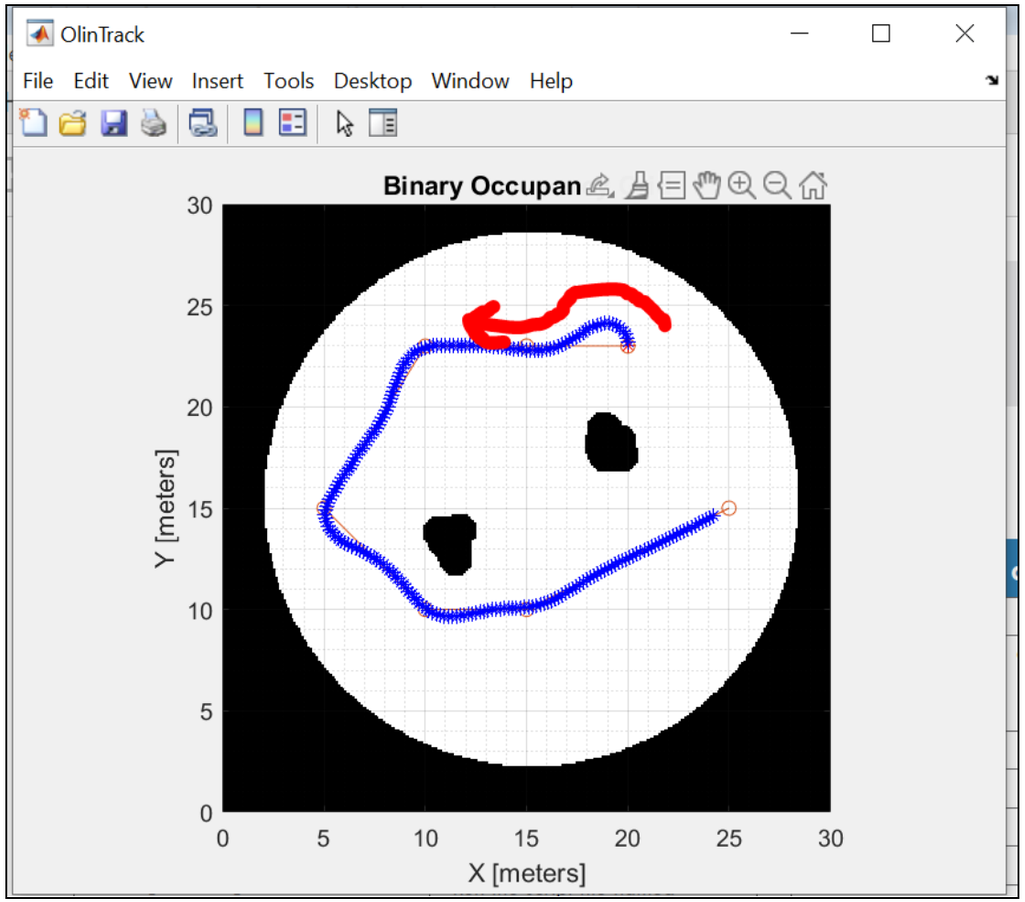
Ok, let's run your new Rover control code next and see what happens, make sure the Rover test bench is properly turned on, the Rover's E-Stop is disengaged, there is a Rover battery plugged in and then run your code!

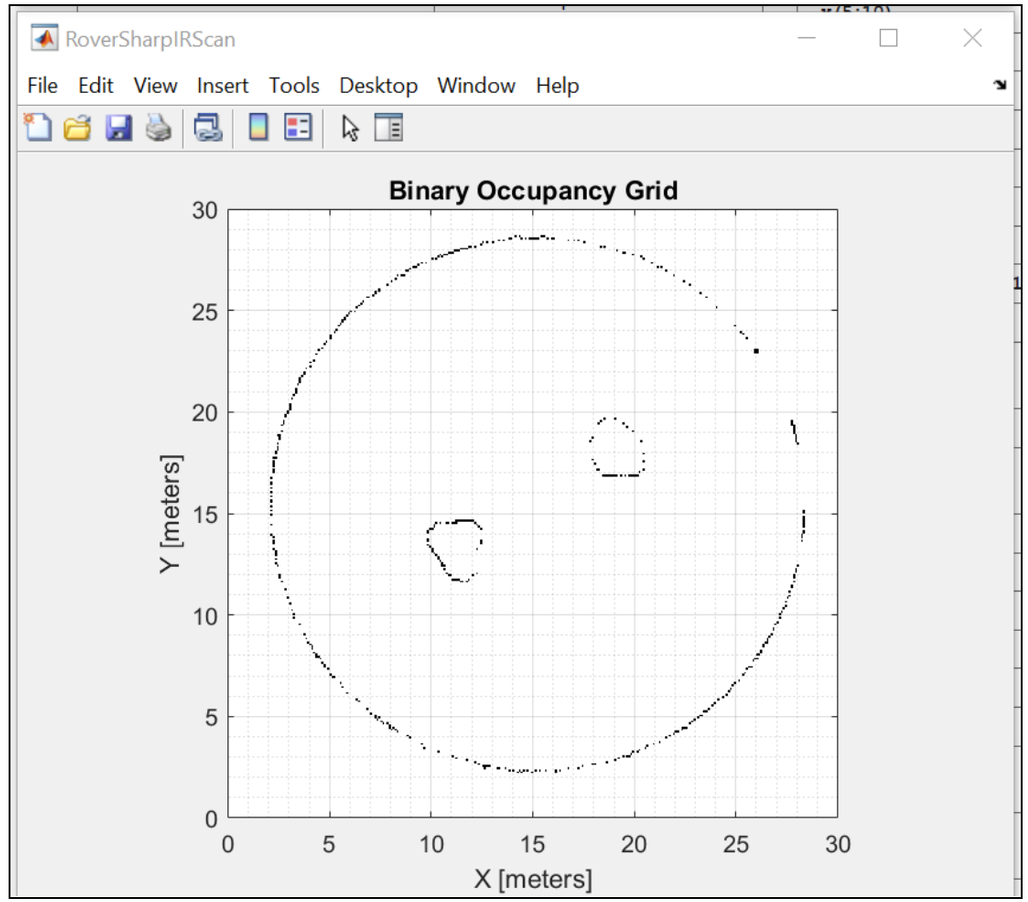


Your code should pop up an **OlinLPBPool** map, a **RoverSharpIRScan** map, and a **localSharpIRMap**, as shown above. And then pause to wait for your command to proceed.



Following that command, your Rover and its pure-pursuit control system should nicely follow the selected waypoints in a circumnavigation of the test track: The blue asterisk marks its discrete position at each time step. Leaving a wake-trail like this is a great way to help both understand and debug what your rover is doing at any instant in time and will prove critical to sorting out behaviors in the next section of this work.

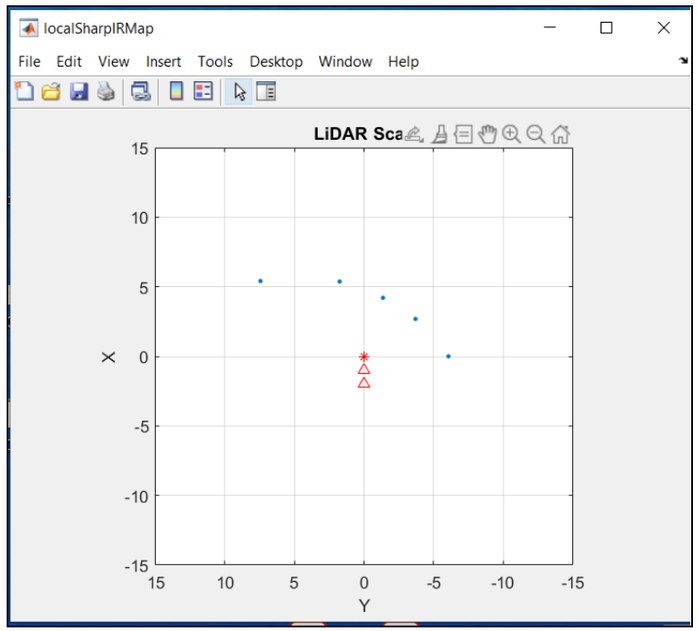




Note: You will want to take a screen capture of each experimental run's graph for the final report as the program will delete them when you exit it.

As well as got live data from your 6 forward pointing Sharp IR range sensors:

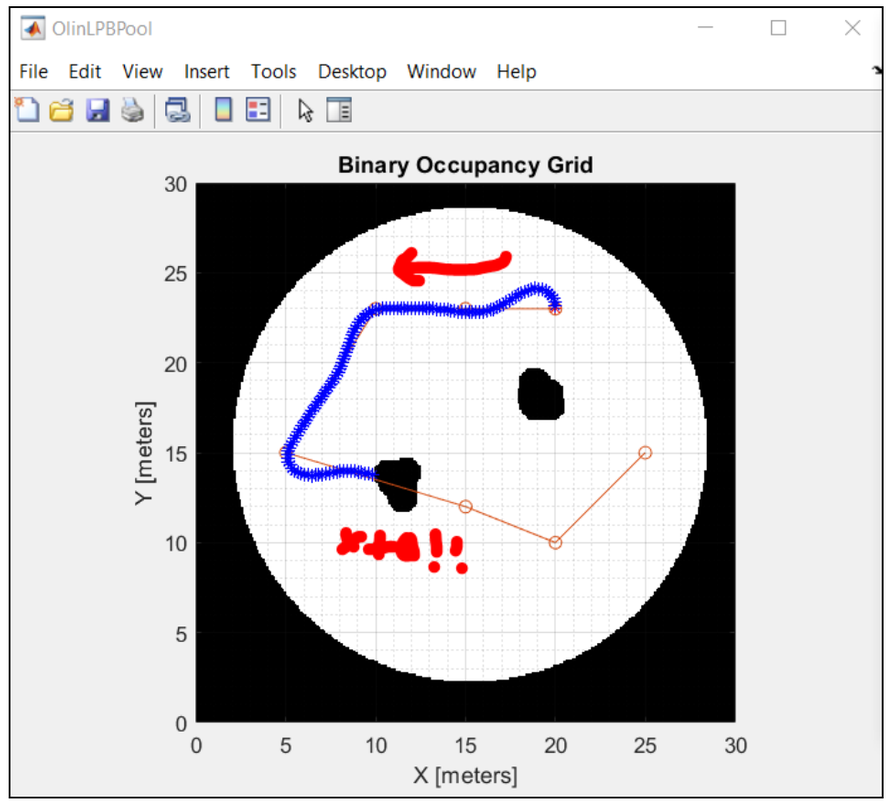
While it ran you also collected sharp range data and built up a pretty nice map of the track:



Run the same simulation a few times. Watch the Rover centered Sharp IR screen as your Rover drives around the track. You can develop a good sense of what the sharps IR sensors can and can't see after a few runs, which will help you to develop effective behaviors in the next section of this work. **Congratulations! You've built a great new Robot Rover control system.** If this were a control theory class you'd be done. *But it is not !* It's a Robot class, so go back into your code and comment out the first collection of waypoints and un-comment in the second set.

```
Create a test path of waypoints to sail through the pool for gathering range sensor readings.
64  %path = [ 20 23; 15 23; 10 23; 5 15; 10 10; 15 10; 25 15];    % clesr path waypoints
65  path = [ 20 23; 15 23; 10 23; 5 15; 15 12; 20 10; 25 15];  % crash path waypoints
66
```

Run your code again with the new Rover's path waypoints, And get:



WHAM !!! Congratulations again! You just crashed your Rover head-on, right into an immovable obstacle at full throttle.

This is a somewhat contrived, but graphically convincing, example of **the brittleness of using just a standard control system for a dynamic robot** that has to operate in an ever changing dynamic environment. Without perfect sensors and perfect knowledge of all potential obstacles in your operating space and knowledge of not just where they are, but how they are moving, a control algorithm by itself is highly susceptible to generating a set of legitimate commands that will cause a serious crash.

Creating a robust Behaviour Based Control for your Rover

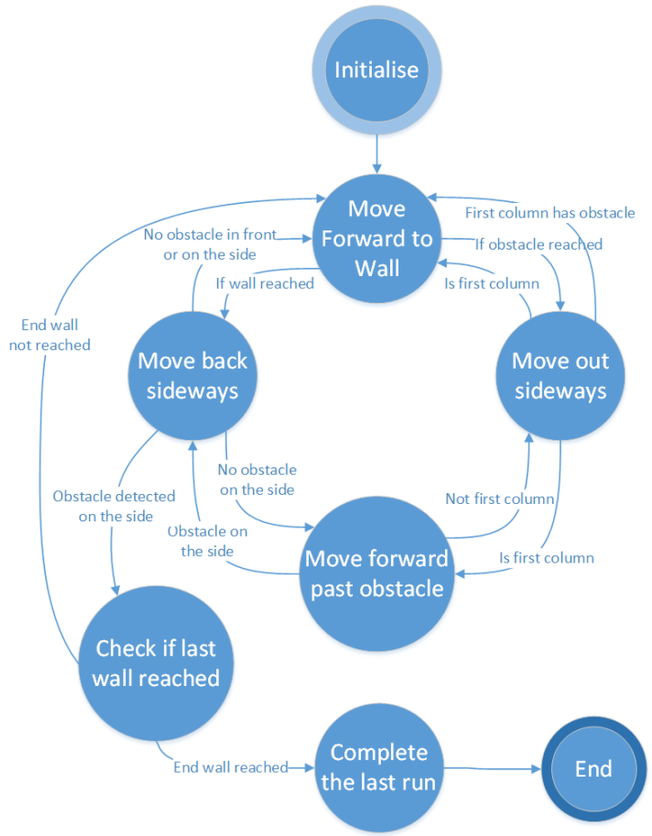
How you go about fixing this, crash into an obstacle problem, is the subject of the second part of this tutorial. Recollecting as a single programming group from the 4 individual pair teams that did the first section, we will ask you to next form one larger integrated team and have each subteam work on generating a part of the needed behavior based robot control code as is described below. Please read through the following section as individuals then form up into Behaviour-Function writing subteams, in any way that is comfortable for you..

Fundamentals of robot behaviors-based control

Complex robots usually use some form of multi-level control with behavioral control at a higher planning level and classical way point path following control at a lower actuator level.A robot can be conceptually thought of as an independent agent, that at any point in time can be in any one of a large number of **behavioral states.** What a robot state is doesn't need to have a precise definition, but it is usually tied to a set of easily observable actions, like wall-following, wandering, avoiding-obstacles, etc. A collection of all possible states that a robot can be in can be shown in a clear graphical form, using a specific type of block diagram, called a **Finite State Diagram**.

A **Finite State Diagram** traditionally shows not only the states a robot can be in, but also the events that transition the robot from one state to another. In practice, the robot's control system flows from one state to the next, along these event paths, either looping or branching at decision points, until the mission (captured in the FSM graph) is complete or an error that can't be overcome is reached. A **FSM** diagram is a very clear, graphical way to portray, not only what the robot is currently doing, and what steps are required to form a complete mission, but also clearly captures what your team of robot engineers designed (and hopes) your robot will do. **FSM** diagrams are also great tools to facilitate critical conversations among your future robot design team members, many of whom won't be active coders. Even a gearhead ME can understand them! In a very explicit way, if it is not in the FSM diagram, it doesn't need to be coded! And by extension, everything in the FSM machine needs to be coded by someone on your lab team.

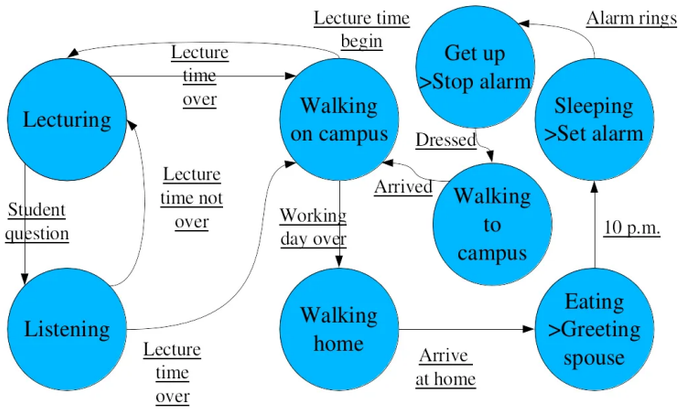
By carefully designing, then double checking, your team's FSM first and coding second, you are adhering to the "Measure Twice - Cut Once" school of design-build. In the process you are probably going to cut your coding workload in half and decrease team frustration by 80%. Planning before coding, really really helps!



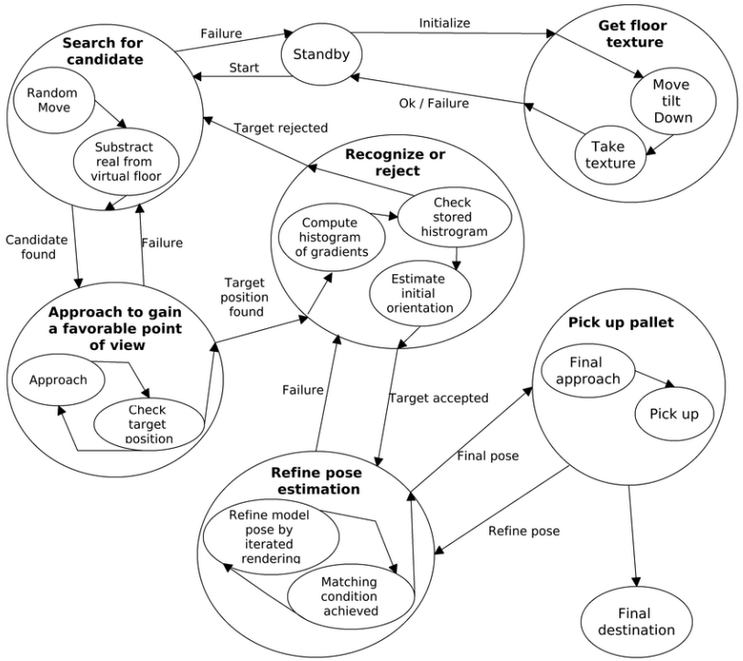
Example Robot Behavior FSM Diagram

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

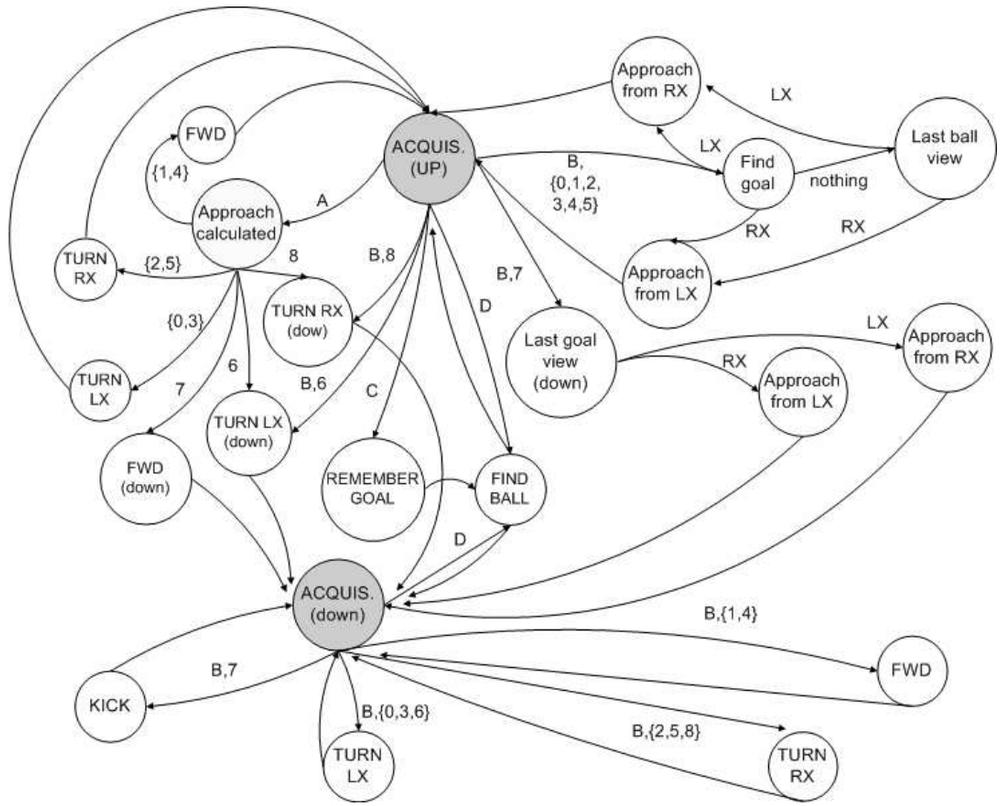
Let's look at a few Finite state machines for robots. Consider a Robot Professor:



A robot warehouse worker:



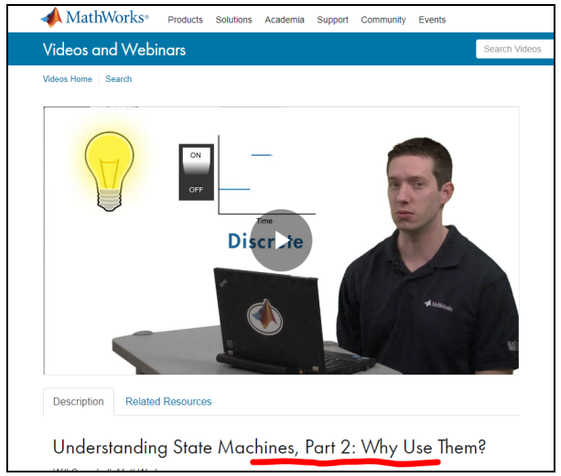
A Robot soccer player:



You can see the control flow here, in each of the above cases. The robot is in one of many possible states exhibiting a behavior, it stays in that state until an event occurs to move it to a new state. Over the course of a task or mission, the robot will cycle through many, but probably not all the possible states it could be in. Designing the right subset of states, not too many, not too few is the primary challenge behind developing a good robust robot controller for your Rover.

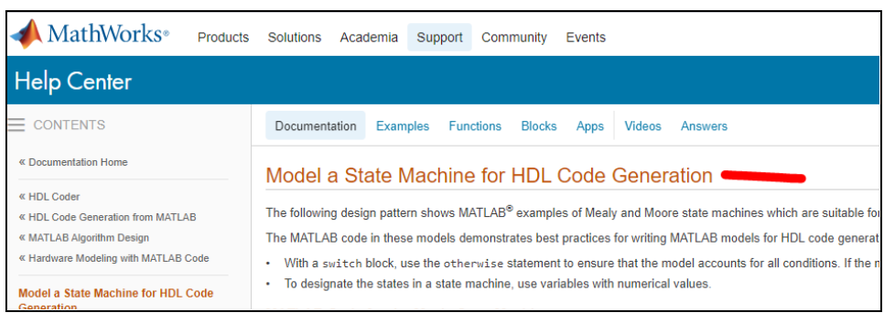
ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

I'll ask you to pause here and go look at the 4-short 3m video tutorials listed on the Canvas page for this tutorial, about what a finite state machine is and how to draw Mealy or Moore Finite State Machine diagrams before proceeding:

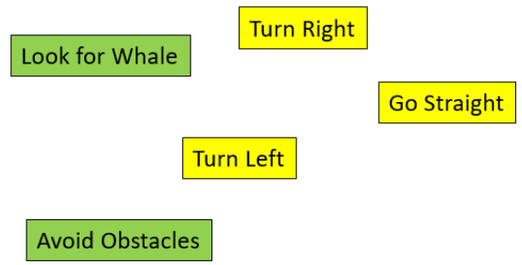


Next please take a quick look at how to code Mealy or Moore finite state machines in Matlab at this reference (summary, **switch** statements make good FSMs):

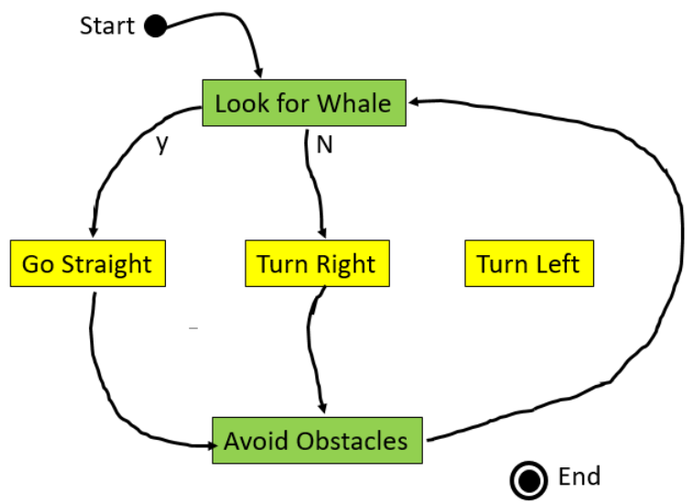
https://www.mathworks.com/help/hdlcoder/ug/model-a-state-machine-for-hdl-code-generation.html



Let's look at creating your first behavior based finite state machine, Consider a simple first case where the robot Rover has just five behaviors:

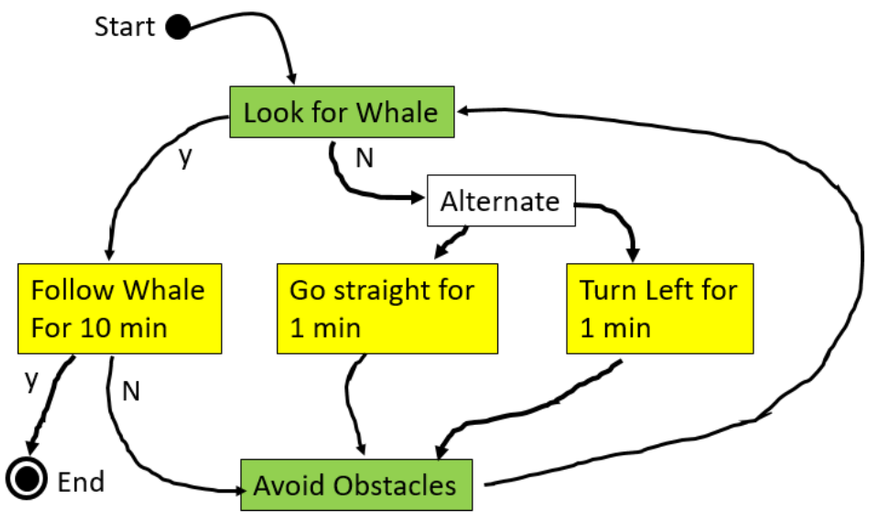


For clarity, we will color code behaviors as Green, for things Rover should always be doing, and Yellow for behaviors that Rover switches between. For example, there is no practical reason to not always look for the whale, but the Rover can only be going straight or turning, it can't be doing both. Let's arrange this into a Finite State Flow:



Here, after starting, the Rover looks for the whale, if it sees it, the **Rover goes straight** toward it, if not, **Rover turns to the right** looking for it. It's not a bad first pass at a FSM, but it has some deficiencies. We have a behavior we never use. If the Rover doesn't see the whale it just loops to the right in place forever and there is no pre-planned way to finish the mission.

Consider a set of modifications. Add a new behavior that follows the whale by centering it in camera, if the whale isn't seen, **Rover goes straight** for 1 min, then **Rover turns left** for a minute and repeats this random search through the pool until you do find the whale. If you successfully **follow the whale** for 10 minutes, you've collected enough data, call the mission off and let the poor Narwhal rest up a bit!



This is just one of many possible ways to approach an autonomous target-following mission. Your team will get to design your own behavior based FSM in the final section of this tutorial. But it has all of the necessary features to demonstrate what a FSM looks like and you can see one of the strengths of the FSM method. Everyone on the team can get a very clear common idea of what the robot can and should be doing, whether they wrote code for that part of the flow or not.

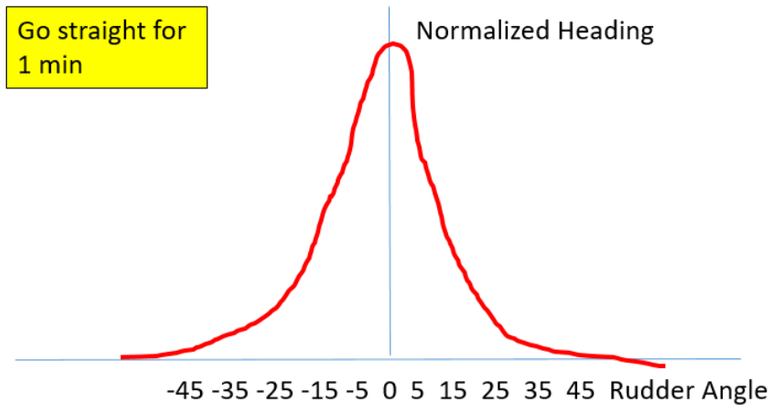
Looking at the diagram, its fairly clear you could just use a Matlab **Switch** statement

```
n = input('Enter a number: ');

switch n
    case -1
        disp('negative one')
    case 0
        disp('zero')
    case 1
        disp('positive one')
    otherwise
        disp('other value')
end
```

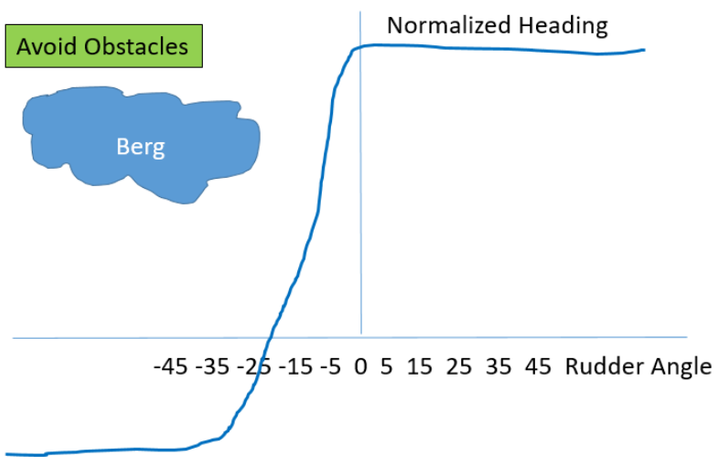
To toggle program flow between the Yellow (do one of, not all) behaviors. And placing each robot behavior into its own Matlab function, would keep your code clean, readable and modular. But one of the classical challenges confronting behavior based robotics is to now best merge commands from multiple behaviors into one set of actuator commands. In this case, we might need to merge the rudder commands for going straight, with the rudder commands for obstacle avoidance to avoid an iceberg right off the bow. This blending of the two desired behavior commands is often done in some form of **Behavior Arbiter.**

Let's look at **behavior arbitration** in a bit more detail. Consider that we create a behavior function for each exclusive yellow state shown in the diagram. In order to achieve robust, real-world tolerant behavior, these functions don't produce a single

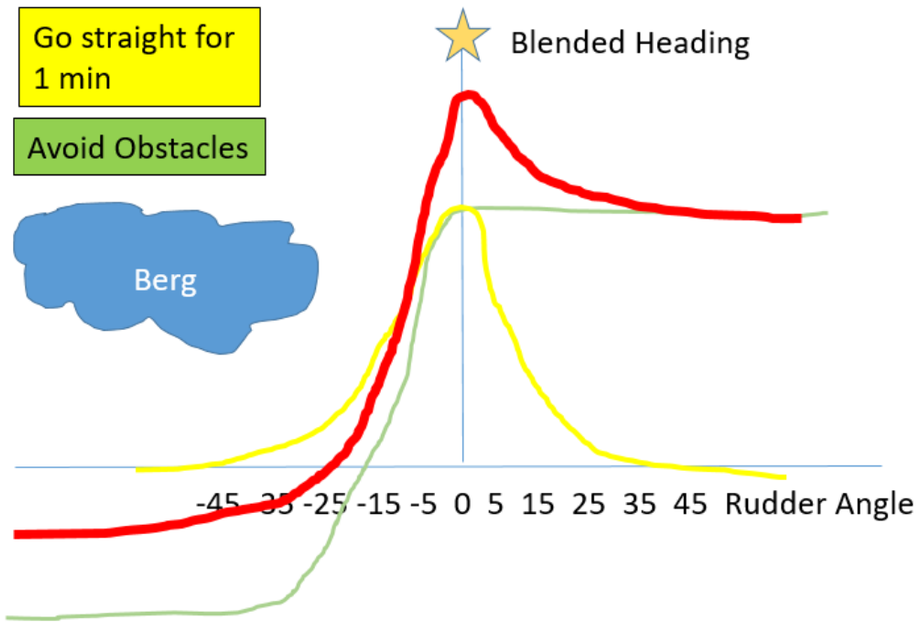
desired heading, but rather a probabilistic distribution of desired headings with the peak situated at the optimum heading, but with many surrounding possible headings being almost as good.Probabilistic distributions are robust.Take a look at Go straight:



Then consider Obstacle Avoidance behavior that will allow any rudder angle except for those that will cause a collision:



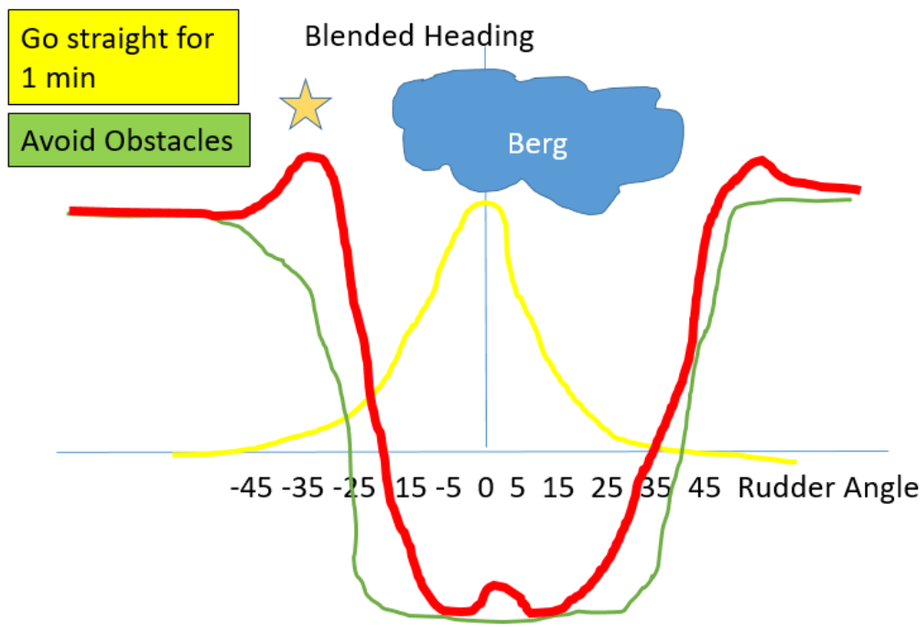
These probabilistic behavior curves are sometimes affectionately called **brain waves**. If we add the two brain waves together and then look for where the maximum peak is, we are in effect selecting the one best heading that satisfies the goal of both behaviors. **This is a simple behavior arbiter.**



By having each behavior generate a range of probabilistically weighted heading outputs, from more desirable to less, we build in a very robust response to any unknown, unplanned disturbances that the Rover encounters as it steams through the pool. If the iceberg drifts in front of the Rover away from its mapped position, this will seamlessly handle it. This type of arbiter is often called a weighted poll arbiter and it is central to the THINK part of many robot controllers. Using this arbiter, your Rover has moved one step up from a simple control system and is now weighing inputs from several competing behaviors to determine what rudder setting is the best

to send on to the Act servo control functions. **In some simple direct way, the Rover is thinking.**

Consider what would happen if we replicated the Rover being commanded by the **Go straight** behavior to crash into an iceberg, as we did in the controls part of this tutorial:



Consulting the behavior brainwave curves above, it can be seen that although the primary **Go Straight** behavior would prefer a heading that would impact the iceberg, it's also ok with heading to the left and right of it. The **Avoid Obstacles** behavior strongly prefers any heading except those that would intersect with the iceberg. When the two desired heading curves are added up into an arbiter curve and its maximum is selected, it can be seen that the Rover will safely steam around the iceberg either to its right or left depending on which peak is higher.

Moving this overall concept into Matlab's vector based toolkit is shockingly easy. If each Rover behavior is a function, then each function just needs to output a weighted vector of desired rudder angles at some reasonable granularity. For example: [0.0 0.1 0.2 0.3 1.0 0.3 0.2 0.1] = GoStraight();

Each active behavior creates its own vector of outputs to meet its own design goals.

The arbiter can do a simple vector addition of many incoming behavior outputs, find the maximum of that new blended vector and send the rudder angle that corresponds to that maximum of the weighted sums of all the behaviors inputs to the ACT code to drive the servos to that desired rudder angle.

In practice you could, and probably should, have one blending arbiter for each commanded servo motor.

In this case, one for the rudder and one for the propeller speed. For your terrestrial rover, one for steering and one for drive wheel speed. As a conceptual example, there can be many officers on your Rover's bridge yelling at the helmsperson to turn the ship's wheel to a certain heading, but the ship's steering wheel can only go to one angle. Your arbiter is the helmsperson that merges all the desired officer's (behavior's) rudder positions, into a single, arguably best, one that can be sent to the rudder's servo.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Let's move on and write the Matlab code to demonstrate a simple example of this. Open up your **RobotRoverLab4a_2023.mlx** and save a new copy as **RobotRoverLab4b_2023.mlx.**

Here again, we are going to use the power of building on working existing code, rather than trying to write this code all again from scratch.

Your full team can modify the first section as show:

RobotRoverLab4bFSM_2023.mlx is a hardware in the loop simulation of a FSM controlled robot rover

This code controls an autonomous robot rover performing a simulated missions in a 30 foot circular simulated test track.

The fundamental difference between a control system and a robotic system is that a control system doesn't "think" about what to do next. Given one set of inputs, it will always generate the same output. A robot sytem wil take that same set of inputs, consider multiple responses and choose the best output to accomplish the mission. This script will give you hands on expereince with a **FInite State Machine** using **Behavoirs and an Arbiter** to blend multiple synchronous robot behaviors to more robustly accomplish the same mission goal**.** Its sister program **RobotRoverLab4a_2023, did this mission with a simple pure pursuit controller.**

Written (pair programmed) by **your** name and **partners** name 2023. Revision A

```
clc                     % clear command window
clear                   % clear MATLAB workspace
```

You can leave most of the **Code that Runs Once** part intact, but comment out the path waypoint code as shown below (you might want to reuse it later for your team robot code to help build a waypoint following behavior). Reuse:

Set up Rover control system (code that runs once)

Set up your Arduino to control the rover. See function code for details, can take 5-60 seconds, be patient!

```
disp('Connecting to Rover Arduino over USB, note comport number must be set right');
[roverArduino, driveServo, steerServo, spinServo, ...
    panServo, tiltServo, blinkLED, cam] = ArduinoSetup();
```

Load a map of Olin Oval test track (to be built into a Binary Occupancy Grid)

And the section that makes the test track:

Load a map of Olin Oval test track (to be built into a Binary Occupancy Grid)

TestTrack.png can be downloaded from Lab 4 Canvas files page too.

```
img =  imread('TestTrack.png');                % load track image (from Canvas)
grayimage = im2gray(img);                      % convert to grayscale
bwimage = grayimage < 0.5;                     % convert to blackand white
MapOfTrack = binaryOccupancyMap(bwimage,10);   % convert to binaryoccupancymap in meters

% comment out after writing and debugging
% show(MapOfTrack);                            % show map

% Inflate the map with the RobotRover size,so Rover can be treated as a point. Assume
% robot rover is 0.25m long bow to stern
MapOfTrackInflated = copy(MapOfTrack);         % make a copy, don't lose original data
inflate(MapOfTrackInflated, 0.25);             % dilate map to accomidate robot body size

% livescript does some funky stuff when you try to update in line plots quickly, so
% create a free floating figure for map to deal with livescript update problem
testTrack = figure('name','OlinTrack','NumberTitle','off','Visible','on');
figure(testTrack)                              % go to ovalTrack figure for next plot
        show(MapOfTrackInflated);              % show new map in a new figure window
        grid on;
        grid minor;
```

Just go through line by line and make sure there are no small discrepancies. Then run and add each code section edit in sequence, debugging as you go.Create Sharp sensor array:

Create a sharpIR range sensor based map of world (to be filled in as the Rover drives around the Track)

```
% Create an empty map of the same dimensions as the test pool map
[mapdimx,mapdimy] = size(bwimage);                          % find size of map image
sharpIRMap = binaryOccupancyMap(mapdimx,mapdimy,10,"grid");   % make empty map the same size

% create a free floating figure for map to deal with livescript update problem
roverSharpIRScan = figure('name','RoverSharpIRScan','NumberTitle','off','Visible','on');
figure(roverSharpIRScan)
        show(sharpIRMap);
        grid on;
        grid minor;
%  yes it should start out blank, it will get filled in as the rover drives and
%  collects data
```

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Check through the next section:

Create a differential drive rover (this model takes robot vehicle speed and heading for inputs)

```
diffDriveRover = differentialDriveKinematics("VehicleInputs","VehicleSpeedHeadingRate");
```

Create a pure pursuit Rover controller (this controller generates the robots speed and heading needed to follow a desired path, set desired linear velocity and max angular velocity in m/s and rad/s)

```
RoverController = controllerPurePursuit('DesiredLinearVelocity',2,'MaxAngularVelocity',3);
```

And the next:

Create 180 degree sharp IR range sensor suite for your rover.

- Create a sensor with a max range of 10 meters. This sensor simulates
- range readings based on a given pose and map. The Test Track map is used
- with this range sensor to simulate collecting sensor readings in an unknown environment
- Its a little hard to find documentation on **rangeSensor** function, so right click on it and choose "help" or f1

```
sharpIRPod = rangeSensor;        % creates a rangeSensor system object see help for info
sharpIRPod.Range = [0.1,10];                  % sets minimum and maximum range of sensor
sharpIRPod.HorizontalAngle =[-pi/2, pi/2];        % sets max min detection angle
sharpIRPod.HorizontalAngleResolution = 0.628318;  % sets sharp pod resoultion to 6 sensors
testPose = [20 23 pi/2];                          % set an inital test pose for lidar

% Plot the test spot for lidar scan on the reference Test Track Map
figure(testTrack)                       % designate free floating poolTrackMap to draw on
hold on
plot(20,23, 'r*');                      % plot robot initial position in pool
hold off

% Generate a SharpIR test scan from test position.
    [ranges, angles] = sharpIRPod(testPose, MapOfTrackInflated);
    scan = lidarScan(ranges, angles);

% Visualize the test lidar scan data in robot coordinate system.
% create a new free-standing figure to display the local sharpIR range data,
% rover at center of plot
    localSharpIRPlot=figure('Name','localSharpIRMap','NumberTitle','off','Visible','on');
    figure(localSharpIRPlot)
    plot(scan)                  % positive X forward, positive Y left
    axis([-15 15 -15 15])
    hold on;
    plot(0,0, 'r*');            % plot robot position
    plot(-1,-0, 'r^');          % plot robot position
    plot(-2,-0, 'r^');          % plot robot position
    hold off;
```

Comment out waypoint code for now, but save (you may want it later):

Create a test path of waypoints to sail through the pool for gathering range sensor readings.

```
% path = [ 20 23; 15 23; 10 23; 5 15; 10 10; 15 10; 25 15];    % clear path waypoints
% path = [ 20 23; 15 23; 10 23; 5 15; 15 12; 20 10; 25 15];  % crash path waypoints

% Plot the path on the reference map figure.
% figure(testTrack)                                    % select testPoolMap
% hold on
% plot(path(:,1),path(:,2), 'o-');                     % plot path on it
% hold off
```

Use this test path as the set of waypoints the pure pursuit controller will follow:

```
% RoverController.Waypoints = path;                  % set Rover waypoints
```

Modify code and comment out the **goalWayPoint** code, in the next section as shown:

Set the initial pose and final goalWayPoint location based on the path. Create global variables for storing the current pose and an index for tracking the iterations.

```
initRoverPose = [20, 23, pi/2];    % store intial loaction and orientation of rover
%goalWayPoint = [path(end,1) path(end,2)]';   % path end waypoint
sampleTime = 0.10;                            % Sample time [s]
t = 0:sampleTime:100;                         % Time array
robotPoses = zeros(3,numel(t));               % Pose matrix
robotPoses(:,1) = initRoverPose';             % store robot robotPoses
r = rateControl(1/sampleTime);                % reset control loop  rate
reset(r);                                     % reset loop time to zero
aliveLED = true;                              % create a blinky LED variable to toggle
```

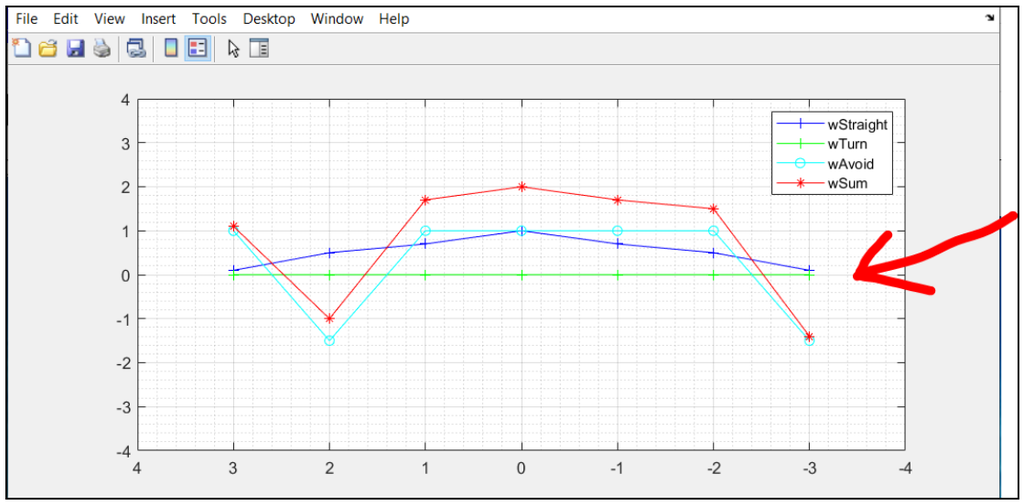
Then add a new section to create the figure to hold your robot's **brainwaves** (Do this underneath the section where you set the initial goalWaypoint location and created global variables, but before you run your robot), see code below:

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Create a free floating figure for robot behavior brainwave display

```
% create a free floating figure for map to deal with livescript update problem
brainWaves = figure('name','Behavior Brain Waves','NumberTitle','off','Visible','on');
figure(brainWaves)                              % go to brainWave figure for next plot
        pbrain = plot( [ -3 -2 -1 0 1 2 3], [ 0 0 0 0 0 0 0]);
        axis([-4 4 -4 4]);
        grid on;
        grid minor;
```

Which will be critical for **debugging a moving robots behaviors** and states and will look like this:



Before you move on, please modify your closing code to clean up this new Brainwave figure when you no longer need it. It's good practice to deconstruct objects rigorously right after you construct them, so as to not forget to do it and leave your team with a messy code base. Make changes as shown:

Clean shut down

finally, with most embeded robot controllers, its good practice to put

all actualtors into a safe position and then release all control objects and shut down all

communication paths. This keeps systems form jamming when you want to run again.

```
% Stop program and clean up the connection to Arduino
% when no longer needed
  writePosition(driveServo, 0.5);      % always end servo at 0.5
  writePosition(steerServo, 0.5);      % always end servo at 0.5
  writePosition(spinServo, 0.5);       % always end servo at 0.5
  clc
  disp('Rover control program has ended');
  clear robotArduino

  wrapUP = input('Type 1 to delete figures and exit');
  close(testTrack);           % clear figure
  close(roverSharpIRScan);    % clear figure
  close(localSharpIRPlot);    % clear figure
  close(brainWaves);          % clear figure

  beep                % play sytem sound to let user know program is ended
  disp('All done!');
```

Followed the new brain wave figure, by the operator trigger code to start the robot:

Ready to run robot, ask operator if they are set to run rover and suggest they move plots to a visable position

```
runRover = input('Ready to run Rover? Arrange your plots for visability, then type 1 and hit Enter ');
clc                                          % clean up cmmand window
```

That was the easy part, please run this modified code, debug as needed and then dive into the next significant rewrite.

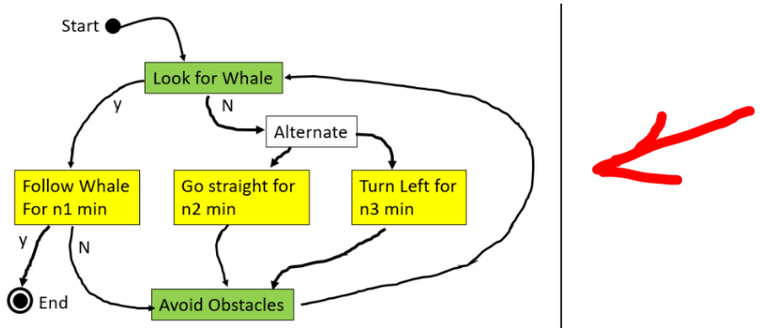
ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Now for the heavy robot code lifting, please modify the control loop (code that runs over and over) to have the following header:

Run robot control loop (code that runs over and over)

This control loop is an augmented Finite State Machine that cycles Robot Tug through a discret set of behaviors

as described by the follwing behavior diagram:



The N behaviors actuator-commands are merged in a polling type arbiter to achieve robust motion

in the presence of unplanned real world obstacles.

```
% create a set of control loop variables
controlIndex = 1;                           % create a robot loop control
behavoirCycleTime = 0;                      % create a resetable behavior clock
```

One of the really nice things about **Matlab live scripts** documentation capability is that you can put graphics in the comments. Having your FSM diagram as part of the code is just an awesome help when you are debugging late at night. It can't get separated from the code, can't get lost and every subteam writing functions can use it as a guide to be taking the same mountain. Feel free to cut and paste the tutorial picture for now, to be replaced with your FSM diagram after you create it.

Next, expand out the set of working variables for the control loop as shown and drop a section break after them to help you with future debugging if needed.
Add variables as shown:

```
% create a set of control loop variables
controlIndex = 1;                           % create a robot loop control
behavoirCycleTime = 0;                      % create a resetable behavior clock
followClock = 0;                            % create a cone follwing clock
vStraight = [0 0 0 0 0 0 0];                % create straight behavior vel vector
wStraight = [0 0 0 0 0 0 0];                % create staight behavior w vector
vTurn = [0 0 0 0 0 0 0];                    % create Turn behavior vel vector
wTurn = [0 0 0 0 0 0 0];                    % create Turn behavior w vector
vAvoid = [0 0 0 0 0 0 0];                   % create straight behavior vel vector
wAvoid = [0 0 0 0 0 0 0];                   % create staight behavior w vector
legTime = 20;                               % create a random legtime variable
vRef=0;                                     % initial Rover velocity
wRef=0;                                     % initial Rover rotation rate
```

Moving to the **while** loop itself, comment out the end waypoint break code, but leave it there (you might want it later). Then edit the SENSE section by updating the Sharp IR Range sensor **SENSE** function as shown and add a new **IsThereACone** function

```
% create FSM control loop
while (controlIndex < numel(t))                 % loop for number of elements in t
    writeDigitalPin(roverArduino,blinkLED,aliveLED);   % blink alive LED
    position = robotPoses(:,controlIndex)';     % rovers current position
    roverLocation = position(1:2);              % current rover X and Y from position

    % End loop if rover has reached goal waypoint within tolerance of 1.0m
    %    dist = norm(goalWayPoint'-roverLocation);
    %    if (dist < 1.0)                        % robot reaches end goal
    %        disp("Goal position reached")
    %        break;                             % stop control loop
    %    end

    % SENSE: collect data from robot lidar
    [ranges, angles] = SENSE(position, MapOfTrackInflated, sharpIRMap, ...
        roverSharpIRScan, localSharpIRPlot, ...
        sharpIRPod);
    [coneSeen, headingToCone] = IsThereACone();   % Function to use WebCam to find cone
```

The **IsThereACone** function will be just a placeholder now that always returns **no whale**. In the next section of this tutorial, you will get to use the Webcam on your

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Rover and a bit of startup code you can pull from this document to create your own working version of the cone finding function.Let's take a look at both SENSE functions in detail:

Sense Functions (store all Sense related local functions here)

```
function  [ranges, angles] = SENSE(position, mapOfTrack, blankSharpMap, fig_SharpMap, fig_localSharpPlot, lidar)
% SENSE scans the reference map using the range sensor and the current pose.
% This simulates normal range readings for driving in an unknown environment.
% Update the sharp map with the range readings.
% inputs are  tug position, mapOfTrack, sharpMap, figure to plot global sharp IR data,
% figure to plot local sharp IR, sensor name
% outputs are sharp IR array ranges and angles in local coordinate system
% Betsy Yu 2021  Rev A

    [ranges, angles] = lidar(position, mapOfTrack);
    scan = lidarScan(ranges,angles);
    validScan = removeInvalidData(scan,'RangeLimits',[0,lidar.Range(2)]);
    insertRay(blankSharpMap,position,validScan,lidar.Range(2));

    figure(fig_SharpMap);
        hold on;
        show(blankSharpMap);
        grid on;
        grid minor;
        hold off;

    figure(fig_localSharpPlot);   % positive X is forward, positive Y left on graph
    plot(scan);
    axis([-15 15 -15 15])
    hold on;
    plot(0,0, 'r*');              % plot robot position
    plot(-1,-0, 'r^');             % plot robot position
    plot(-2,-0, 'r^');             % plot robot position
    hold off;

end
```

It is largely unchanged, Go code reuse!

And the new **IsThereACone** one:

```
end

function [coneSeen, headingToCone] = IsThereACone()
% IsThereACone is a placeholder function for Think lab student generated
% WebCam generated code to find and track an orange safety cone
% Your Name here Spring 23
    coneSeen = 0;          % not seen = 0, seen = 1)
    headingToCone = 0;     % replace with code that takes an image, processes it,
                           % coverts the x coordinate of cone blob
                           % to a rover steering heading

end
```

It's good coding practice to create placeholder functions like this, while you generate your original code. They help hold the main flow of the control structure in place. And they can then easily be filled out by a sub-team without struggling to not break the main structure of the program.

Finally, getting to the very brain of the robot, please modify the THINK section as shown below, we will go over piece by piece in detail to follow.

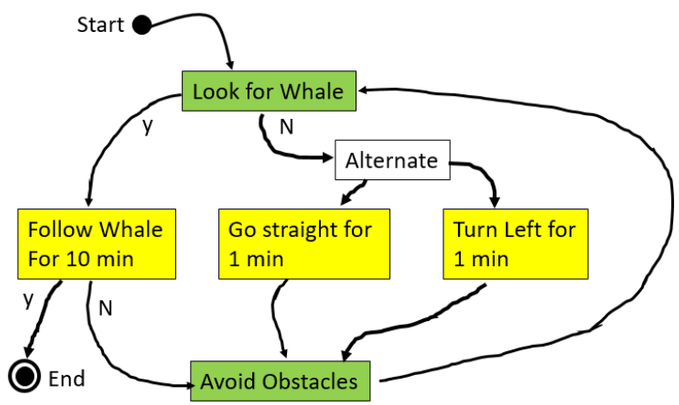
```
% THINK: compute what robot should do next-----------------------------
  % First create behaviors that alternate running based on if cone is
  % seen or not; namely randomly run THINKGoStraight and ThinkTurnLeft
  % to explore the track till a cone is seen and then switch to THINKFollowCone
  % if it is. Each behavior sets a vector of rover speed and angular
  % velocity

  switch coneSeen   % FSM switch between behaviors based on if cone is seen or not

      case 0        % cone NOT SEEN, random search behavior *********************************
        behavoirCycleTime = behavoirCycleTime+1;  % increment behavoir alternating clock
        if (behavoirCycleTime > legTime +5)       % check if behavior has run too long
            behavoirCycleTime = 0;                % reset alternating behavior timer
            legTime = randi([20 80],1);           % choose a random transit leg time
        end

        if (behavoirCycleTime < legTime)    % run GoStraight Behavior for legtime cycles
           [roverX,roverY,vStraight,wStraight,robotPoses] = THINKGoStraight(robotPoses,sampleTime,...
                                                            diffDriveRover,controlIndex,vRef,wRef);
           vTurn = [0 0 0 0 0 0 0];         % during straight behavior surpress all turn vel commands
           wTurn = [0 0 0 0 0 0 0];         % during straight behavior surpress all turn angvel commands
                                            % vStaight,wStraight,vTurn,WTurn go to arbiter

        else                                % run TurnLeft Behavior for legtime+5 cycles
           [roverX,roverY,vTurn,wTurn,robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveRover,...
                                                    controlIndex,vRef,wRef);
           vStraight = [0 0 0 0 0 0 0];     % during turn behavior surpress all straight vel commands
           wStraight = [0 0 0 0 0 0 0];     % during turn behavior surpress all straight angvel commands
                                            % vStaight,wStraight,vTurn,WTurn go to arbiter
        end

      case 1   % cone IS SEEN, head for cone behavior ***************************************
        if (followClock < 20)
            followClock = followClock+1;    % increment whale follwing time
            [roverX,roverY,vFollow,wFollow,robotPoses] = THINKFollowCone(robotPoses,sampleTime, ...
                diffDriveRover, controlIndex,vRef,wRef);
        else
            break;                          % follwed cone long enough, stop loop, mission complete!
        end

      otherwise                             % deal with case numbers not listed above
          disp("illegal behavior called")
          break;                            % stop loop on switch error
  end
```

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Looking at the overall structure of the control flow first, there is a **switch** statement that switches between searching behaviors if you haven't seen a cone **case 0** and a following behavior **case 1** if you have. Both cases are followed by an obstacle-avoid behavior and then all the behaviors feed into the arbiter. The Arbiter blends their individual desired velocity commands all together and drops the resultant Rover linear **vRef** and rotary velocity **wRef** into the following ACT function to carry out.

```
    otherwise                          % deal with case numbers not listed above
        disp("illegal behavior called")
        break;                         % stop loop on switch error
end

% Run Obstacle Avoidance behavior based on Sharp IR range data
[vAvoid,wAvoid] = THINKAvoid (ranges, angles);

% Arbitrate behavior outputs
[vRef,wRef]=THINKRoverArbiter(vStraight,wStraight,vTurn,wTurn,vAvoid,wAvoid,brainWaves);

% Check to see if Rover is hitting a physical obstacle
crash = checkOccupancy(MapOfTrackInflated,[roverX roverY]);
  if (crash == 1)
     disp('Rover crashed, all stop!');
     break;
  end
```

Jumping back, this is an explicit Matlab code structure that copies the original FSM diagram, show below:



The yellow states are selected by the **switch** structure plus a little logic code, all the states are followed by the Avoid Obstacle state, which cycles back to the look for cone (whale) state over and over.

Let's look at the **switch** code in more detail:

```
switch coneSeen   % FSM switch between behaviors based on if cone is seen or not

    case 0        % cone NOT SEEN, random search behavior *********************************
      behavoirCycleTime = behavoirCycleTime+1;  % increment behavoir alternating clock
      if (behavoirCycleTime > legTime +5)       % check if behavior has run too long
          behavoirCycleTime = 0;                % reset alternating behavior timer
          legTime = randi([20 80],1);           % choose a random transit leg time
      end

      if (behavoirCycleTime < legTime)    % run GoStraight Behavior for legtime cycles
         [roverX,roverY,vStraight,wStraight,robotPoses] = THINKGoStraight(robotPoses,sampleTime,...
                                                          diffDriveRover,controlIndex,vRef,wRef);
         vTurn = [0 0 0 0 0 0 0];         % during straight behavior surpress all turn vel commands
         wTurn = [0 0 0 0 0 0 0];         % during straight behavior surpress all turn angvel commands
                                          % vStaight,wStraight,vTurn,WTurn go to arbiter

      else                                % run TurnLeft Behavior for legtime+5 cycles
         [roverX,roverY,vTurn,wTurn,robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveRover,...
                                                    controlIndex,vRef,wRef);
         vStraight = [0 0 0 0 0 0 0];     % during turn behavior surpress all straight vel commands
         wStraight = [0 0 0 0 0 0 0];     % during turn behavior surpress all straight angvel commands
                                          % vStaight,wStraight,vTurn,WTurn go to arbiter
      end
```

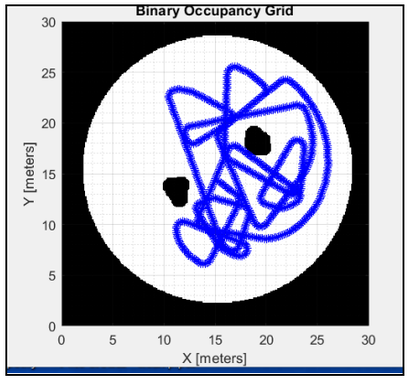
The **switch** statement is driven by **coneSeen**.

If none are, the following code will oscillate between two separate Rover behaviors, going straight and turning left.

The length of time spent in each behavior is controlled by a random number generator so that the Rover will get better coverage of the full track by doing long and short legs around it. If this leg time were fixed, Rover would limit the cycle into a fixed polygon.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Instead it will do a pretty decent random search of the full track::

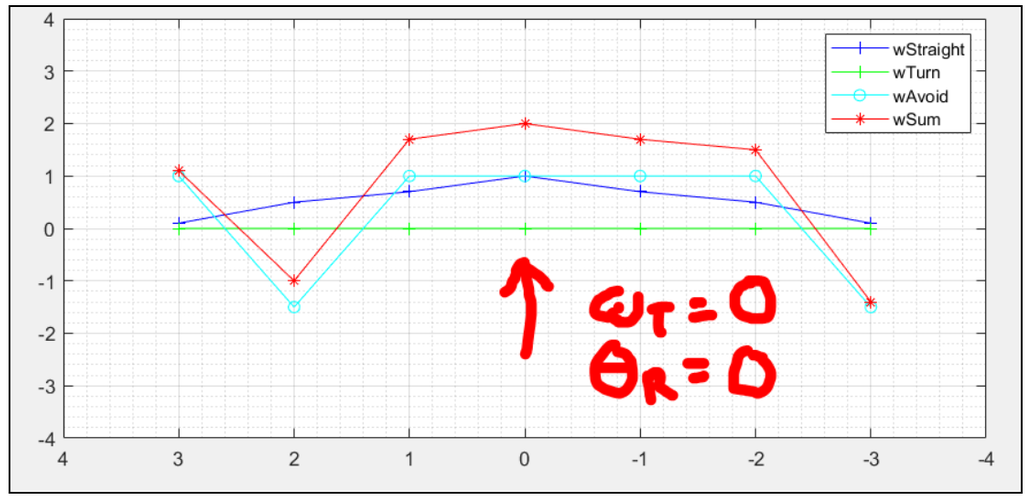


Let's look at the two **Think** behavior functions in more detail:

Think Functions (store all Think related local functions here)

```
function [roverX,roverY,vStraight,wStraight, robotPoses] = THINKGoStraight(robotPoses,sampleTime, ...
                                                    diffDriveRover, controlIndex, vRef, wRef)
% THINKGoStraight behavior locks steering at 0 center position and
% steers rover straight forward. Calculate derivative of robot motion
% based on control commands
% inputs are  rover poses, loop sampleTime, loop index
% outputs are roverX, roverY, robot poses (position of robot)
% Frank Olin 2021  Rev A

        % Replace pure pursuit controller with fixed velocities
        % [vRef,wRef] = ppControl(poses(:,loopIndex));
        vRover = vRef;                              % Rover forward velocity is 2 m/sec
        wRover= wRef;                               % Wheels centered, no rotational velocity

        % Perform forward discrete integration step
        vel = derivative(diffDriveRover, robotPoses(:,controlIndex), [vRover wRover]);
        robotPoses(:,controlIndex+1) = robotPoses(:,controlIndex) + vel*sampleTime;

        % Update rover location and pose
        roverX = robotPoses(1,controlIndex+1);
        roverY = robotPoses(2,controlIndex+1);
        vStraight = [0 0.5 1 2 1 0.5 0];            % create polled desired linear velocity
        wStraight = [0.1 0.5 0.7 1 0.7 0.5 0.1];    % create pooled desired angular velocity
end
```

The **ThinkGoStraight** behavior takes in the Rovers current state, figures out its new position and then outputs two vectors **vStraight** and **wStraight** which are seven element distributed desired goals, like [0 0.1 0.5 1 0.5 0.1 0] for the Rovers linear and rotational velocity. In many circles they are called **brain waves** because they come from a biological brain model and because they are easy to plot on a brainwave graph. These brain waves make it really easy to watch what each behavior would like to do in near real time, for example the **wStraight** blue trace in the brainwave graph below is a desired rudder at center, seven element, go straight behavior:



In many similar systems, the number of discrete elements in the loosely probabilistic vRef and wRef brain wave vectors is tied back to the resolution of your Rovers primary sensor array. Here we have 6 Sharp IR rangefinders, so the brain wave vectors for all behaviors are 7 elements long. If you had something like a Hokuyo Lidar on board with 270 elements, your brain wave could have much finer resolution and let your rover steer through a more complex environment..

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

Taking a look at the ThinkTurnLeft behavior, it operated much the same way as the straight behavior, but produces a brainwave for wRef shifted the the far left for a left hand turn:

```
function [roverX,roverY,vTurn,wTurn, robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveRover, ...
                                                controlIndex, vRef, wRef)
% THINKTurnLeft behavior locks steering wheels at a fixed angular position and
% steers Rover in a tight left turn. Calculate derivative of robot motion
% based on control commands.
% inputs are  rover poses, loop sampleTime, loop index
% outputs are roverX, roverY, robot poses (position of robot)
% Frank Olin 2021  Rev A

        % Replace pure pursuit controller with fixed velocities
        % [vRef,wRef] = ppControl(poses(:,loopIndex));
        vRover = vRef;    % Tug forward velocity is 2 m/sec
        wRover = wRef;    % Rudder set 2 rad/sec rotational velocity

        % Perform forward discrete integration step
        vel = derivative(diffDriveRover, robotPoses(:,controlIndex), [vRover wRover]);
        robotPoses(:,controlIndex+1) = robotPoses(:,controlIndex) + vel*sampleTime;

        % Update rover location and pose
        roverX = robotPoses(1,controlIndex+1);
        roverY = robotPoses(2,controlIndex+1);
        vTurn = [1.8 2 1.8 1.7 0 0 0];    % create tolled desired linear velocity
        wTurn = [1.1 1 0.9 0.8 0 0 0];     % create tooled desired angular velocity
end
```

Returning to the **case 0** statement, both straight and turn behaviors suppress the brain wave output of the other when they are active so that the arbiter is not mixing in incorrect signals. This is one of the main foundations of the subsumption architecture at work.

```
    if (behavoirCycleTime < legTime)    % run GoStraight Behavior for legtime cycles
        [roverX,roverY,vStraight,wStraight,robotPoses] = THINKGoStraight(robotPoses,sampleTime,...
                                                         diffDriveRover,controlIndex,vRef,wRef);
        vTurn = [0 0 0 0 0 0 0];         % during straight behavior surpress all turn vel commands
        wTurn = [0 0 0 0 0 0 0];         % during straight behavior surpress all turn angvel commands
                                         % vStaight,wStraight,vTurn,WTurn go to arbiter

    else                                 % run TurnLeft Behavior for legtime+5 cycles
        [roverX,roverY,vTurn,wTurn,robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveRover,...
                                                 controlIndex,vRef,wRef);
        vStraight = [0 0 0 0 0 0 0];     % during turn behavior surpress all straight vel commands
        wStraight = [0 0 0 0 0 0 0];     % during turn behavior surpress all straight angvel commands
                                         % vStaight,wStraight,vTurn,WTurn go to arbiter
    end
```

In **case 1** when the cone is seen, code is left to your team to design. It has just enough structure in place to hold your whale following behaviors and not more.

```
   case 1   % cone IS SEEN, head for cone behavior ***************************************
     if (followClock < 20)
         followClock = followClock+1;    % increment whale follwing time
         [roverX,roverY,vFollow,wFollow,robotPoses] = THINKFollowCone(robotPoses,sampleTime, ...
             diffDriveRover, controlIndex,vRef,wRef);
     else
         break;                          % follwed cone long enough, stop loop, mission complete!
     end

   otherwise                             % deal with case numbers not listed above
       disp("illegal behavior called")
       break;                            % stop loop on switch error
end
```

Your team can use the next section's computer vision code to fill out the placeholder function **ThinkFollowCone** left for you in this case.

Case 1 is followed by the **otherwise** case, that is there mainly to handle illegal case numbers some future programmer might try to use.

For reference the placeholder ThinkFollowCone function is shown below:

```
function [roverX,roverY,vFollow,wFollow,robotPoses] = THINKFollowCone(robotPoses,sampleTime, diffDriveRover,...
                                                    controlIndex,vRef,wRef)
% THINKFollwCone takes in the the state of the Rover and using webcam to
% find cone steers rover to follow it. Student teams write code and
% documentation
%
% Frank Olin 2021  Rev A

roverX = 20;
roverY = 20;
vFollow = [0 0 0 0 0 0 0];
wFollow = [0 0 0 0 0 0 0];

end
```

The final bit of THINK code contains three functions, two you write, one is part of the Matlab robotics toolbox:

```
    % Run Obstacle Avoidance behavior based on Sharp IR range data
    [vAvoid,wAvoid] = THINKAvoid (ranges, angles);

    % Arbitrate behavior outputs
    [vRef,wRef]=THINKRoverArbiter(vStraight,wStraight,vTurn,wTurn,vAvoid,wAvoid,brainWaves);

    % Check to see if Rover is hitting a physical obstacle
    crash = checkOccupancy(MapOfTrackInflated,[roverX roverY]);
      if (crash == 1)
         disp('Rover crashed, all stop!');
         break;
      end
```

The last one, **checkOccupancy** is the built in toolbox function and will tell you if any particular location in an occupancy grid is free or has an object in it. If we check the Rover's current position, with it, each time the loop runs, we can tell if the Rover has hit an obstacle. When it does, the **break** statement stops the loop and simulation terminates. This is the equivalent of crashing in real-life.

Let's look at the other two functions in some detail. The **THINKAvoid** function takes in the SharpIR array's range data and uses it to create an obstacle avoidance brainwave that will let the Rover go in any direction unhindered, except in the direction of a nearby obstacle. Please create the following code for it:

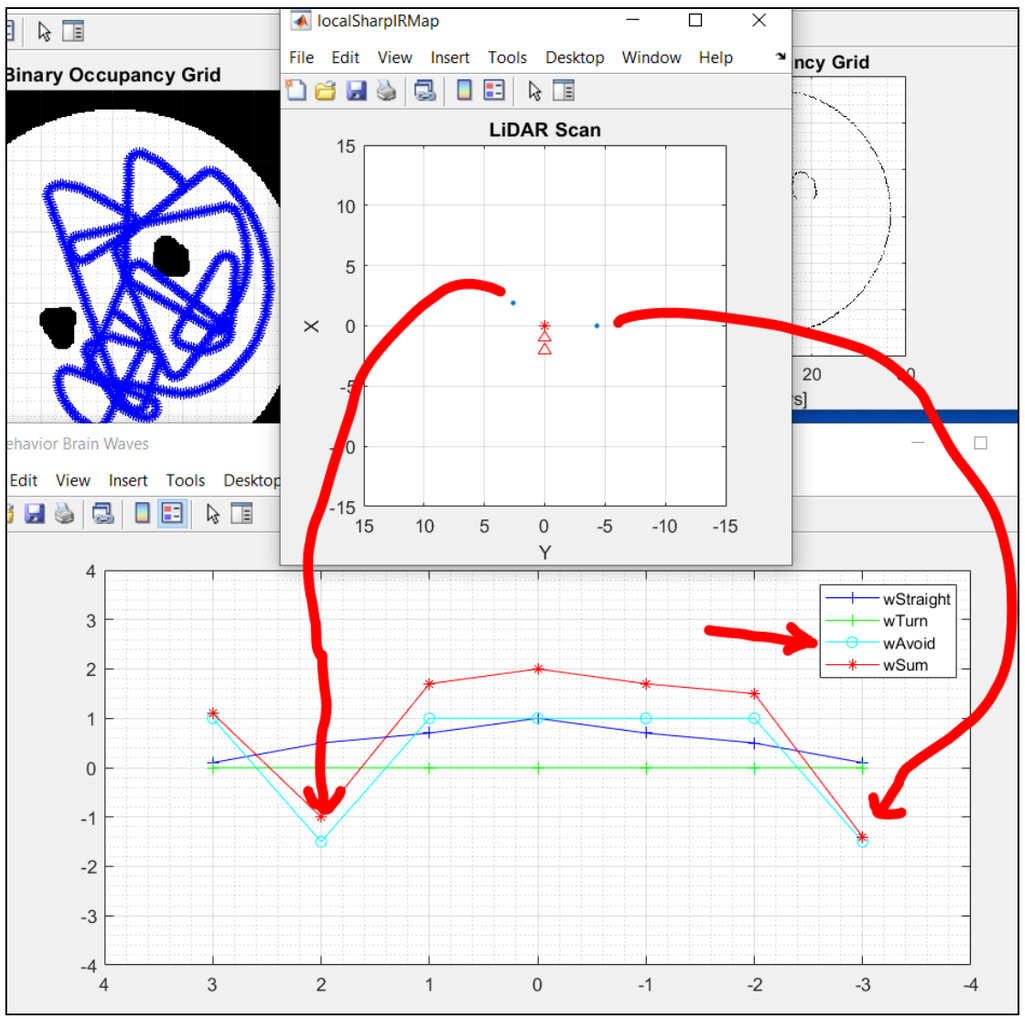
```
function [vAvoid,wAvoid] = THINKAvoid (ranges, angles)
% THINKAvoid behavior takes in the ranges from the rovers Sharp Ir array and
% converst them into a set of polled linear and rotational velocity
% designed to avoid crashing into percieved obstacles
% sharpIRPod.Range = [0.1,10];              sets minimum and maximum range of sensor
% sharpIRPod.HorizontalAngle =[-pi/2, pi/2];  sets max min detection angle
% sharpIRPod.HorizontalAngleResolution = 0.628318; sets sharp pod resoultion to 6 sensors
% Frank Olin 2021  Rev A

starboardSensors = ranges(1:3);                 % Sharp Irs pointing right
portSensors = ranges(4:6);                      % Sharp IRs pointing left
middleSensor = (ranges(3)+ranges(4))/2;         % make up a virtual middle sensor
s1 = vertcat(starboardSensors,middleSensor);    % add virtual middle to left array
sensorArray = vertcat(s1,portSensors);          % add right array
TF = isnan(sensorArray);                        % find any Nan values
sensorArray(TF)= 30;                            % replace NaN values
    for iScanTransform =1:7
        if(sensorArray(iScanTransform) < 7)     % if range is shorter than N vote to
        sensorArray(iScanTransform)= -1.5;      % not turn in this sensor direction
        else
        sensorArray(iScanTransform)= 1;         % else can turn this way
        end
    end
wAvoid = sensorArray';                  % flip vertical ranges to horizontal wAvoid
wAvoid = flip(wAvoid);                  % flip array to match top down boat coordinate system
vAvoid =[0 0 0 0 0 0 0] .* angles;       % set better slow down velocity policy in your student arbiter
end
```

The first part of the code pads the 6 sharp IRs out to 7 by creating a virtual center IR that averages the ranges of the right and left front one. This gives the array and then the brainwave it generates symmetry around the Rovers main axis. Straight ahead is then zero wRef.

The next bit of code finds any NaNs (not a number) in the sensor data and replaces it with an arbitrarily long value of 30. The arbiter can't deal with Nans when adding up waves.The **if** statement loads the wRef brainwave with a constant -1.5 if an obstacle is seen closer than 7 meters from the Rover and +1 elsewise. This is a pretty basic Obstacle Avoid system, go or no go, you could write one based on actual linear sensor range that would be much more elegant (and more robust).

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

The final part flips the command array so that it will match a top down view of the Rover to make it easier for human operators to figure out what is going on. Here is an example where you can see the **Obstacle Avoid Behavior** (in cyan) is voting against wRef turn rates that would head into the two obstacles it can see, while allowing all other headings:



Finally all the brain wave vectors from all running behaviors go to the very heart of your Rover's brain, the **Think** section behavior arbiter. Please create the code below.

```
function [vRef,wRef]=THINKRoverArbiter(vStraight,wStraight,vTurn,wTurn,vAvoid,wAvoid, brainWaves)
% THINKRoverArbiter takes in the desired outputs of the Robot rovers
% many behaviors, merges just the headings into a single rudder position.
% Speed arbitation can be added by student teams for better performance
% Frank Olin 2021  Rev A

    turnSpeeds = [ 3 2 1 0 -1 -2 -3];        % create a vector of possible turn rates

    figure(brainWaves)                       % go to brain waves figure
    %clf                                     % clear figure of old waves
    plot( turnSpeeds, wStraight,"b-+");      % plot staight waveform
     ax=gca;
     ax.XDir = 'reverse';
    axis([-4 4 -4 4]);
    grid on;
    grid minor;
    hold on
    plot( turnSpeeds, wTurn,"g-+");              % plot turn waveform
    plot( turnSpeeds, wAvoid,"c-o");             % plot avoid waveform

    %vSumOfWaves = vStraight + vTurn + vAvoid;     % sum velocity waveforms
    wSumOfWaves = wStraight + wTurn + wAvoid;     % sum rotational velocity waveforms
    plot(turnSpeeds, wSumOfWaves,"r-*");          % plot sum of rot waveform
    legend('wStraight','wTurn','wAvoid','wSum')   % label plot
    hold off

    %[vMaxValue, vMaxIndex] = max(vSumOfWaves);    % Find peak of summed linear velocity waveform
    [wMaxValue, wMaxIndex] = max(wSumOfWaves);    % Find peak of summed angular velocitywaveform

    % vRef= vSumOfWaves(vMaxIndex);                 % could use this value to build a better arbiter
    vRef=2;                                       % set a fixed 2m/s forward speed
    wRef = turnSpeeds(wMaxIndex);                 % set tug turn rate based on peak of behavior voating curve
end
```

This is a very simple summation-maximum arbiter. It is a simple example that only sets the **wRef** turn rate and holds the **vRef** linear speed of Rover constant. It adds up all the incoming brain wRef wave vectors, then finds the maximum value of the resulting **wSumofWaves** vector and then maps that maximum's index into a lookup table of possible turn rates stored in **turnSpeeds.** You can see the arbiter at work plotting out all the incoming brainwaves and then the **wSum** wave in the plot on the previous page. In this case, the maximum is in the center because both the go straight and obstacle avoid behaviors agree most highly there (add up to highest value).

As part of your follow on project work, you can design a much better version of this, perhaps arbitrating speed as well. To assist in that good work, many additional values (currently unused) are included as function inputs. Moving on to the ACT section, it's mostly unchanged

```
    % ACT: command robot actuators --------------------------------
    ACT(roverX, roverY, vRef, wRef, driveServo, steerServo, spinServo, testTrack);

    controlIndex = controlIndex + 1;              % increment control loop index
    waitfor(r);                                   % wait for loop cycle to complete
    aliveLED = ~aliveLED;                         % toggle blinky alive LED

end
beep                              % give audio indication mission end is reached
```

Using the same original code **ACT** function:

Act Functions (store all Act related local functions here)

```
function ACT(roverX, roverY, vRef, wRef, driveServo, steerServo, spinServo, fig_testTrack)
% ACT Increment the robot pose based on the derivative and drives Rover
% servos appropriately to simulate actual motion on a virtual test track.
% inputs are curretn X and Y position of rover and figure to plot rover in.
%
% outputs are no  outputs
% D Barrett 2023 Rev A

        % Plot Rover location on track
        figure(fig_testTrack)
        hold on
        plot(roverX,roverY, 'b*');              % plot robot position on track
        hold off

        % Drive Rover hardware servo motors
        steerScale=[-3 wRef 3];             % build sim to real world scaling vector wRef -3 to 3 rad/sec
        driveScale=[0 vRef 2];              % build sim to real world scaling vector vRef 0 to 2 m/sec
        steerCom = rescale(steerScale);     % convert angular rate wRef to steer servo command from 0->1
        driveCom = rescale(driveScale);     % converts vRef to drive servo command from 0->1
        if (vRef > 0.5)
         turnTableCom= rescale(steerScale);  % converts steer angle into turntable servo command 0->1
        else
          turnTableCom = 0.5;               % stop turntable servo if drive servo is off
        end

        % scale simcomand based on real world limits [low speed,comspeed, highspeed]
        % servo command in center of xCom vector [ 0 , comRate, 1 ]
        writePosition(steerServo, steerCom(2));          % command steer servo to run 0-1
        writePosition(driveServo, driveCom(2));          % command drive servo to run 0-1
        writePosition(spinServo, turnTableCom(2));       % command turntable servo to run 0-1

end
```

This new ACT function draws the Rover on the test track as well as it drives your Rovers steering wheels and drive wheels in real-life in a hardware-in-the loop simulation of the Rover actually motoring around the track.

This is a THINK lab, so we want you to focus on that part of your Rover's brain and not the actuators and so supply this function. But don't fret, you either learned about RC servo control in the previous lab or you will in the following one.

Wrapping up this code, please edit the last shut down section as shown:

Clean shut down

finally, with most embeded robot controllers, its good practice to put

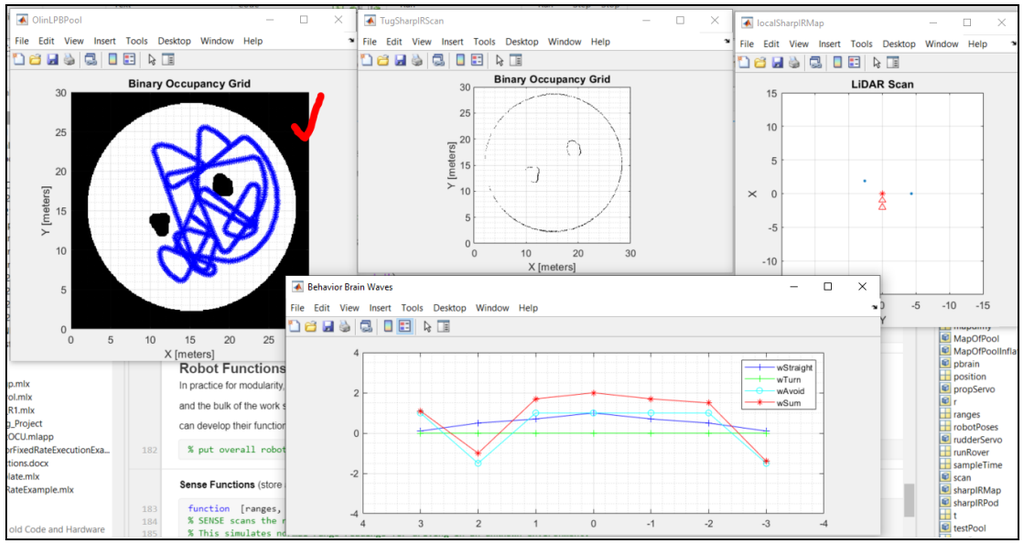
all actualtors into a safe position and then release all control objects and shut down all

communication paths. This keeps systems form jamming when you want to run again.

```
% Stop program and clean up the connection to Arduino
% when no longer needed
  writePosition(driveServo, 0.5);      % always end servo at 0.5
  writePosition(steerServo, 0.5);      % always end servo at 0.5
  writePosition(spinServo, 0.5);       % always end servo at 0.5
  clc
  disp('Rover control program has ended');
  clear robotArduino

  wrapUP = input('Type 1 to delete figures and exit');
  close(testTrack);          % clear figure
  close(roverSharpIRScan);   % clear figure
  close(localSharpIRPlot);   % clear figure
  close(brainWaves);         % clear figure

  beep                % play sytem sound to let user know program is ended
  disp('All done!');
```

When you run your code, you should get a fully autonomous Rover steaming around the track in a quasi random pattern, cleanly avoiding hitting walls and obstacles. Please note the robust obstacle avoidance compared with your previous control system program 4a crashing straight into one. **This is a graphic depiction of the difference between a robot and a control system.**

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d



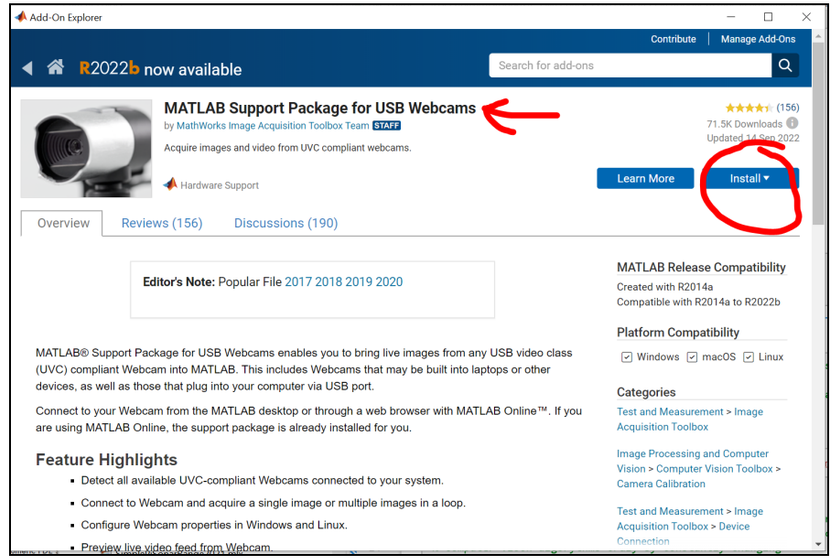
Brief walkthrough of the orange cone tracking code

Before moving on to your final task, writing your own Rover THINK code behaviors, we are going to provide you with a working cone Tracking Function for the Rovers Web-Cam. You will either have worked previously with this camera in the SENSE lab or will work with it after this one.

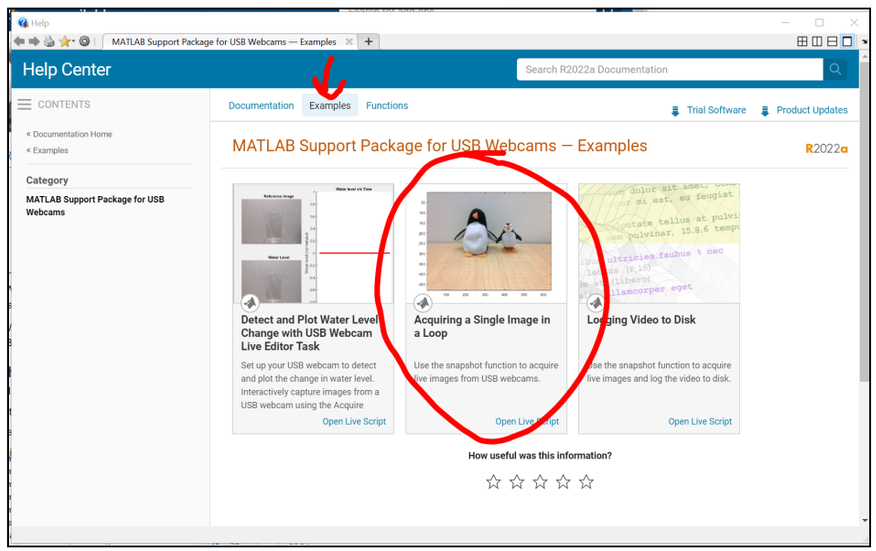
Now that we have the Web-Cam connected, let's walk through how to use it with Matlab. In this example, we will use Matlab's colorThresholder function to specify a target to track based on color, which will work well for cone-tracking.

In order for this code to work, you need to have the USB Webcam support package loaded into your Matlab.

First go to the Add-On Explorer, search for USB WebCam and install it:



Although you will cover this material in depth in the sense lab, it would be good to quickly run through the **Acquiring a Single Image In a Loop** example:



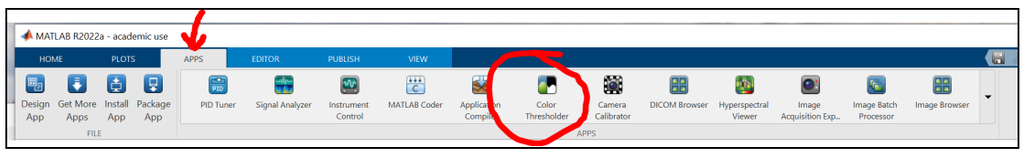
Next, you will need to create a function to set up your Rover's USB camera within the MATLAB environment. Please enter the one below. Don't stress the details, we will cover those in the Sense Lab. This is old school straight MATLAB function code, and not a fancy new livescript, but enter it as is and it should work fine:

```
function [robotCam, imgForFilter] = SetupUSBCam()
% This function create and configures a webcam to be a simple vision
% system. It takes a standard webcam attached to your computer and
% optimizes its parameters for computer vision work.
% It returns a Matlab webcam object; robotCam and leave a captured image in
% the workspace for future filter work.
% Microsoft webcam has the tunable properties:
%{
                    Name: 'Microsoft® LifeCam Cinema(TM)'
      AvailableResolutions: {1×11 cell}
                Resolution: '640x480'
                Saturation: 83
     BacklightCompensation: 0
                 FocusMode: 'auto'
              ExposureMode: 'manual'
          WhiteBalanceMode: 'manual'
                       Pan: 0
                  Exposure: -8
                  Contrast: 5
                      Zoom: 0
             WhiteBalance: 4500
                     Focus: 1
                      Tilt: 0
                Brightness: 100
                 Sharpness: 25
%}

% Identify available webcams
webcamlist;
robotCam = webcam(1);

% Preview Video Stream
preview(robotCam)

% wave hand in front of camera to see raw image
input('move hand in front of camera, then type g, then hit Enter ', 's');
clc;
closePreview(robotCam);

% Fix auto exposure light variance by setting it to manual
robotCam.ExposureMode = 'manual';

% Set whitebalance light vaiance to manual too
robotCam.WhiteBalanceMode = 'manual';

% Enter best experimentally determined values here
robotCam.Exposure = -8;
robotCam.Brightness = 100;

% Preview Video Stream with new hand=set parameters
preview(robotCam);

% wave hand in front of camera
input('move hand in front of camera, then type g, then hit Enter ', 's');
closePreview(robotCam);

% save an image of target to create color filter with
imgForFilter = snapshot(robotCam);  % capture an image from USB camera
save('testTargetImage.jpg',"imgForFilter");
clc;

end
```

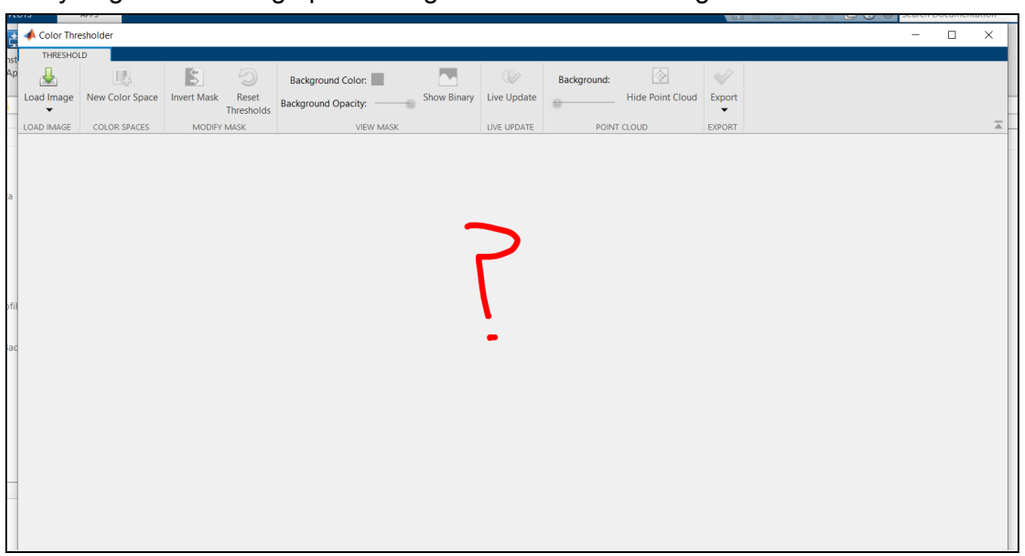
Please run the function from the command line, it will first bring a live feed from the raw USB camera, then set some of its parameters to facilitate color object tracking and finally drop an image into both a file and the MATLAB workspace to help you create a good color filter for cone tracking.

Next, we are going to use one of MATLABS built in computer vision Apps to create a color filter that will segment out and find just objects of the particular color that you choose, in this case the orange cone. It will generate a really cool function that will take your input image and reliably find just those parts of it that have the color you specified.

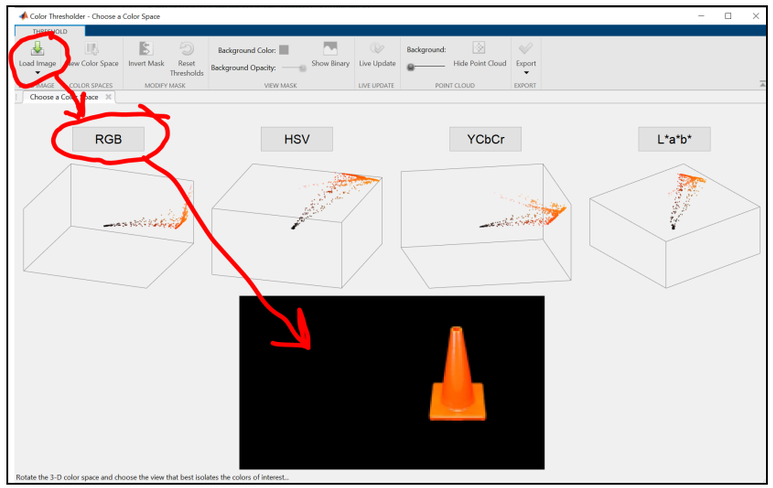
Go to **APPS tab** then **ColorThresholder**



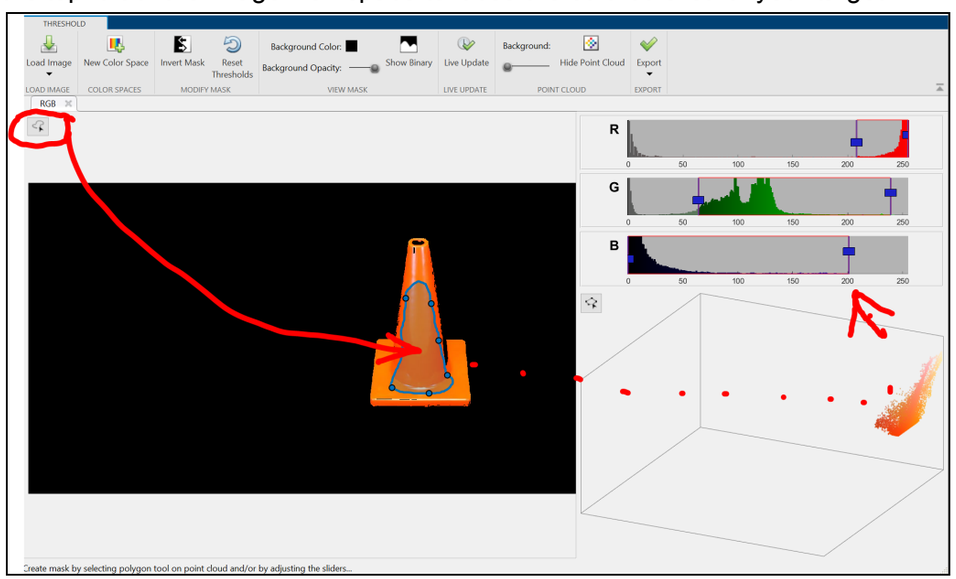
There are a lot of APPS in 2022b, you may need to scan down the full page of APPS until you get to the image processing ones. Click on it and get:



Click on **Load Image** and load one of your saved image of cone from the MATLAB workspace, it will appear in the center work space:

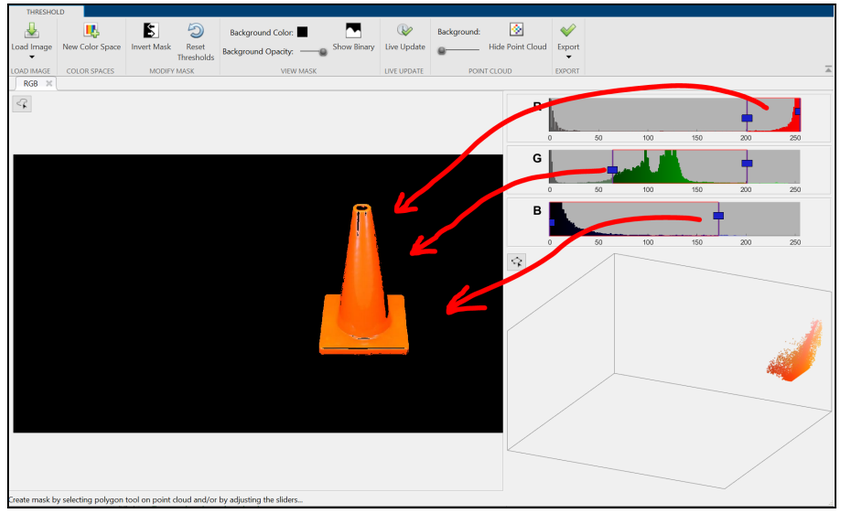


Select RGB color space (we will start out with RGB, but you might find other color spaces let you better distinguish between similar colors) and then use the **lasso tool** in the top left of the image workspace to lasso the colored area of your target:

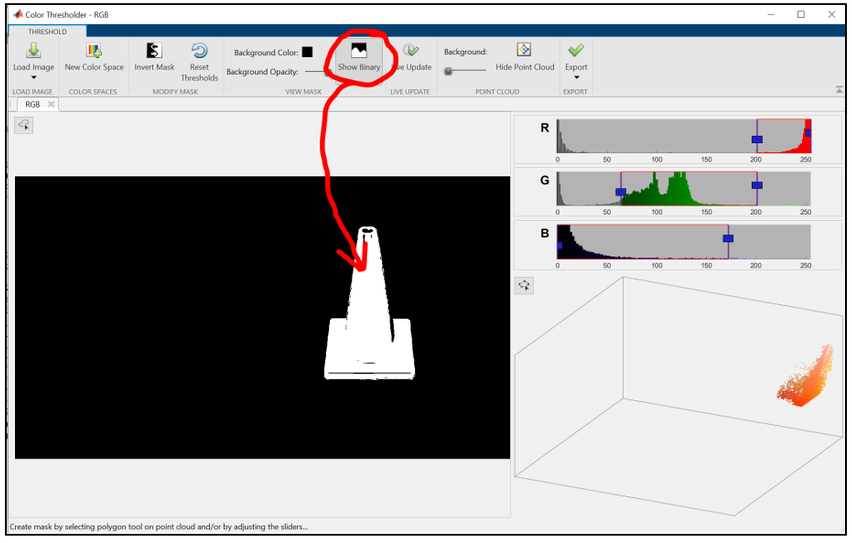


ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2023d

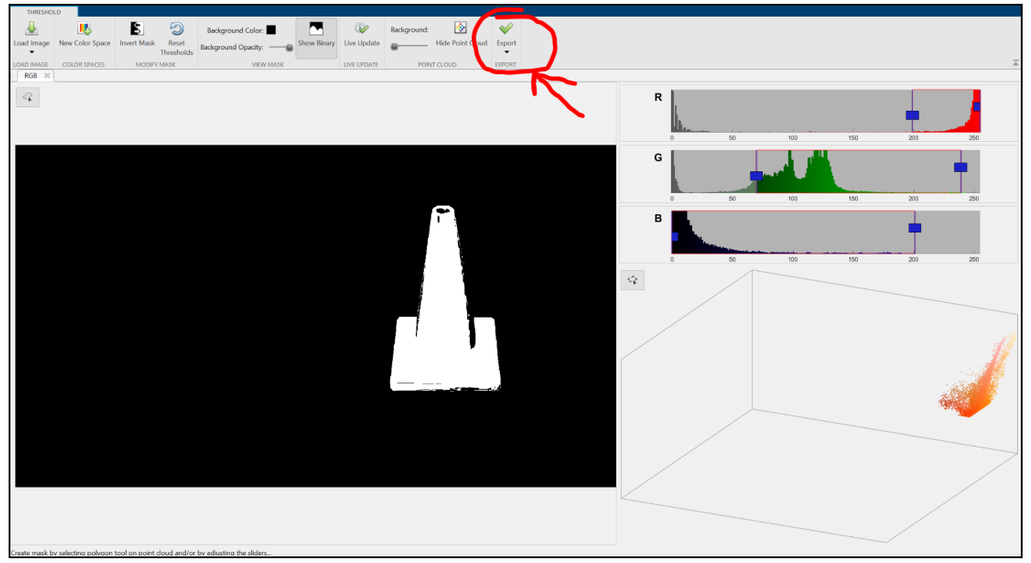
Use the two bar sliders on the R G B tracks to fine tune the filter to capture the best segmentation of the image that you can.



You can click on **Show Binary** button to see the full binary mask that you just created with this App tool:



Click on **Export** choose **Export Function**



and the App will drop a new masking function, fully documented into your Editor window. It just did a whole lot of image processing in a few mouse clicks

```
function [BW,maskedRGBImage] = createMask(RGB)
%  createMask  Threshold RGB image using auto-generated code from colorThre
%  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
%  auto-generated code from the colorThresholder app. The colorspace and
%  range for each channel of the colorspace were set within the app. The
%  segmentation mask is returned in BW, and a composite of the mask and
%  original RGB images is returned in maskedRGBImage.

% Auto-generated by colorThresholder app on 06-Feb-2023
%------------------------------------------------------


% Convert RGB image to chosen color space
I = RGB;

% Define thresholds for channel 1 based on histogram settings
channel1Min = 201.000;
channel1Max = 255.000;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 64.000;
channel2Max = 201.000;
```

Edit with updated comments and save it into your working Matlab Drive directory with a useful name like createConeMask.m:

```
function [BW,maskedRGBImage] = createConeMask(RGB)
%  createMask  Threshold RGB image using auto-generated code from
%  colorThresholder app. It finds orenge cones against a dark background
%  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
%  auto-generated code from the colorThresholder app. The colorspace and
%  range for each channel of the colorspace were set within the app. The
%  segmentation mask is returned in BW, and a composite of the mask and
%  original RGB images is returned in maskedRGBImage.

% Auto-generated by colorThresholder app on 06-Feb-2023
%------------------------------------------------------


% Convert RGB image to chosen color space
I = RGB;

% Define thresholds for channel 1 based on histogram settings
channel1Min = 201.000;
channel1Max = 255.000;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 64.000;
channel2Max = 201.000;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 0.000;
channel3Max = 172.000;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% Initialize output masked image based on input image.
maskedRGBImage = RGB;

% Set background pixels where BW is false to zero.
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;

end
```

Next, you will enter a starter function for **IsThereACone.m.** Enter text as show:

```
function  [coneSeen, headingToCone] = IsThereACone(robotCam)
% IsThereACone is a placeholder function for Think lab student generated
% WebCam generated code to find and track an orange safety cone
% robotCam must be a MAtlab USB camera object, please create it with
% SetupUSBCam.m
%
% Your Name here Spring 23

img = snapshot(robotCam);  % capture an image form USB camera
imshow(img);               % display it
%imtool(img);               % explore image with tool


%% Use Color Threshold App to create a colorMask function (coneMask)
%  and place function in same directory as this script. Your functions
%  will have different names depending on color of target you want to
%  track, apply colorMask Function to captured image
%  this will create two new images
% [BW,maskedRGBImage] = createConeMask(RGB)
%  BW is binary mask of image
%  maskedRGBImage will be the color image inside the mask

   [BW,maskedRGBImage]=createConeMask(img);
   imshowpair(BW,maskedRGBImage,"montage");
   input('check out masked image, type G, then hit ENTER ','s');
```

At a high level, the steps that this function walks through are:

1. Takes a image
2. Applies ConeMask Filter to it.

Then we will do a little image processing on the masked filtered image. Don't get bogged down in these steps, we will cover in greater detail in the Sense lab web camera station.

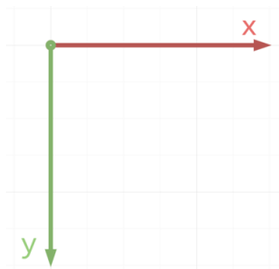
Please add the following code to the tail of this function:

```
%% Preprocess image to remove noise

   se = strel('disk',7);              % structured element erosion functi
   cleanImage= imopen(BW, se);         % Morphologically open image
   imshowpair(BW,cleanImage,"montage");
   input('check out cleaned image, type H, then hit ENTER ','s');
   clc;

%% Calculate the centoid of the white part of masked area (target)
   targetCenter = regionprops(cleanImage,'centroid');
   % Store the  x and y coorinates in a two column matrix
   centroids = cat(1,targetCenter.Centroid);
   % Display the original image with the centroid locations superimposed

   imshow(img)
   hold on
   plot(centroids(:,1),centroids(:,2),'b*')
   text(centroids(:,1),centroids(:,2), '  Target Centroid');
   hold off
   input('check out target center, type j, then hit ENTER ','s');
   clc;

%%  Calculate the area of the target
   targetArea = regionprops(cleanImage,'area');
   area=targetArea.Area;
   % Store the  x and y coorinates in a two column matrix
   centroids = cat(1,targetCenter.Centroid);
   % Display the binary image with the centroid locations superimposed.

   imshow(img)
   hold on
   plot(centroids(:,1),centroids(:,2),'b*')
   text(centroids(:,1),(centroids(:,2)+ 10),num2str(area));
   hold off
   input('check out target area, type k, then hit ENTER ','s');
%% convert the x coordinate of centroid to a heading
%  and determine if a cone is seen
   if ( area < 10)
      coneSeen = 0;        % cone not seen = 0, seen = 1)
   else
      coneSeen = 1;        % conot seen = 0, seen = 1)
   end

   % student must map position of cone centroid to a rover heading
   headingToCone = centroids(:,1);    % 0-640 pixels left to right
   clc

end
```

3. The preprocess section clears up image
4. The centroid section finds the center of the cone
5. The area section finds the area
6. The final section returns whether the camera detects a cone and a numerical value to its heading.

Now that we have this function, we can use it for tracking!

Aside: When using this function, it's important to note the coordinate system of the targetX and targetY values. In Matlab and most other image-processing software, the origin is located at the upper left of the image, with the x-axis pointing right and the y-axis pointing down. Keep this in mind if you plan on using this function to control your Rover!



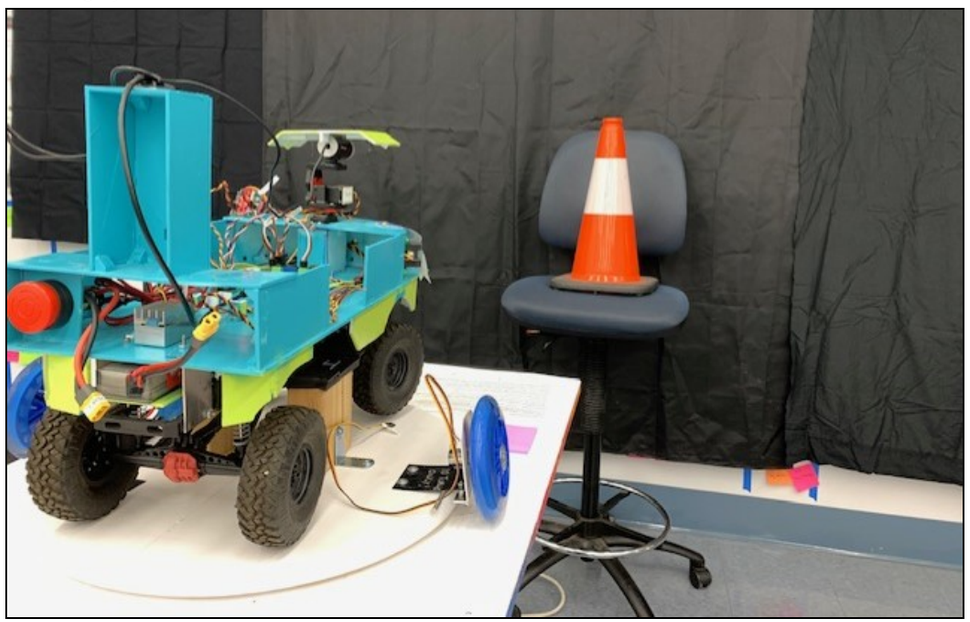
Matlab image coordinate system

From here on, you are on your own to create a cone tracking behavior. We will recommend you comment out all image showing and operator input parts of the **isThereACone** function for speed, once you get it all working.

Developing your Cone Tracking Rover Demo

From the previous program, you can see that just three simple behaviors; go-straight, turn and avoid, let the Rover robustly and in the presence of dynamic obstacles, explore the virtual track looking for a cone. Your team's mission is to use that code provided as a starting template. Using the given Matlab functions for both SENSE, THINK and ACT, controlling the Rovers actual motors and turntable your team should:

1) first create a new FSM behavior diagram to clearly describe your Rover's behaviors. It's ok to go a bit wild here. Be daring! Maybe you should spiral out from the center of the track? Maybe you want to precode in an optimum search path and then write a pure pursuit **GoToWaypoint** behavior to follow it. While you are searching, you will need to constantly both avoid track walls and obstacles,



as well as be on the lookout for the orange cone !

If and when you find the cone(being waved about by a course Ninja), you need to track it for at least 10 seconds to have a successful video capture mission. You have carte-blanche to redesign sensors, behaviors, and your arbiter to improve performance (and probably should improve all given functions too).

2) having created a clean FSM, show it to course Ninjas and Instructors for both scope feedback and ideas, before diving in to code it. Remember we are trying to bound this to a two week experience.

3) divide up the work evenly into 4 pair-programming sub-teams. Each team should have at least one behavior function and preferably two to work on. Reworking the general code structure to improve it would count toward that goal. Please resist the urge to rewrite the underlying structure from scratch for all the reasons given previously in this course. And always remember, the goal is simple and elegant. Not complex!

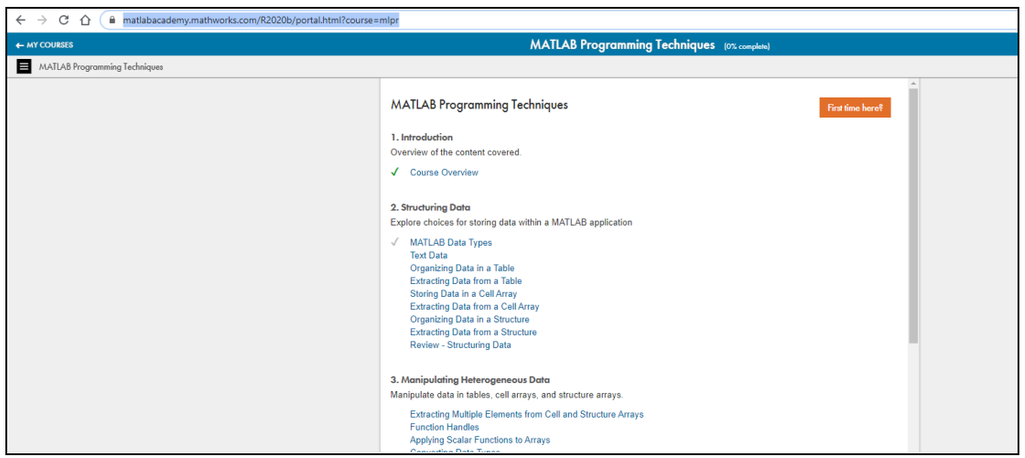
Two week demo: Please write this code, debug it, and be prepared to demo both your control system and a full cone following Robot Rover.

As always, please see an instructor or Ninjas for help with any part of the above.

MATLAB Skill Building Tutorials

While you are working on your lab hardware, we would like you to continue developing your MATLAB skills in anticipation of needing them a little during the hardware labs and a lot for the final project. The On-Ramp was a nice introduction, but you can only learn depth in core technical skills by use and repetition. You can get this depth in MATLAB for this course by working through their more advanced **MATLAB Programing Techniques** on-line tutorial

https://matlabacademy.mathworks.com/R2020b/portal.html?course=mlpr



Working in 2 two week long segments (while doing the 3, 2 week long hardware labs) we will ask you to read through and do the short tutorial examples in the following order. Regardless of which lab you start with, it would be best for you to proceed through the tutorials segments in the order shown. If you already are pretty proficient in MATLAB via other Olin courses, you can just quickly scan this material for a fast refresher. There are no hard goals here, we are just trying to get each robot lab team up to a reasonable common level of MATLAB coding skill at a reasonable pace in preparation for the big final project after the labs. If you are a novice MATLAB programmer, it is recommended that you work through the material fairly consistently, but feel free to skip over any parts you don't feel are relevant, you can always return to them later. If you are already highly skilled in MATLAB, you can just use it to brush up on the finer points of things like data structures. There is no formal deliverable for this work, beyond uploading a screen capture of your progress with each lab, but you will both learn a lot more and get a lot more out of this course if you aren't constantly asking your fellow teammates or the Ninjas how to **Plot** or how to write a simple **Switch** structure.

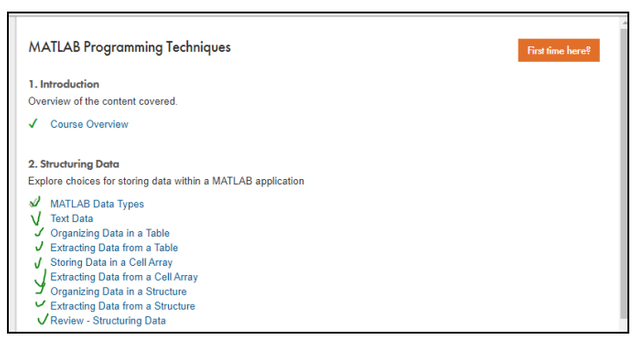
Please see a course instructor or Ninjas for MATLAB help as needed.

Lab Week 1-2:

Please work through:

1. Introduction
2. Structuring Data
3. Manipulating Heterogeneous Data

Grab a screen capture of your progress and upload as **yourname.Programming123.jpg** along with your hands-on tutorial report:



Lab Week 3-4:

Please work through and then upload a **yourname.Programming456.jpg***screen capture of:*

4. Optimizing Your Code
5. Creating Flexible Functions
6. Creating Robust Applications

Lab Week 5-6:

Please work through and then upload a **yourname.Programming789.jpg***screen capture of:*

7. Verifying Application Behavior
8. Debugging Your Code
9. Organizing Your Projects

Wrap up MATLAB tutorials. Upon completing all of these technical skill building tutorials, you will have acquired a pretty solid undergraduate MATLAB coding skill set and will be well on your way to creating your own elegant robot code. You can continue your self-learning of more in-depth MATLAB skills by reading through and trying some of the more sophisticated features of MATLAB presented on line tutorials on their website