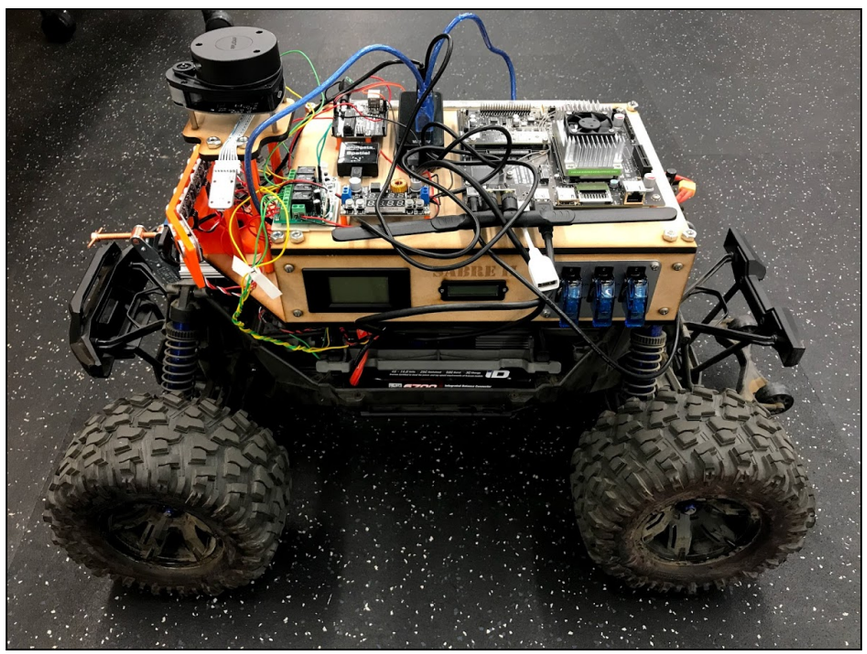
ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

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 (often called 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 a planetary 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.

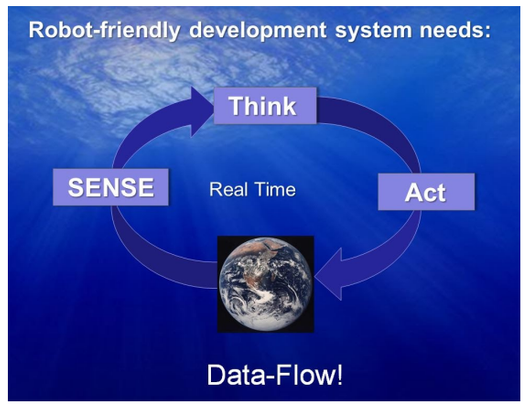


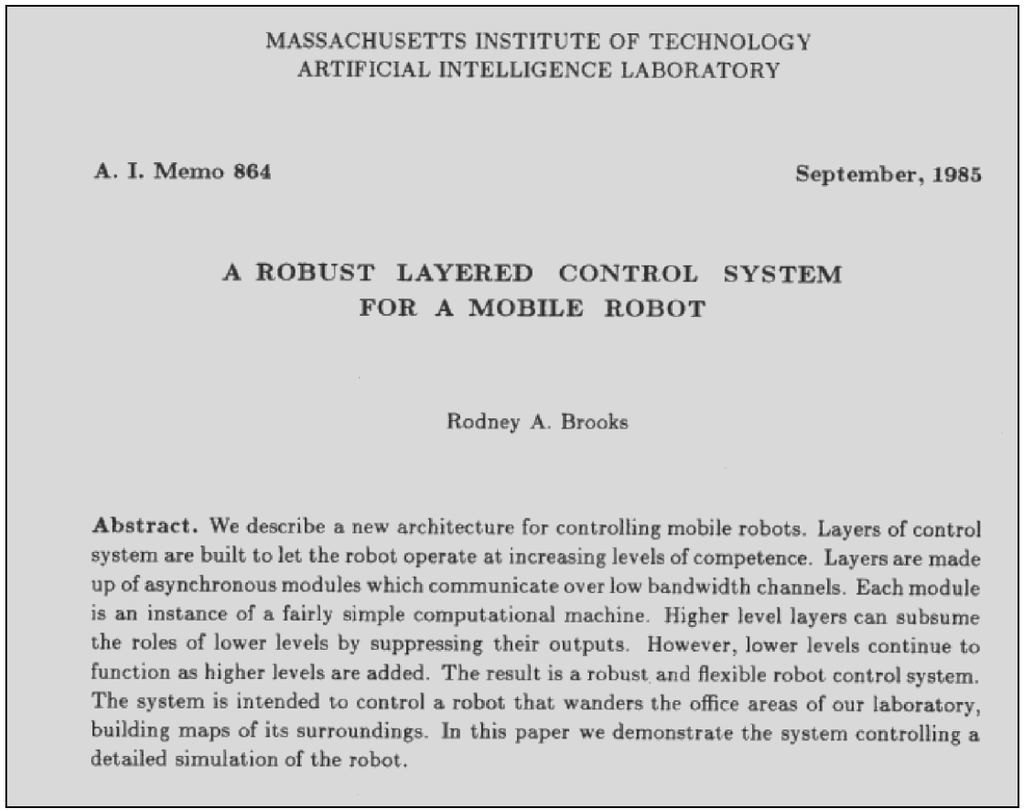
Figure : Sense-Think-Act Control Loop

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}, 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 the Arduino-Raspberry Pi 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) 2021c

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 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:

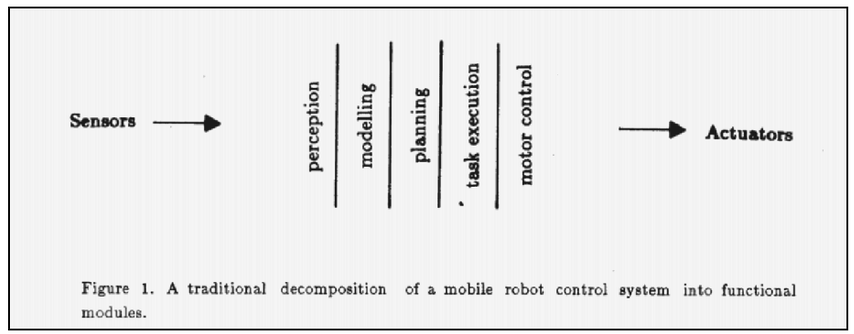


Figure 1. A traditional decomposition of a mobile robot control system into functional modules.

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 breath 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:

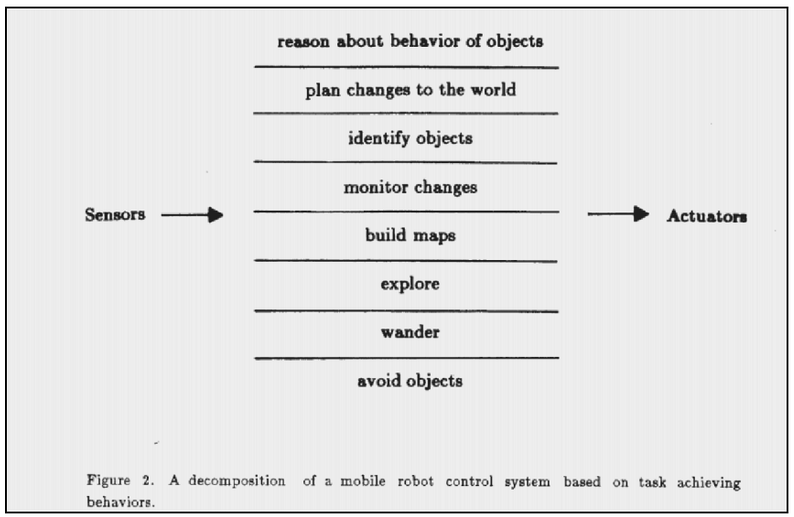


Figure 2. A decomposition of a mobile robot control system based on task achieving behaviors.

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:

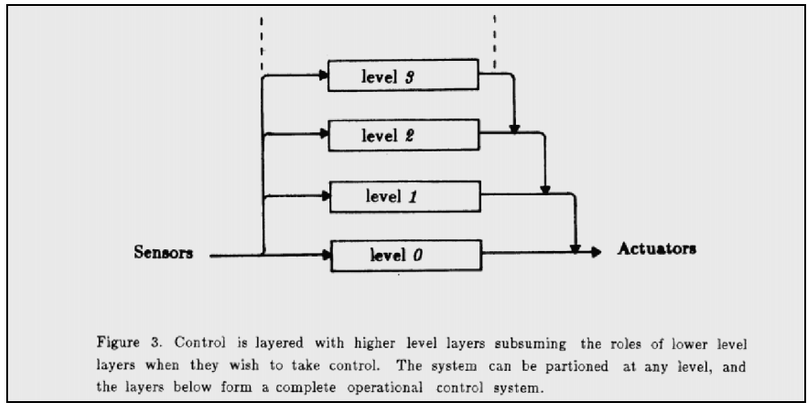


Figure 3. Control is layered with higher level layers subsuming the roles of lower level layers when they wish to take control. The system can be partioned at any level, and the layers below form a complete operational control system.

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

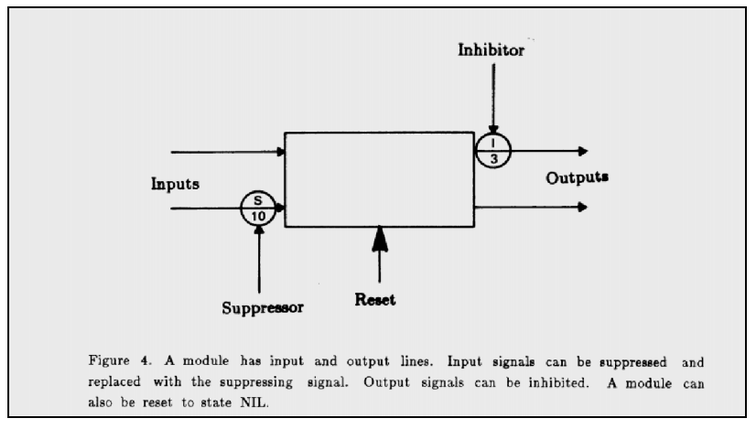


Figure 4. A module has input and output lines. Input signals can be suppressed and replaced with the suppressing signal. Output signals can be inhibited. A module can also be reset to state NIL.

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

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

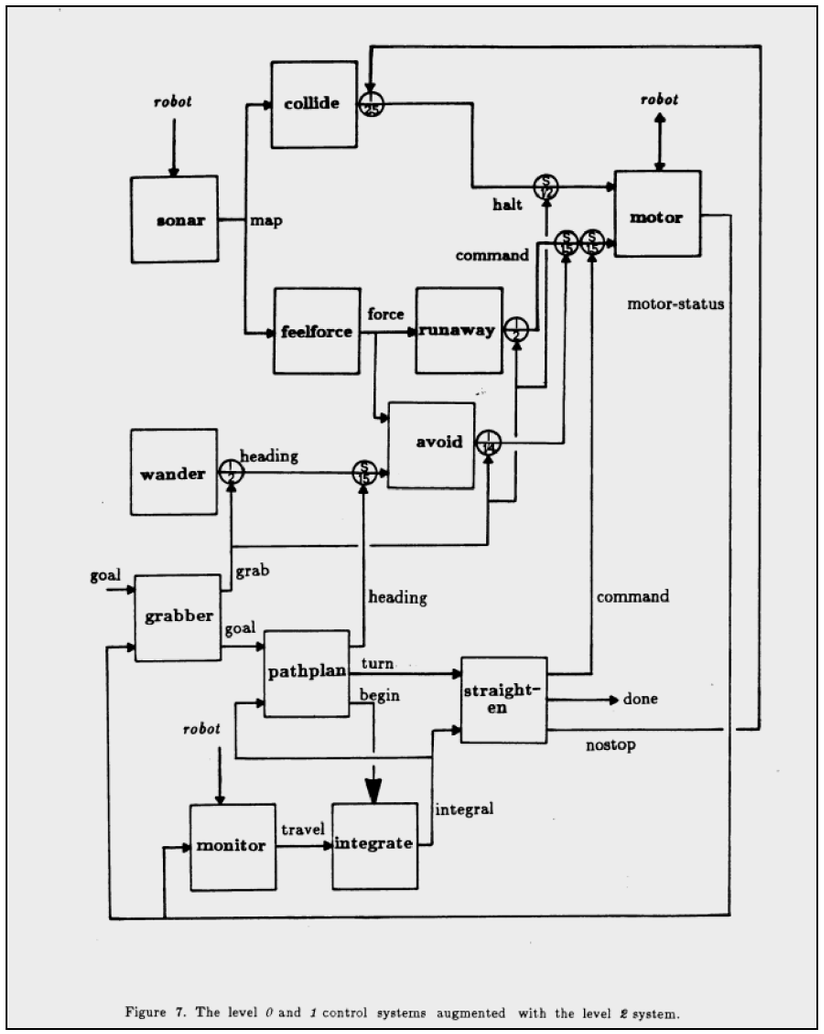


Figure 7. The level 0 and 1 control systems augmented with the level 2 system.

In this lab, we will give you the SENSE and ACT functions for the tug 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-tugboat behaviors:

-Explore: Head out on a random time switching heading
-Avoid: Head away from obstacles in front of boat
-Follow: If you see a sought after target (a brightly colored purple NarWhal plush toy), follow it.

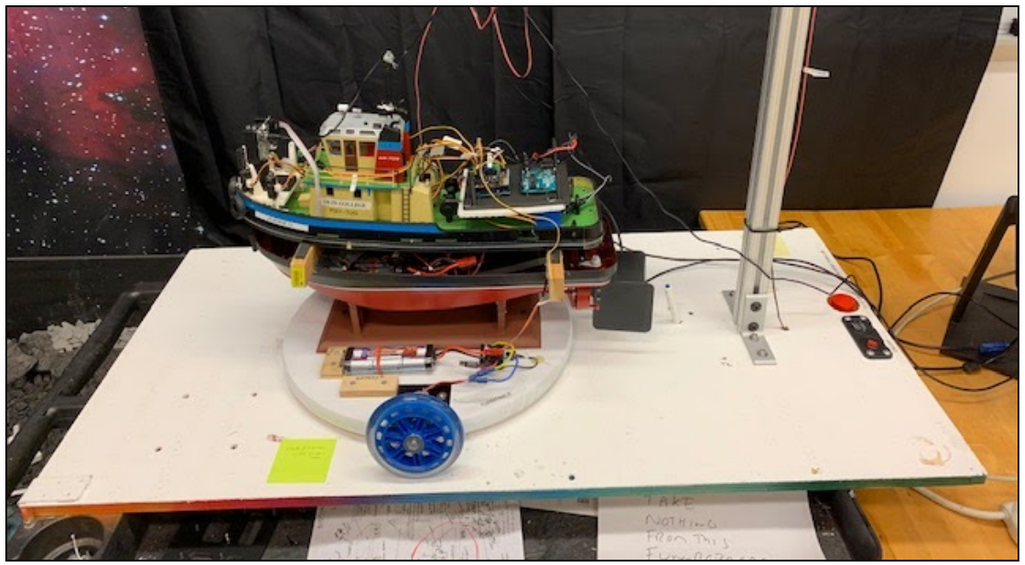
And help your team write two simple arbiters;
-Heading: one to control the commanded rudder angle of the tug
-Speed: and a seperate one to control the speed of its propellers.

We will use a hardware in the loop simulation model to drive your physical tug (on a lab test cart turntable) in a virtual 30' diameter pool.

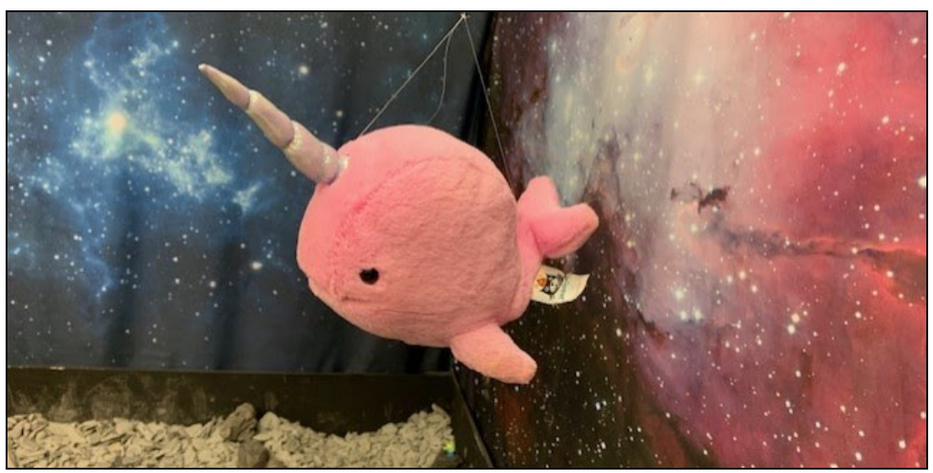
Finally, as in all 4 labs, we will ask you to, in parallel with your lab work (in part 2 of this assignment), to keep advancing your MATLAB programming skills by carrying out an additional set of simple MATLAB tutorials. Please do them spread out over the next two weeks.

Think Lab

This lab contains a fully functional robotic tugboat mounted on a driven turntable. The tugboat has a Raspberry Pi Robot controller on board and forward looking Pi Cam and an array of Sharp Infrared range sensors as its main perception suite, and a set of embedded servo controls for the rudder and propellers. 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 Tug Hardware

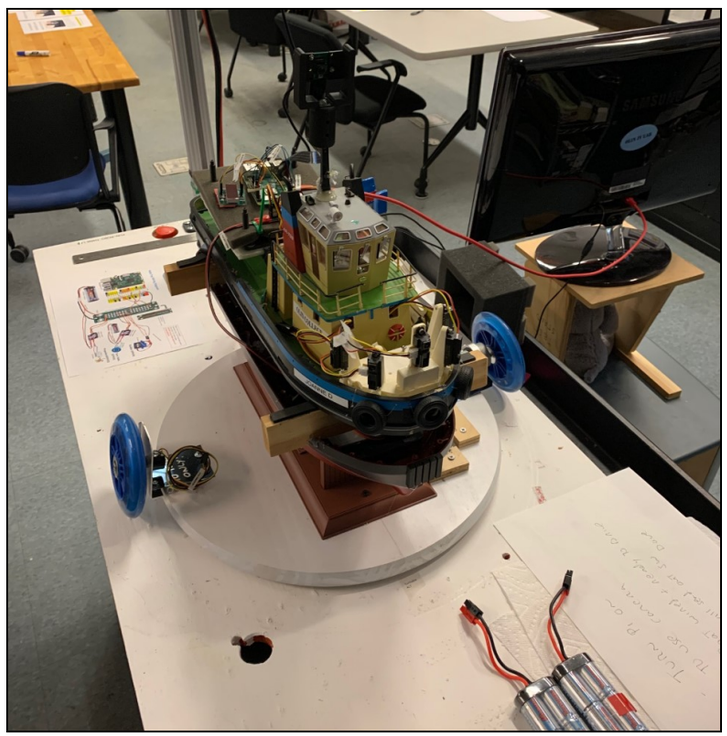


NarWhal Target

Description of Robot Tug SENSE code functions

The THINK lab robot tugboat has three types of sensors on it, an IR-Range suite to see and help avoid obstacles, a Pi Camera to find the purple Narwhal and a turntable mounted Potentiometer to stand in for the boats compass to provide a boat heading. Each sensor system is described below:

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



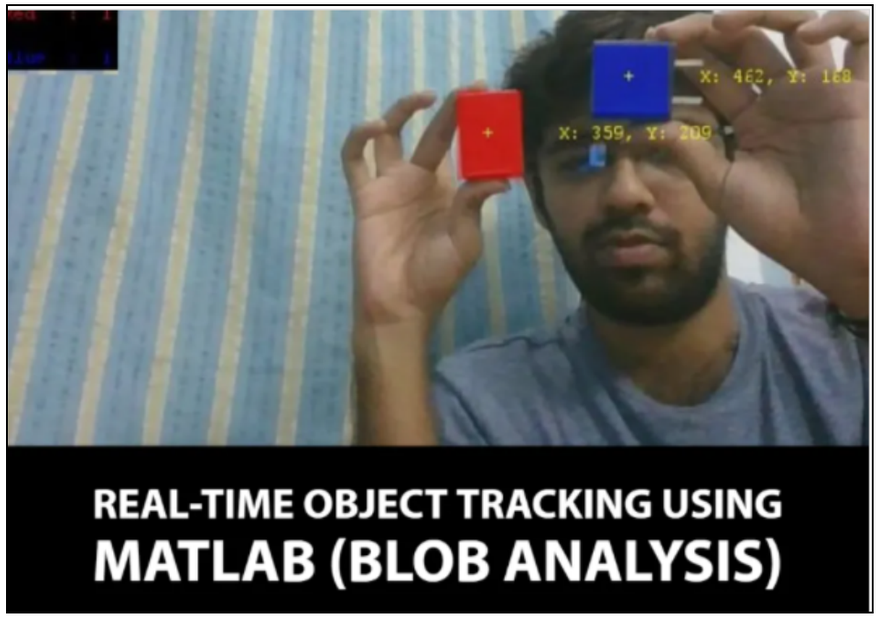
Sharp IR Range Sensor Array and PiCam

They can be used for obstacle avoidance to give you a set of ranges to whatever is in front of your tug. We will be using virtual Sharp IRs in the simulated tank environment to give you range data for what is in front of the tug. 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 PiCam to let you find and track the purple NarWhal. The PiCam is a quite high quality computer vision sensor that works seamlessly with the Raspberry Pi. A quick orientation video from the MathWorks can be found here:

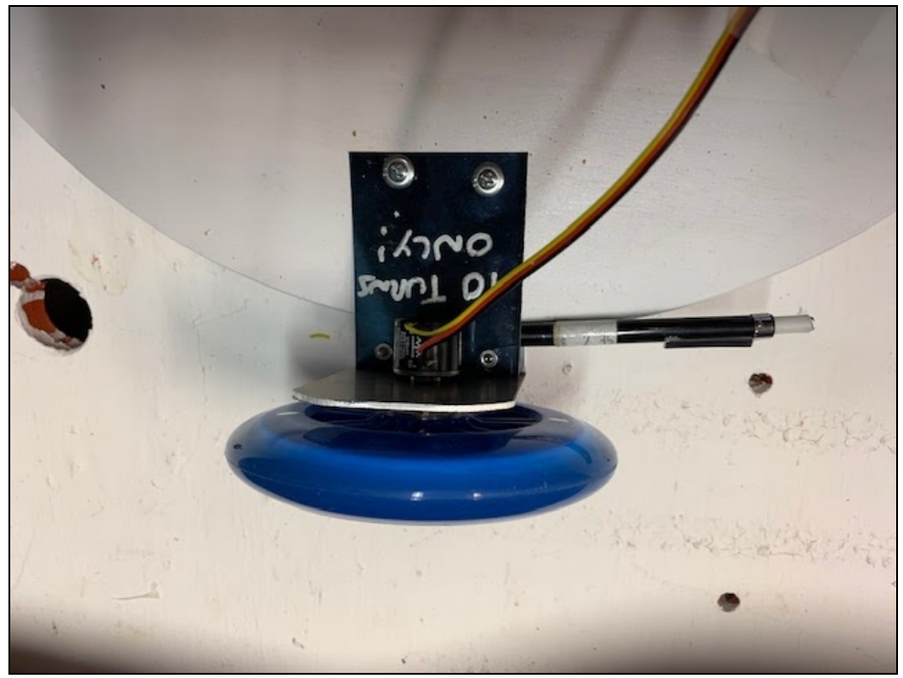
   https://www.mathworks.com/videos/using-matlab-with-a-raspberry-pi-camera-board-94192.html

   A Typical color tracking output of PiCam shown here:



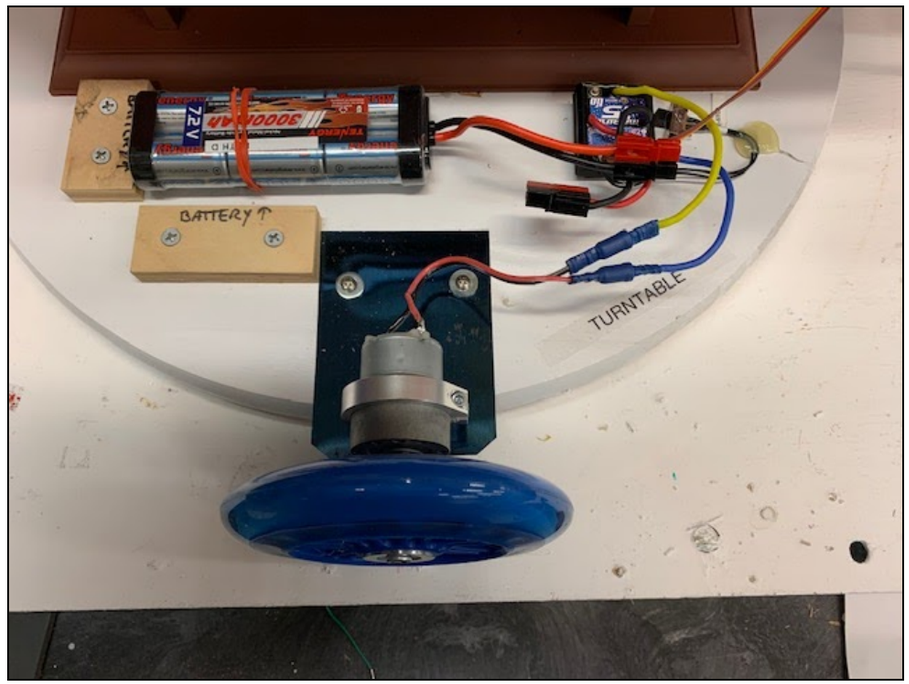
   You will work the basics of computer vision and with the PiCam extensively in one of the SENSE lab's test stations. So that you can focus on THINK for this lab, we will supply you a set of completed functions here.

3. There is a multi-turn potentiometer to tell you the boat'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 Tug connected to the twin rudders. We will provide a function to let you position control them.

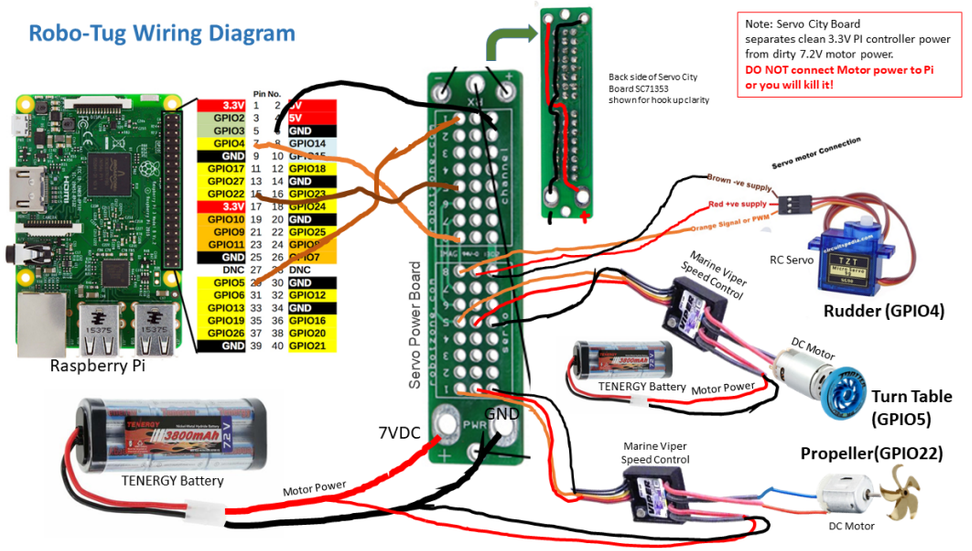
6. The twin propellers are driven by a second Marine-Viper speed control unit located inside the Tugs hull. Its propulsion battery sits inside the hull next to the speed control. We will provide you with a function to control the propellor's speed forward and back.

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

Description of Robot Tug Control Wiring

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

A downloadable larger pdf copy can be found on the ThinkLab canvas site.

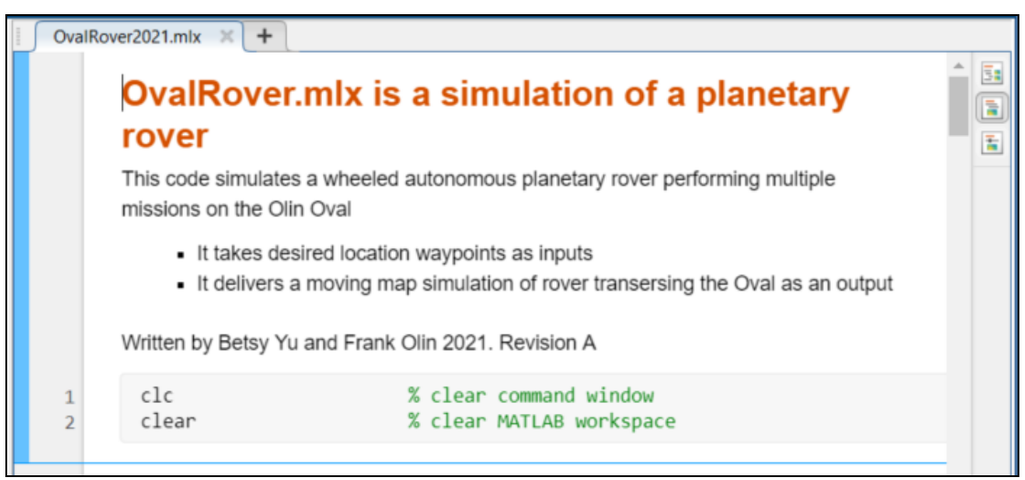


ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

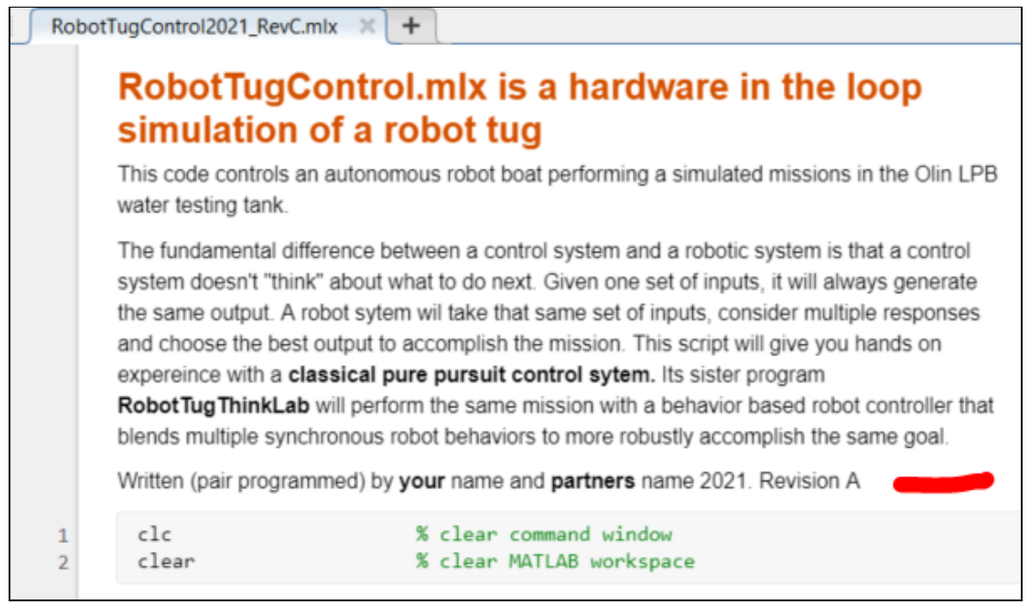
Create ThinkLab Robot Tug control code

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 tugboat pure pursuit way point controller. You will use it to crash into unplanned icebergs in a virtual test pool. The second will copy the first code, make a few light changes to add a behaviour based robot brain, and (hopefully) your Robot Tug will avoid those icebergs with ease. You can demo both on demo day.

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 **RobotTugControl.mlx.** Open **Oval Rover**:



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



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 your most of the Oval Rover code structure for your hardware-in-the-loop tugboat controller and 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) 2021c

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 demo it. We will then reform you into one bigger team and have each sub-team take on writing and deploying one of the tug's new robot behaviors. 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 tug control system (code that runs once)

Set up your Raspberry Pi to control tug. Call **TugRaspPiSetup** function to create and configure your Pi.

Download function from Canvas. See function for documentation on how it works.

```
3  disp('Downloading code to Raspberry Pi may take about 10 sec');
```

```
Downloading code to Raspberry Pi may take about 10 sec
```

```
4  [robotPi,rudderServo,propServo, blinkLED, cam] = TugRaspPiSetup();
```

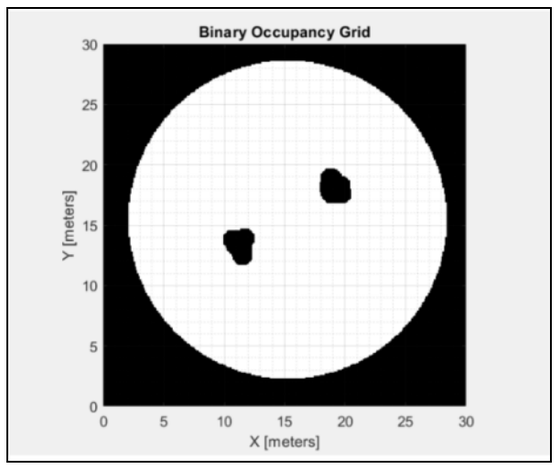
Load a map of Olin Oval test pool (to be built into a Binary Occupancy Grid)

TestPool.png can be downloaded form Canvas too.

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

The additions are using a new function called **TugRaspPiSetup()** in line 4 that will take care of the detailed configuration of the tug's Raspberry pi. You can download it from this assignments Canvas page and make sure to add it to the working directory your team's code is in. You can download and add the **TugServoControl** function as well.

Next you will modify your code to use a premade map of Olin's big circular test pool (out in LPB) with two icebergs floating in the middle of it (download **TestPool.png** and put it in the same directory). If you run this first code section line by line you will notice that it takes about 10-12 seconds to connect and download code to Raspberry Pi (that is normal for embedded type controllers in this class). And it will generate a pretty stand alone figure of the pool:

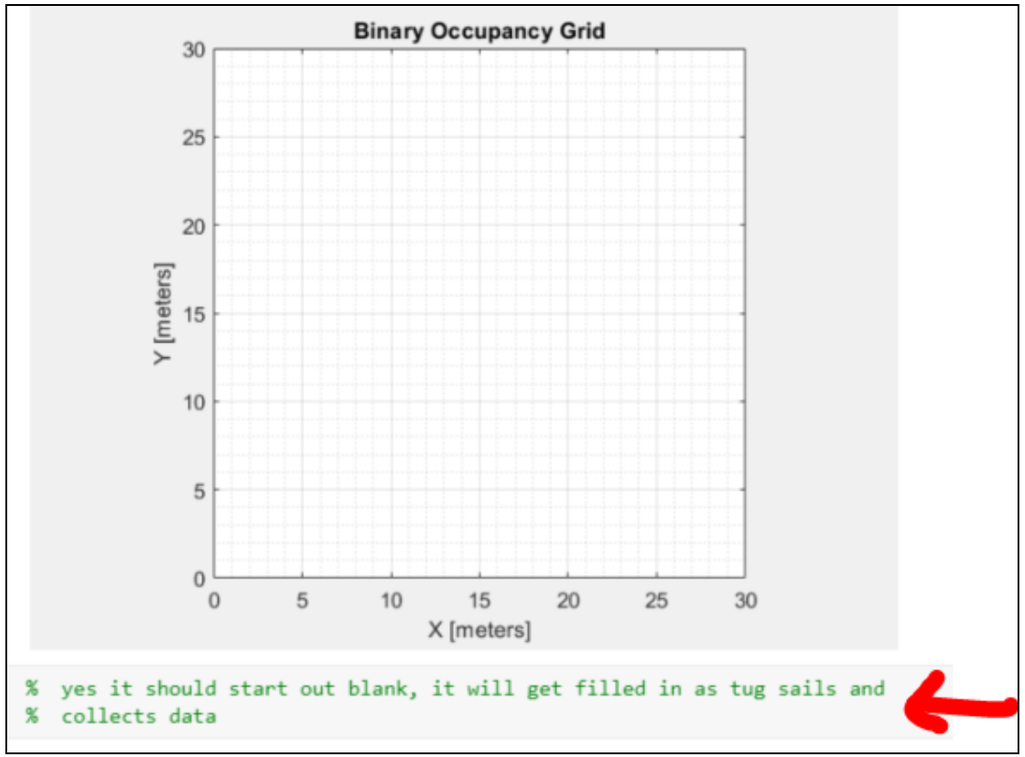


Next you will modify code to switch from a LIdar to 6 discrete SharpIR range sensors, so change code to set up a figure to see this new data in:

Create a sharpIR range sensor based map of world (to be filled in as Tug sails around the pool)

```
23 % Create an empty map of the same dimensions as the test pool map
24 [mapdimx,mapdimy] = size(bwimage);                    % find size of map image
25 sharpIRMap = binaryOccupancyMap(mapdimx,mapdimy,10,"grid");   % make empty map the same size
26
27 % create a free floating figure for map to deal with livescript update problem
28 tugSharpIRScan = figure('name','TugSharpIRScan','NumberTitle','off','Visible','on');
29 figure(tugSharpIRScan)
30         show(sharpIRMap);
31         grid on;
32         grid minor;
```

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c




```
%  yes it should start out blank, it will get filled in as tug sails and
%  collects data
```

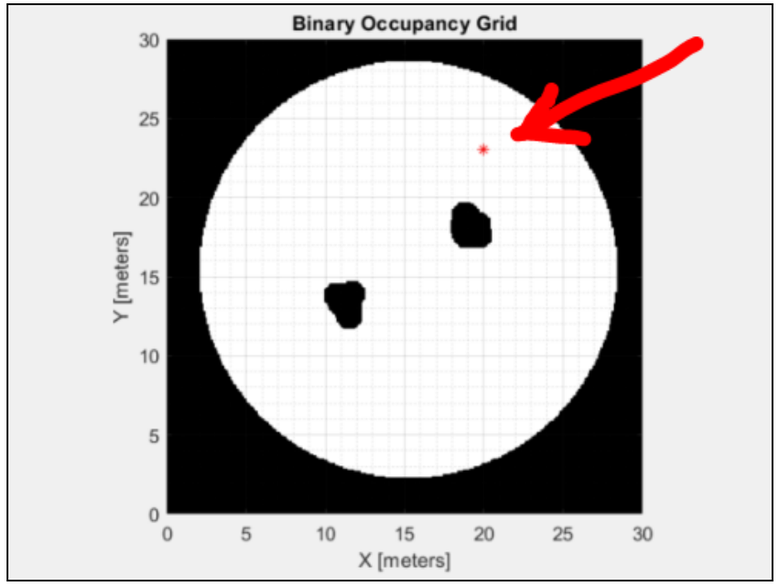
Moving on please modify the original rover control code to reflect the new tug robot as shown below:

```
Create a differential drive robot tug(this model takes robot vehicle speed and heading for inputs)
35  diffDriveTug = differentialDriveKinematics("VehicleInputs","VehicleSpeedHeadingRate");
Create a pure pursuit Tug 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)
36  TugController = 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 pool. 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 Olin Pool 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

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

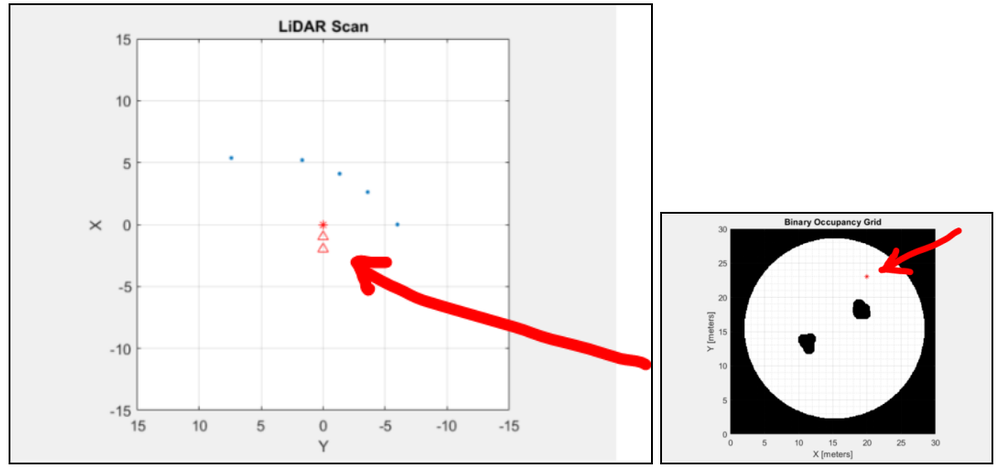
The unchanged part of code places tug in pool to give frame of reference for sharp pool scan to come:



Then update code as shown to take a first test scan of pool around Tug:

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

Generating the following plot:



Where you can clearly see 5 of the six SharpIR range beams getting valid range data from the pool wall. Tug 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 Tug. +X is forward of Tug bow, -X to its stern.

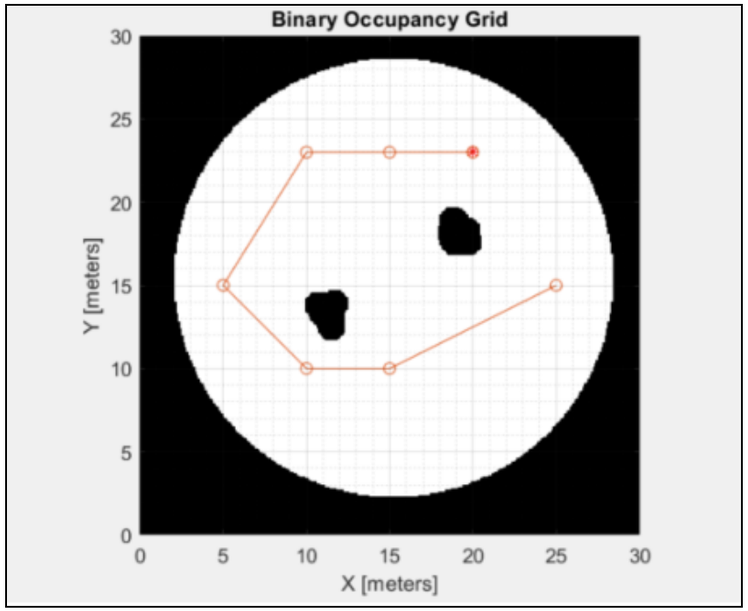
Next we will have you create two waypoint paths to follow, the first is a clear circumnavigation of the tank (looking for NarWhal). The second will take you through an uncharted iceberg. Please comment the second out for now, get your whole system working and then go back and run the second path to see what happens.

Change your code to :

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];    % clesr 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(testPool)                                    % select testPoolMap
hold on
plot(path(:,1),path(:,2), 'o-');                     % plot path on it
hold off
```

Generating the waypoint path:



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

```
72  TugController.Waypoints = path;                % set Tug waypoints
```

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.

```
73  initTugPose = [path(1,1) path(1,2), pi/2]; % store intial loaction and orientation of rover
74  goalWayPoint = [path(end,1) path(end,2)]';  % path end waypoint
75  sampleTime = 0.10;                          % Sample time [s]
76  t = 0:sampleTime:100;                       % Time array
77  robotPoses = zeros(3,numel(t));             % Pose matrix
78  robotPoses(:,1) = initTugPose';             % store robot robotPoses
79  r = rateControl(1/sampleTime);              % reset control loop  rate
80  reset(r);                                   % reset loop time to zero
81
```

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

```
82  runRover = input('Ready to run Tug? Arrange your data plots for clear visability, then type 1 and hit Enter
```

And you have finished revising **your code that runs once.**

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

This is why it is so critically important to write code for future use (by yourself and others), employing the twin tools of clean understandable common 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 **Tug Sense-Think-Act control loop**, first take a look at the following rewrite of the code that runs continually:

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

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

```
controlIndex = 1;                           % create a robot loop control
while (controlIndex < numel(t))             % loop for number of elements in t
    position = robotPoses(:,controlIndex)'; % rovers current position
    tugLocation = 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'-tugLocation);
        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, MapOfPoolInflated, sharpIRMap ,tugSharpIRScan, localSharpIRPlot, sharpIRPod);

    % THINK: compute what robot should do next
    [tugX,tugY,vRef, wRef,robotPoses] = THINK(robotPoses,diffDriveTug,TugController,sampleTime, controlIndex);

    % Check to see if Tug is hitting a physical obstacle
        crash = checkOccupancy(MapOfPoolInflated,[tugX tugY]);
        if (crash == 1)
           disp('Tug crashed, all stop!');
           break;
        end

    % ACT: command robot actuators
    ACT(tugX, tugY, vRef, wRef, rudderServo, propServo, testPool);

    controlIndex = controlIndex + 1;   %increment control loop index
    waitfor(r);                        % wait for loop cycle to complete

end
```

First notice that the main **Sense**, **Think** and **Act** functions got a lot longer. We are explicitly passing 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 step by step and explore what each function does.

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

```
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
```

The **SENSE** function takes in the map of the pool, the Tugs 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 tug'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 tug itself. You will need the second one to design behaviors for tug in the next part of the tutorial work. Please enter and modify your code to look like the code show 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 straight control system running a standard **pure pursuit algorithm** to produce tug actuation commands for the Act function to carry out. If you command tug 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:

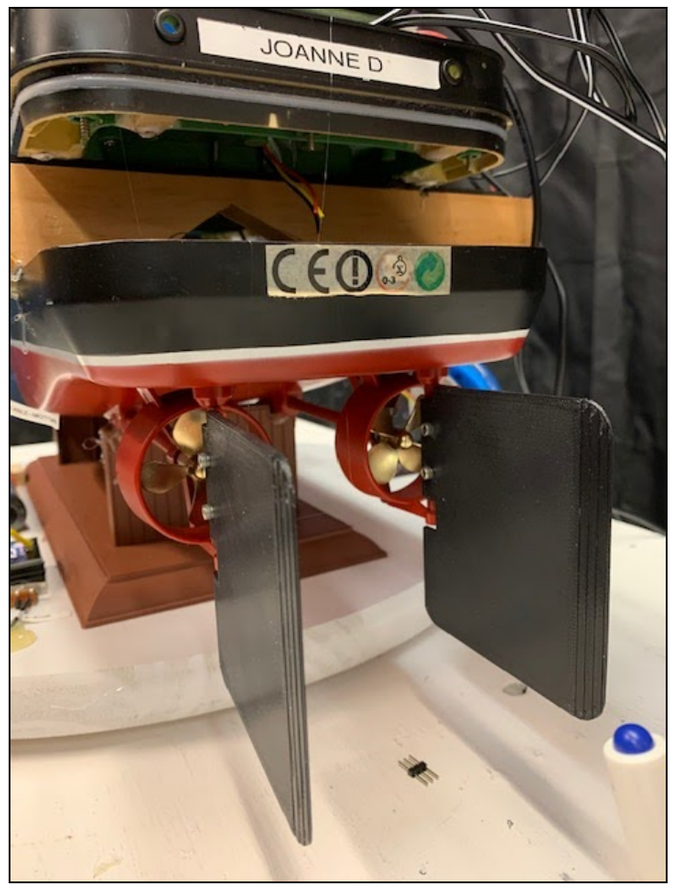
Think Functions (store all Think related local functions here)

```
157  function [tugX,tugY,vRef,wRef, poses] = THINK(poses,diffDrive, ppControl,sampleTime, loopIndex)
158  % THINK gets control commands from pure pursuit controller
159  % to drive to next waypoint. Calculate derivative of robot motion
160  % based on control commands.
161  % inputs are  rover poses, diffDrive dynamics, loop sampleTime, loop index
162  % outputs are roverX, roverY, robot poses (position of robot)
163  % Frank Olin 2021  Rev A
164
165          % Run the Pure Pursuit controller and
166          % convert output to wheel speeds
167          [vRef,wRef] = ppControl(poses(:,loopIndex));
168
169          % Perform forward discrete integration step
170          vel = derivative(diffDrive, poses(:,loopIndex), [vRef wRef]);
171          poses(:,loopIndex+1) = poses(:,loopIndex) + vel*sampleTime;
172
173          % Update rover location and pose
174          tugX = poses(1,loopIndex+1);
175          tugY = poses(2,loopIndex+1);
176
177  end
```

This new function will both calculate the new position and orientation of the tug, 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 Tug's propeller speed and its rudder angle.

Finally take a look at the new **ACT** function. The original function did a good job of plotting the Tug's position on the map of the pool. Because we are going to do hardware in the loop simulation, you will need to add a few more operations to command the Raspberry Pi to drive the propellor speed controller and move its rudders to the correct angle.

Note: Your Tug has a single RC type servo that is connected to the Tugs twin rudders via a linkage. It can move them from -45 to 45 degrees with 0 degrees being straight on. The Tug also has a Marine Viper 15 speed control that commands the two DC motors driving the two propellers from full reverse to full forward speed. In practice you'd develop calibration curves for both actuators that mapped them into the 0-180 control that the Matlab Raspberry Pis servo function takes as an input. You will have either had or will get to have hands-on experience with both in the ACT lab test stand. For this lab, we will give you those functions to use without further development.



Tug twin rudders and propellers

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

Act Functions (store all Act related local functions here)

```
178  function ACT(tugX, tugY,vRef, wRef, rudderServo, propServo, fig_testTrack)
179  % ACT Increment the robot pose based on the derivative.
180  % inputs are curretn X and Y position of rover and figure to plot rover in.
181  % outputs are no  outputs
182  % Frank Olin 2021  Rev A
183          %Plot Tug location in pool
184          figure(fig_testTrack)
185          hold on
186          plot(tugX,tugY, 'b*');              % plot robot position in oval
187          hold off
188
189          % Drive Tug hardware serco motors
190          rudderAngle = wRef*15;     % convert angular rate wRef to rudder degrees
191          propSpeed = vRef;          % vref (-2 to 2 m/sec)
192          ok = TugServoControl(rudderServo, propServo, rudderAngle, propSpeed);
193
194  end
```

It now takes in **vRef**, and **wRef** as commanded inputs as well as the objects naming the rudderServo and propServo. The first part of the code draws the tugs position on the pool map, the second uses the provided **TugServoControl** to drive the actual tug rudder and propeller actuators so as the tug goes about on its virtual mission in the simulated tank, the actual tug will turn rudders and drive props 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 waders and chase your boat through chest-high water in duckweed when a small bit of code goes wrong.

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)

```
controlIndex = 1;                          % create a robot loop control
while (controlIndex < numel(t))            % loop for number of elements in t
    position = robotPoses(:,controlIndex)';  % rovers current position
    tugLocation = 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'-tugLocation);
        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, MapOfPoolInflated, sharpIRMap ,tugSharpIRScan, localSharpIRPlot, sharpIRPod);

    % THINK: compute what robot should do next
    [tugX,tugY,vRef, wRef,robotPoses] = THINK(robotPoses,diffDriveTug,TugController,sampleTime, controlIndex);

    % Check to see if Tug is hitting a physical obstacle
        crash = checkOccupancy(MapOfPoolInflated,[tugX tugY]);
        if (crash == 1)
           disp('Tug crashed, all stop!');
           break;
        end

    % ACT: command robot actuators
    ACT(tugX, tugY, vRef, wRef, rudderServo, propServo, testPool);

    controlIndex = controlIndex + 1;   %increment control loop index
    waitfor(r);                        % wait for loop cycle to complete

end
```

This control loop consists of a Matlab **while** loop that will run continuously until the Tug 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 a clean shutdown. **SENSE** uses the sharp lrs to see what is around Tug, **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 tug 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 the Pi in a good state each time you do a code development run. Please modify your existing shutdown code to look like this.

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

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

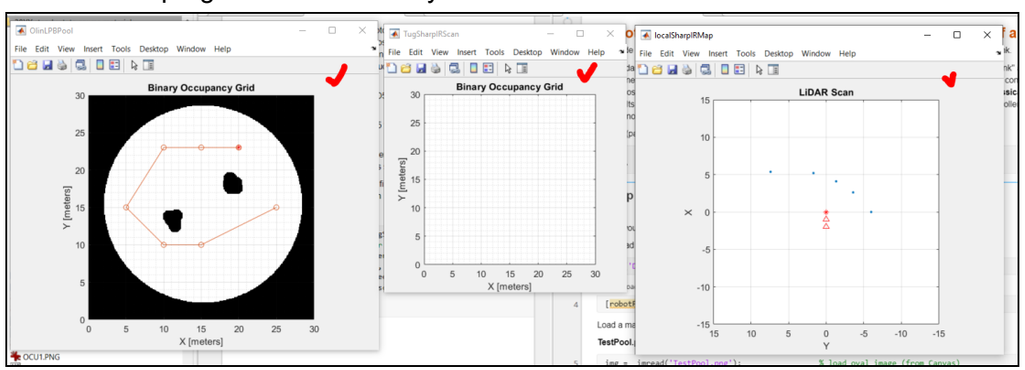
communication paths. This keeps systems form jamming when you want to run again.

```
117   % Stop program and clean up the connection to Raspberry Pi
118   % when no longer needed
119     writePosition(rudderServo, 90);      % always end servo at 0.5
120     writePosition(propServo, 90);        % always end servo at 0.5
121     clc
122     disp('Raspberry Pi program has ended');
```

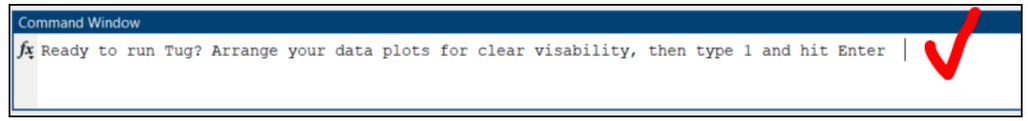
```
Raspberry Pi program has ended
```

```
123     clear robotPi
124     beep              % play sytem sound to let user know program is ended
```

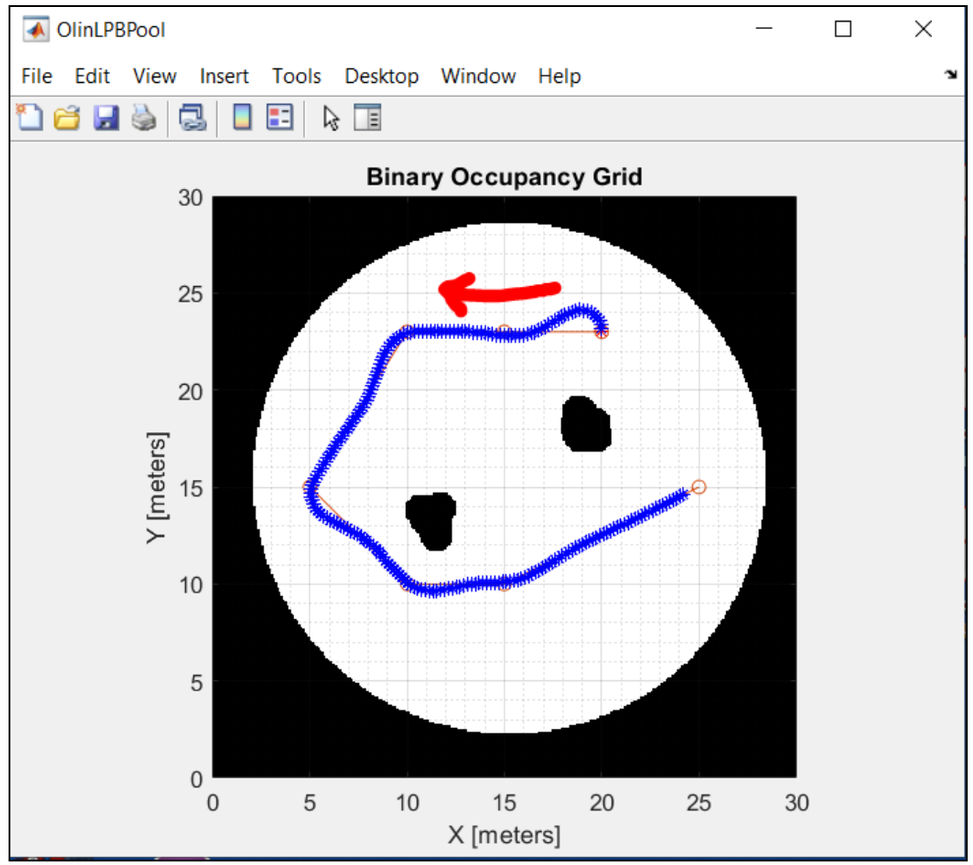
Ok, lets run your new Tug control code next and see what happens, make sure the Tug test bench is properly turned on, the two Pi functions we gave you are in same folder as this program and then run your code from the start:



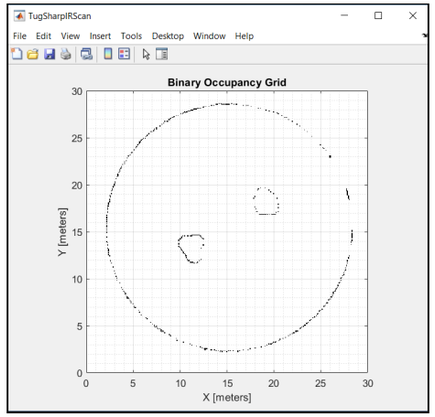
Your code should pop up an **OlinLPBPool** map, a **TugSharpIRScan** map, and a **localSharpIRMap**, as shown above. And then pause to wait for your command to proceed.



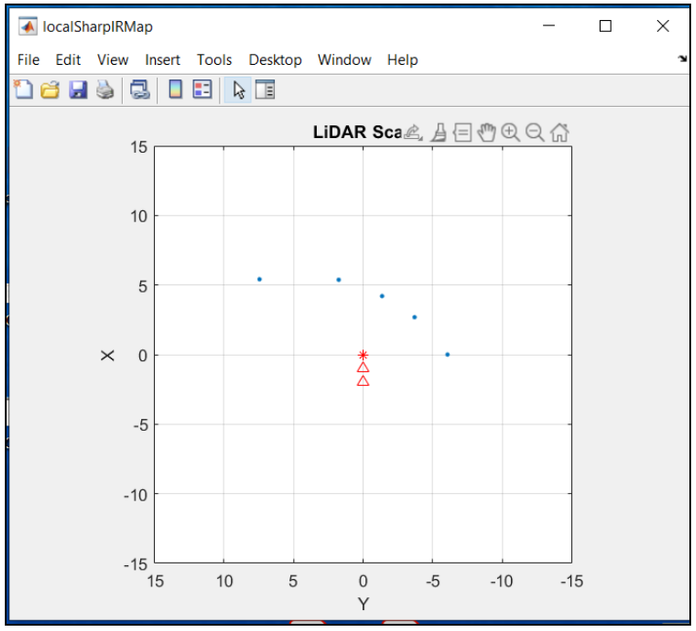
Following that command, your Tug and its pure-pursuit control system should nicely follow the selected waypoints in a circumnavigation of the test pool:



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



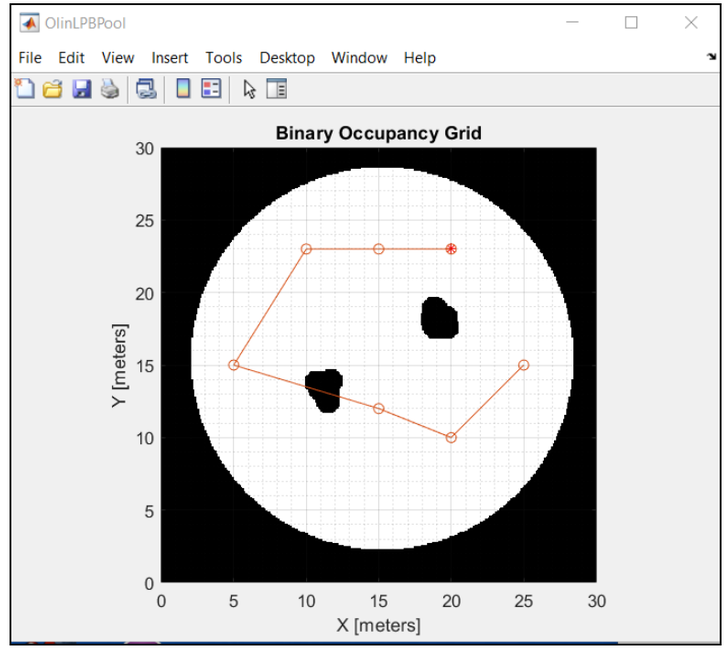
As well as got live data from your 6 forward pointing Sharp IR range sensors:



Run the same simulation a few times. Watch the Tug centered Sharp IR screen as your Tug steams through the pool. You will need to develop a good sense of what the sharps can and can't see to develop effective behaviors in the next section of this work. **Congratulations! You've built a great new Robot Tug control system.** If this were a controls 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.

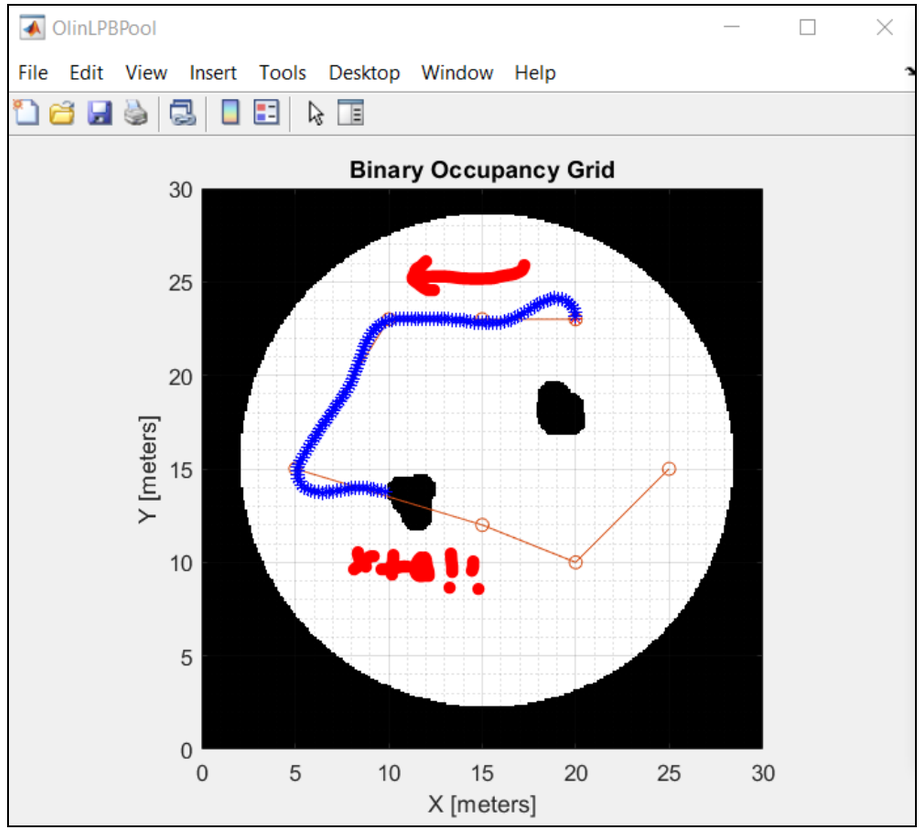
```
Create a test path of waypoints to sail through the pool for gathering range sensor readings.
64  %path = [ 20 23; 15 23; 10 23; 5 15; 10 10; 15 10; 25 15];    % clesr path waypoints
65  path = [ 20 23; 15 23; 10 23; 5 15; 15 12; 20 10; 25 15];  % crash path waypoints
66
```

Run your code again with the new Tug path waypoints:



:

And get:



WHAM !!! Congratulations again! You just crashed your Tug head-on, right into an iceberg at full throttle.

This is a somewhat contrived, but graphically convincing example of **the brittleness of using just a control system for a dynamic robot** that has to operate in an ever changing 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 Tug

How you go about fixing this, crash into an iceberg problem, is the subject of the second part of this tutorial. Recollecting as a 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.

Fundamentals of robot behaviors-based control

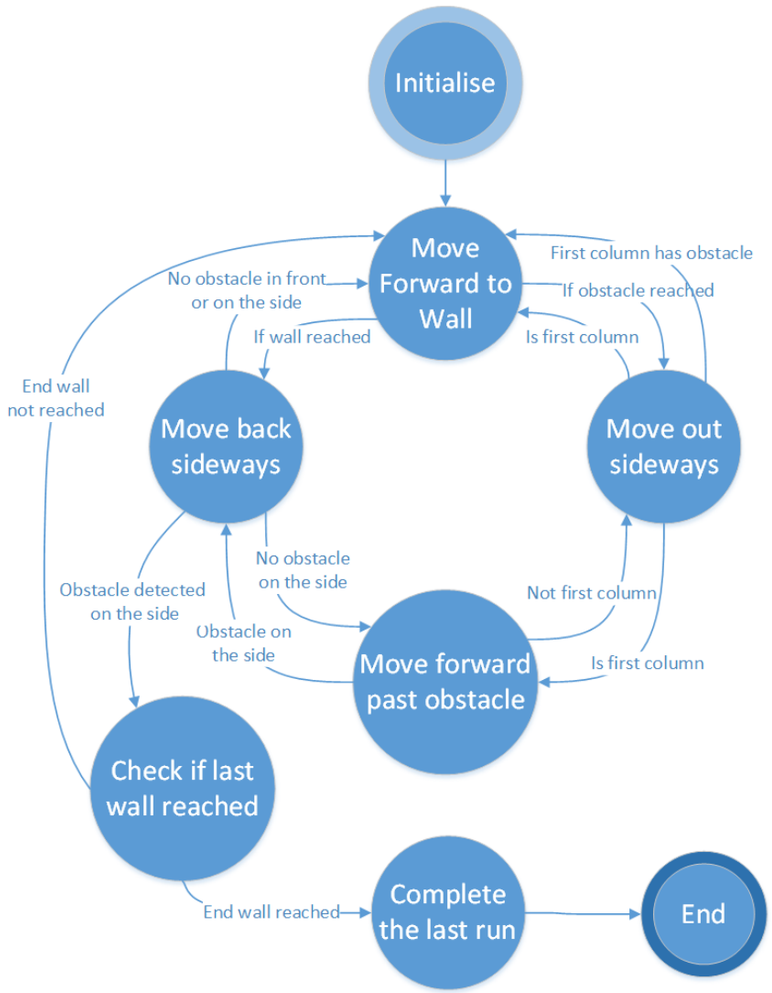
Complex robots usually use some form of behavioral control at a higher planning level and classical control at a lower actuator level.

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, block diagram like, graphic called a **Finite State Diagram**. This 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 flows from one state to the next along these event paths until the mission is complete or an error that can't be overcome is reached.

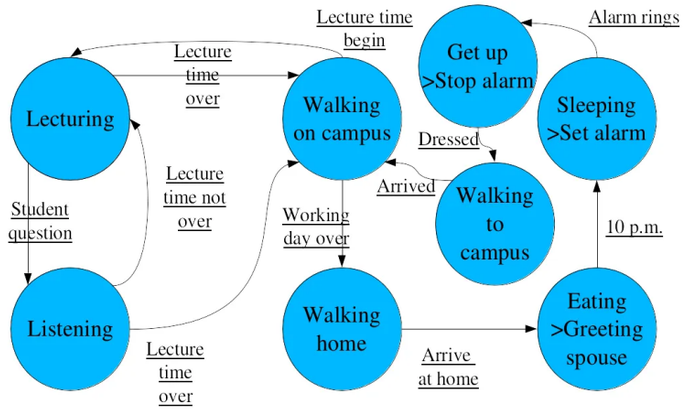
This one **FSM** diagram is a very clear, graphical way to portray, not only what the robot is currently doing, but also what your team designed (and hopes) your robot will do. FSM diagrams are also great tools to facilitate critical conversations among your future robot design team members, many of whom won't be active coders. Even a gearhead ME can understand them!

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

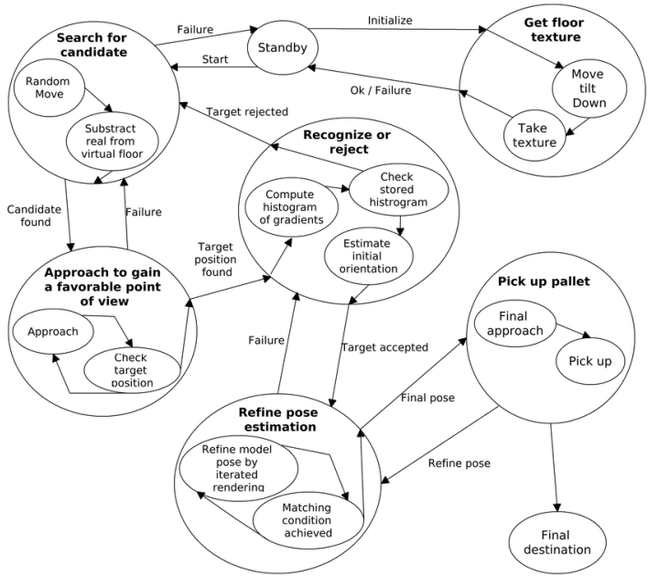


Example Robot Behavior FSM Diagram

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

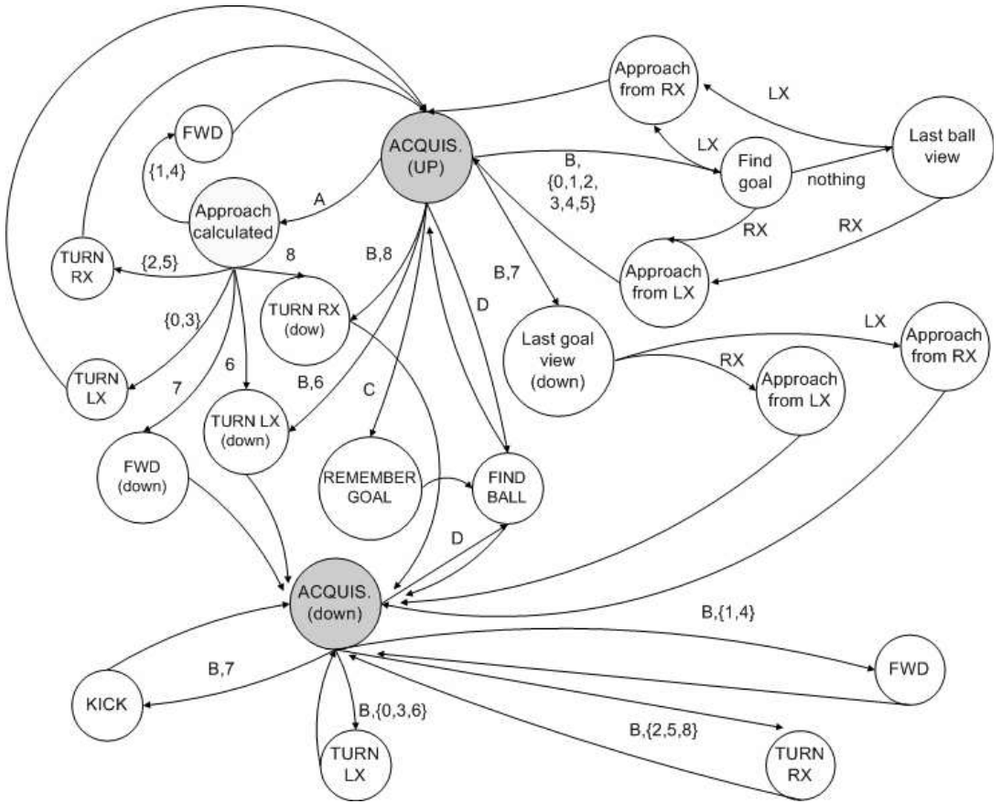


A robot warehouse worker:



A Robot soccer player:

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

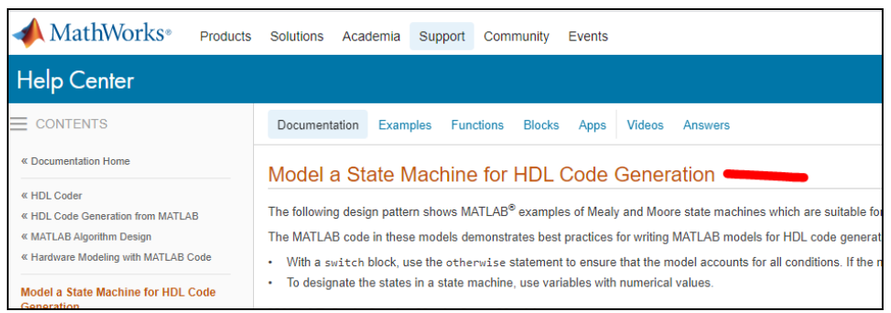


You can see the control flow here, in each of the above cases. The robot is in one of many possible states exhibiting a behavior, it stays in that state until an event occurs to move it to a new state. Over the course of a task or mission, the robot will cycle through many, but probably not all the possible 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 Tug.

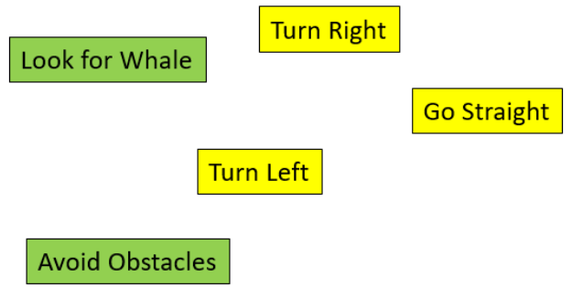
I'll ask you to pause here and go look at the 4-short 3m video tutorials listed on the Canvas page for this tutorial, about what a finite state machine is and how to draw Mealy or Moore Finite State Machine diagrams before proceeding:

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):

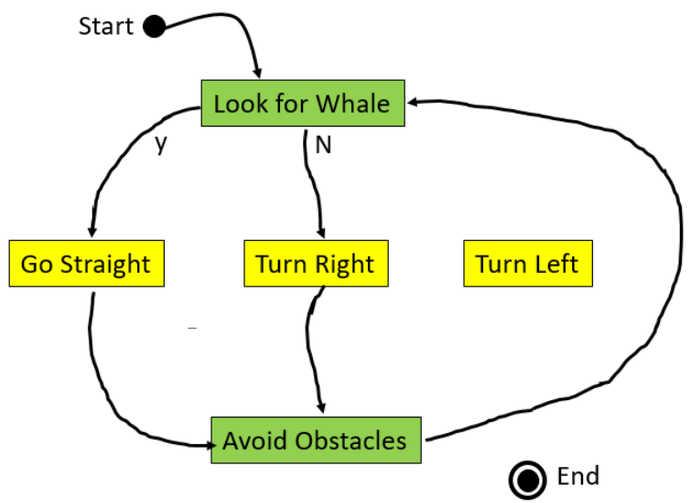
https://www.mathworks.com/help/hdlcoder/ug/model-a-state-machine-for-hdl-code-generation.html



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

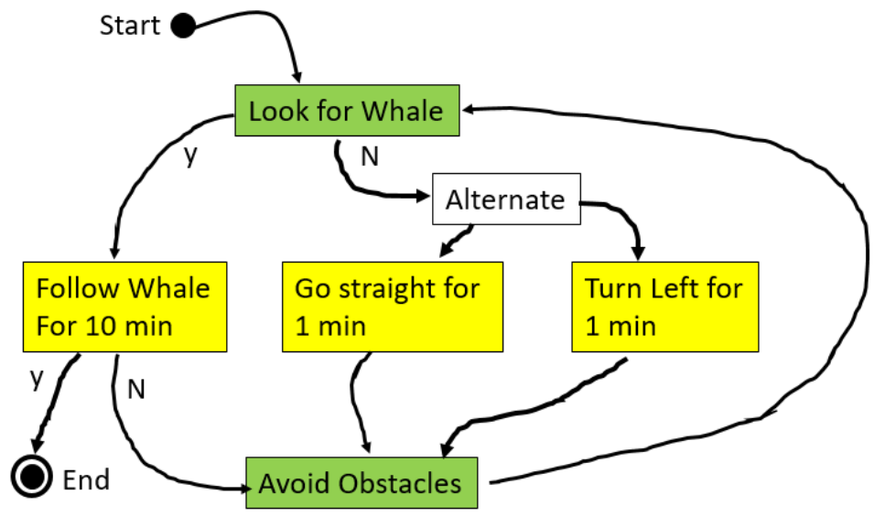


For clarity, we will color code behaviors as Green, for things Tug should always be doing, and Yellow for behaviors that Tug switches between. For example, there is no practical reason to not always look for the NarWhal, but the Tug can only be going straight or turning, it can't be doing both. Let's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



Here, after starting, the Tug looks for the whale, if it sees it, the **Tug goes straight** toward it, if not, **Tug 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 Tug doesn't see the whale it just loops to the right in place for ever 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, **Tug goes straight** for 1 min, then **Tug 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!



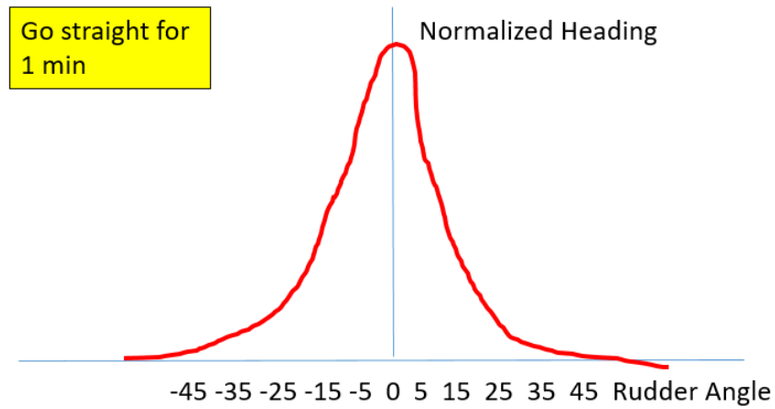
This is just one of many possible ways to approach an autonomous whale-following mission. Your team will get to design your own behavior based FSM in the final section of this tutorial. But it has all of the necessary features to demonstrate what a FSM looks like and you can see one of the strengths 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

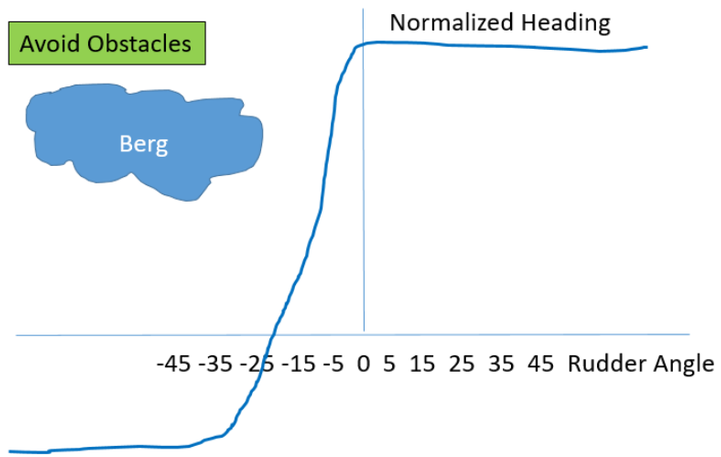
```
n = input('Enter a number: ');

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 you 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:

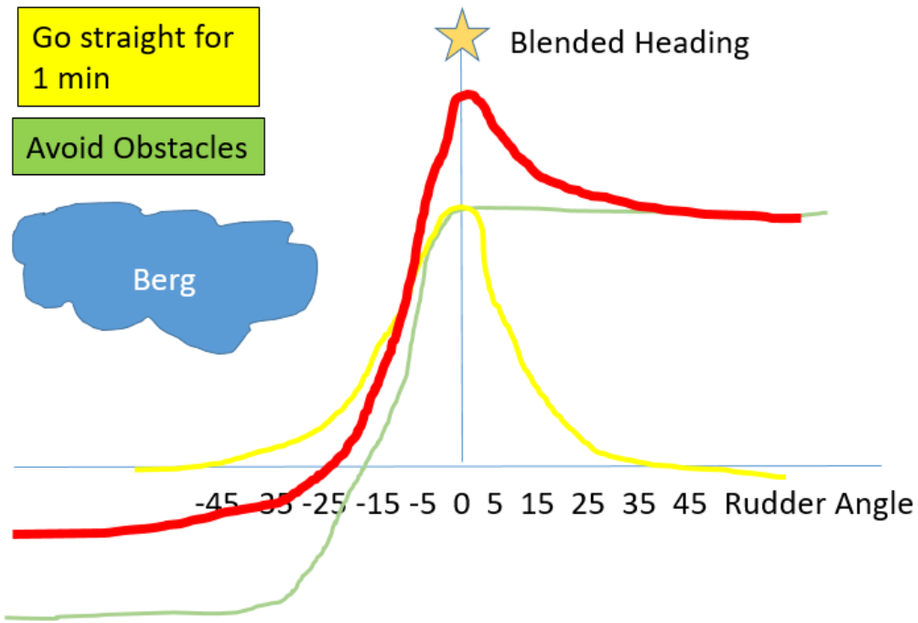


Then consider Obstacle Avoidance behavior that will allow any rudder angle except for those that will cause a collision:



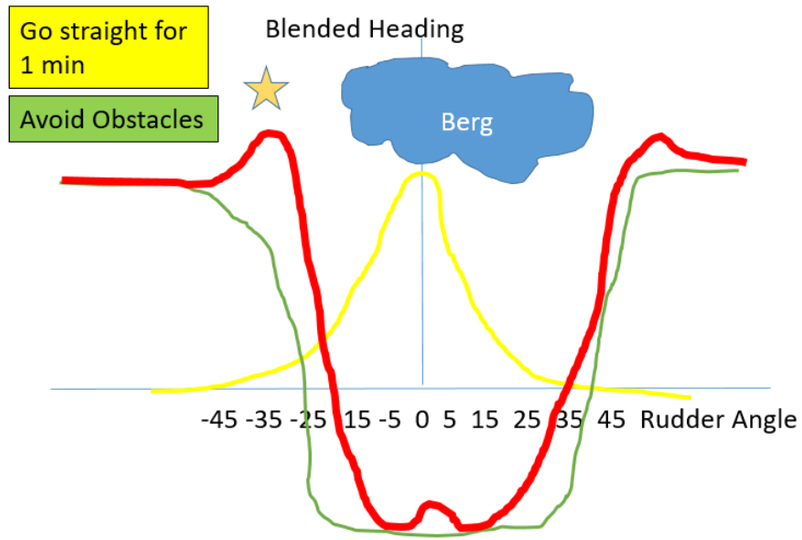
These probabilistic behavior curves are sometimes affectionately called **brain waves**. If we add the two brain waves together and then look for where the

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 tug encounters as it steams through the pool. If the iceberg drifts in front of the tug 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 your robot controller. Using this arbiter, your Tug 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 Tug is thinking.**

Consider what would happen if we replicated the Tug 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 tug will safely steam around the iceberg either to its right or left depending on which peak is higher.

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

The arbiter can do a simple vector addition of behavior outputs, find the maximum of that new vector and send the rudder angle that corresponds to that maximum to the ACT code to drive the servos to that desired rudder angle.

In practice you could have one arbiter for each commanded servo motor. One for the rudder and one for the propellor speed. As a conceptual example, there can be many officers on your tug'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 behaviors desired rudder positions, into a single, arguably best, one that can be sent to the rudder.

Let's move on and write the Matlab code to demonstrate a simple example of this. Open up your **RobotTugControl.mlx** and save a new copy as **RobotTugFSM.mlx.** Here again, we are going to use the power of building on working existing code, rather than trying to write this all again from scratch.

Your full team can modify the first section as show:

RobotTugFSM2021_RevF.mlx

RobotTugFSM.mlx is a hardware in the loop simulation of a finite state machine controlled robot tug

This code controls an autonomous robot boat performing a simulated missions in the Olin LPB water testing tank.

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 robustly blend multiple synchronous robot behaviors to more robustly accomplish the same mission goal.

Written (pair programmed) by **your** name and **partners** name 2021. Revision A

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

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

Set up tug control system (code that runs once)

Set up your Raspberry Pi to control tug. Call **TugRaspPiSetup** function to create and configure your Pi.

Download function from Canvas. See function for documentation on how it works.

```
3  disp('Downloading code to Raspberry Pi may take about 10 sec');
```

```
Downloading code to Raspberry Pi may take about 10 sec
```

```
4  [robotPi,rudderServo,propServo, blinkLED, cam] = TugRaspPiSetup();
```

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

TestPool.png can be downloaded form Canvas too.

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

Just go through line by line and make sure there are no small discrepancies.

****Don't close out of the figures when they pop up, since this will result in `Error using figure`. If you accidentally close them, re-run the code to create the figure again before proceeding****

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

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 Tug sails around the pool)

```
% 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 si

% create a free floating figure for map to deal with livescript update problem
tugSharpIRScan = figure('name','TugSharpIRScan','NumberTitle','off','Visible','on');
figure(tugSharpIRScan)
        show(sharpIRMap);
        grid on;
        grid minor;
```

Check through the next section:

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

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

Create a pure pursuit Tug 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)

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

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 Olin Pool 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 fc
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 Olin Oval Map
figure(testPool)                                 % designate free floating poolTrackMap to
hold on
plot(20,23, 'r*');                               % plot robot initial position in pool
hold off
```

Verify your team's code matches:

```
% Generate a SharpIR test scan from test position.
    [ranges, angles] = sharpIRPod(testPose, MapOfPoolInflated);
    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,
% tug 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 part but save (you may need 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];    % clesr 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(testPool)                                        % 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:

```
%TugController.Waypoints = path;                  % set Tug waypoints
```

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)

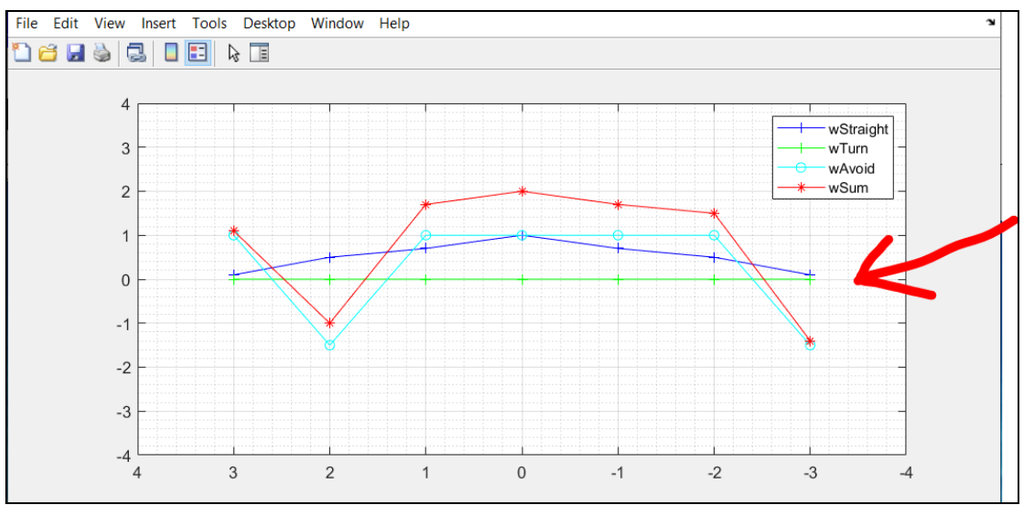
Create a free floating figure for robot behavior brainwave display

```
% 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;
```

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

```
runRover = input('Ready to run Tug? Arrange your data plots for clear visability, then type
```

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

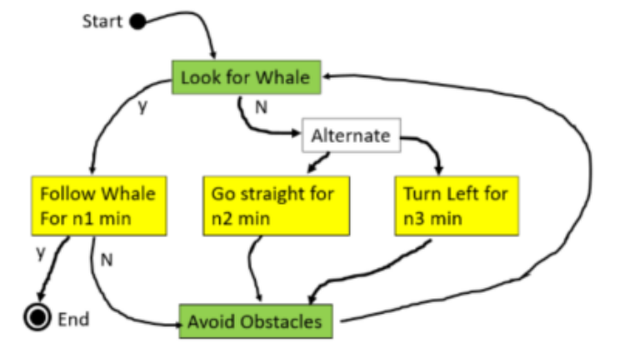


That was the easy part, now for the heavy lifting, please modify the control loop (code that runs over and over) to have the following header:

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** 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:

```
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
followClock = 0;                           % create a whale 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 Tug velocity
wRef=0;                                    % initial Tug 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 **IsThereAWhale** function

```
while (controlIndex < numel(t))              % loop for number of elements in t
    position = robotPoses(:,controlIndex)';  % rovers current position
    tugLocation = 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'-tugLocation);
      %  if (dist < 1.0)                     % robot reaches end goal
      %      disp("Goal position reached")
      %      break;                          % stop control loop
      %  end

    % SENSE-------------------------------------------------------------------
      % SENSE: collect data from Sharp IR sensor array
      [ranges, angles] = SENSE(position, MapOfPoolInflated, sharpIRMap ,tugSharpIRScan, localSharpIRPlot, harpIRPod);
    % SENSE: Look for purple Narwhaal 0=whaleNotSeen, 1 = whaleSeen
      [whaleSeen, headingToWhale] = IsThereAWhale(); % placeholder function for future Pi Cam use
```

The **IsThereAWhale** 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 Pi Cam on the Tug and a bit of startup code you can download from the Canvas site to create your own working version of this.

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
```

And the new **IsThereAWhale** one:

```
function [whaleSeen, headingToWhale] = IsThereAWhale()
% IsThereAWhale is a placeholder function for Think lab student generated
% Rsp-Pi Cam code to find and track a purple Narwhaal
    whaleSeen = 0;          % not seen = 1, seen = )
    headingToWhale = 0;     % replace with code that coverts the x coordinate of whale blob to a tug rudder heading
end
```

Think Functions (store all Think related local functions here)

It's good coding practice to create placeholder functions like this, when you generate your original code. They help hold the main flow of structure in place. And they can

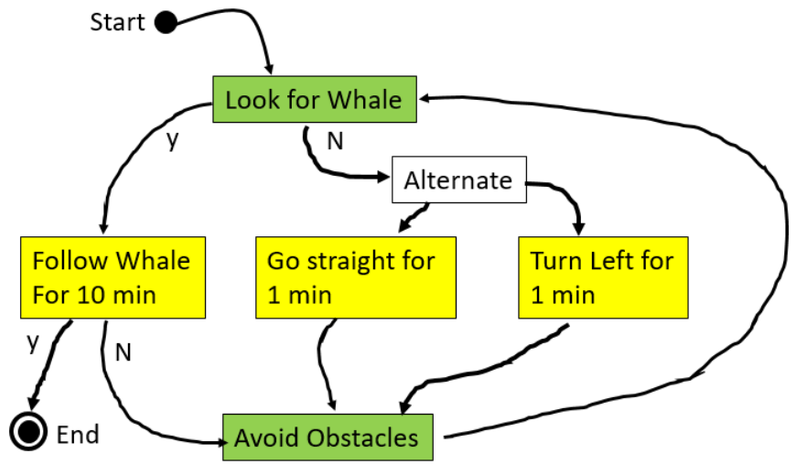
then be easily filled out by a sub-team without struggling to not break the main structure of the program.

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--------------------------------------------------------------------
  % Check for whale and switch Tug behvior based on positive detection
  switch whaleSeen
    case 0        % whale NOT SEEN
        behavoirCycleTime = behavoirCycleTime+1; % increment behavoir alternating clock
        if (behavoirCycleTime < legTime)         % run GoStright for 10 cycles
           [tugX,tugY,vStraight,wStraight, robotPoses] = THINKGoStraight(robotPoses,sampleTime,diffDriveTug,...
               controlIndex,vRef,wRef);
           vTurn = [0 0 0 0 0 0 0];         % if going straight surpress all turning propellor commands
           wTurn = [0 0 0 0 0 0 0];         % if going straight surpress all turning rudder commands
        else                                % run TurnLeft for 10 cycles
           [tugX,tugY,vTurn,wTurn, robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveTug,...
               controlIndex,vRef,wRef);
           vStraight = [0 0 0 0 0 0 0];         % if turning surpress all straight propellor commands
           wStraight = [0 0 0 0 0 0 0];         % if turning surpress all straight rudder commands
        end
        if (behavoirCycleTime > legTime +5)
            behavoirCycleTime = 0;                % reset alternating behavior timer
            legTime = random('Normal', 20, 40);   % choose a random transit leg time
        end
    case 1   % whale SEEN
        if (followClock < 10)
            followClock = followClock+1; % increment whale follwing time
            % place student whale following behavior function here
        else
            break;                      % stop loop mission complete
        end
    otherwise
        % assume whale not seen
  end
  % Run Obstacle Avoidance behavior based on Sharp IR range data
  [vAvoid,wAvoid] = THINKAvoid (ranges, angles);

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

  % Check to see if Tug is hitting a physical obstacle
  crash = checkOccupancy(MapOfPoolInflated,[tugX tugY]);
    if (crash == 1)
       disp('Tug crashed, all stop!');
       break;
    end
```

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 whale **case 0** and a following behavior **case 1** if you have. Both cases then feed into 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 Tug linear **vRef** and rotary velocity **wRef** into the following ACT function to carry out.

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



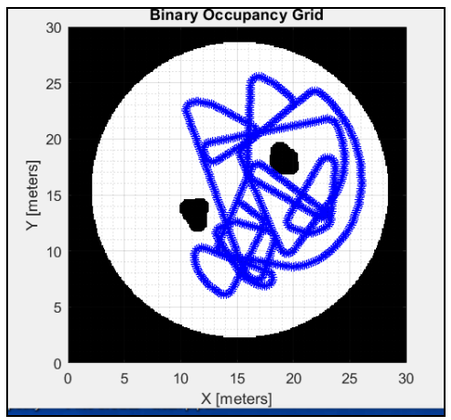
The yellow states are selected by the **switch** structure plus a little logic code, all the states feed into the Avoid Obstacle state, which cycles back to the look for whale state over and over.

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

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

```
switch whaleSeen
  case 0        % whale NOT SEEN
      behavoirCycleTime = behavoirCycleTime+1; % increment behavoir alternating clock
      if (behavoirCycleTime < legTime)         % run GoStright for 10 cycles
         [tugX,tugY,vStraight,wStraight, robotPoses] = THINKGoStraight(robotPoses,sampleTime,diffDriveTug,...
             controlIndex,vRef,wRef);
         vTurn = [0 0 0 0 0 0 0];          % if going straight surpress all turning propellor commands
         wTurn = [0 0 0 0 0 0 0];          % if going straight surpress all turning rudder commands
      else                                 % run TurnLeft for 10 cycles
         [tugX,tugY,vTurn,wTurn, robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveTug,...
             controlIndex,vRef,wRef);
         vStraight = [0 0 0 0 0 0 0];         % if turning surpress all straight propellor commands
         wStraight = [0 0 0 0 0 0 0];         % if turning surpress all straight rudder commands
      end
      if (behavoirCycleTime > legTime +5)
          behavoirCycleTime = 0;               % reset alternating behavior timer
          legTime = random('Normal', 20, 40);   % choose a random transit leg time
      end
```

The **switch** statement is driven by **whaleSeen**. If none are, the following code will oscillate between two separate Tug behaviors, going straight and turning left. The length of time spent in each behavior is controlled by a random number generator so that the Tug will get better coverage of the full pool by doing long and short legs around it. If this leg time were fixed, Tug would limit the cycle into a fixed polygon. Instead it will do a pretty decent random search of the full pool:



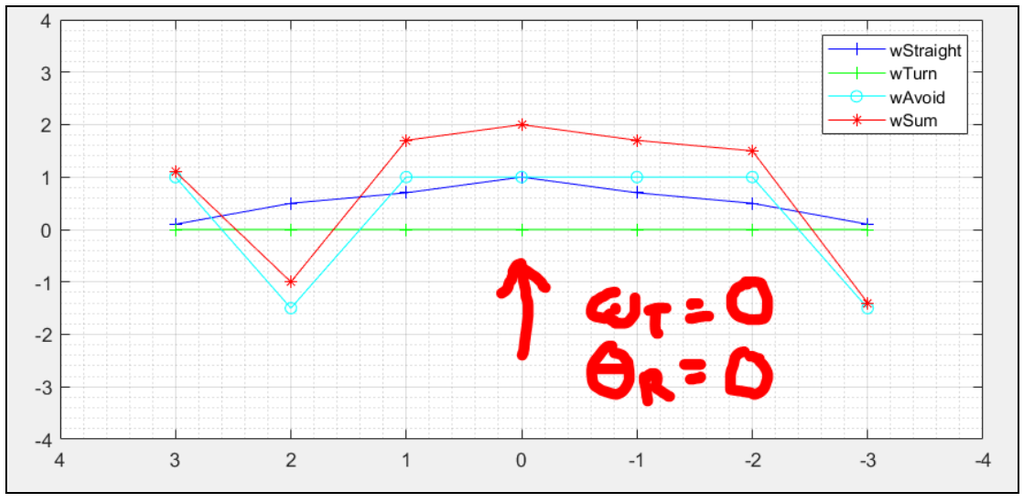
Let's look at the two internal behavior functions in more detail:

Think Functions (store all Think related local functions here)

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

        % Replace pure pursuit controller with fixed velocities
        % [vRef,wRef] = ppControl(poses(:,loopIndex));
        vTug = vRef;    % Tug forward velocity is 2 m/sec
        wTug = wRef;    % Rudder centered, no rotational velocity

        % Perform forward discrete integration step
        vel = derivative(diffDriveTug, robotPoses(:,controlIndex), [vTug wTug]);
        robotPoses(:,controlIndex+1) = robotPoses(:,controlIndex) + vel*sampleTime;

        % Update rover location and pose
        tugX = robotPoses(1,controlIndex+1);
        tugY = 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 Tugs 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 tugs 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:

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c



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 Tugs 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.

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 [tugX,tugY,vTurn,wTurn, robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveTug, controlIndex, vRef, wRef)
% THINKTurnLeft behavior locks rudder at a fixed rudder position and
% steers Tug in a tight left turn. Calculate derivative of robot motion
% based on control commands.
% inputs are  rover poses, loop sampleTime, loop index
% outputs are tugX, tugY, robot poses (position of robot)
% Frank Olin 2021  Rev A

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

        % Perform forward discrete integration step
        vel = derivative(diffDriveTug, robotPoses(:,controlIndex), [vTug wTug]);
        robotPoses(:,controlIndex+1) = robotPoses(:,controlIndex) + vel*sampleTime;

        % Update rover location and pose
        tugX = robotPoses(1,controlIndex+1);
        tugY = robotPoses(2,controlIndex+1);
        vTurn = [1.8 2 1.8 1.7 0 0 0];    % create polled desired linear velocity
        wTurn = [1.1 1 0.9 0.8 0 0 0];     % create pooled 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 subsumption architecture at work.

```
behavoirCycleTime = behavoirCycleTime+1; % increment behavoir alternating clock
if (behavoirCycleTime < legTime)         % run GoStright for 10 cycles
   [tugX,tugY,vStraight,wStraight, robotPoses] = THINKGoStraight(robotPoses,sampleTime,diffDriveTug,...
       controlIndex,vRef,wRef);
   vTurn = [0 0 0 0 0 0 0];          % if going straight surpress all turning propellor commands
   wTurn = [0 0 0 0 0 0 0];          % if going straight surpress all turning rudder commands
else                                 % run TurnLeft for 10 cycles
   [tugX,tugY,vTurn,wTurn, robotPoses] = THINKTurnLeft(robotPoses,sampleTime, diffDriveTug,...
       controlIndex,vRef,wRef);
   vStraight = [0 0 0 0 0 0 0];         % if turning surpress all straight propellor commands
   wStraight = [0 0 0 0 0 0 0];         % if turning surpress all straight rudder commands
end
```

In **case 1** when the whale 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.

```
142         case 1   % whale SEEN
143             if (followClock < 10)
144                 followClock = followClock+1; % increment whale follwing time
145                 % place student whale following behavior function here
146             else
147                 break;                      % stop loop mission complete
148             end
149         otherwise
150             % assume whale not seen
```

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

```
151         end
152         % Run Obstacle Avoidance behavior based on Sharp IR range data
153         [vAvoid,wAvoid] = THINKAvoid (ranges, angles);
154
155         % Arbitrate behavior outputs
156         [vRef,wRef]=THINKTugArbiter(vStraight,wStraight,vTurn,wTurn,vAvoid,wAvoid,brainWaves);
157
158         % Check to see if Tug is hitting a physical obstacle
159         crash = checkOccupancy(MapOfPoolInflated,[tugX tugY]);
160           if (crash == 1)
161              disp('Tug crashed, all stop!');
162              break;
163           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 Tug's current position, with it, each time the loop runs, we can tell if the Tug has hit an obstacle. When it does, the **break** statement jumps out of 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 brainway that will let the Tug go in any direction unhindered, except in the direction of a nearby obstacle. Please create the following code for it:

```
270     function [vAvoid,wAvoid] = THINKAvoid (ranges, angles);
271     % THINKAvoid behavior takes in the ranges from the Tugs Sharp Ir array and
272     % converst them into a set of polled linear and rotational velocity
273     % designed to avoid crashing into percieved obstacles
274     % sharpIRPod.Range = [0.1,10];              sets minimum and maximum range of sensor
275     % sharpIRPod.HorizontalAngle =[-pi/2, pi/2];  sets max min detection angle
276     % sharpIRPod.HorizontalAngleResolution = 0.628318; sets sharp pod resoultion to 6 sensors
277     % Frank Olin 2021  Rev A
278
279     starboardSensors = ranges(1:3);                 % Sharp Irs pointing right
280     portSensors = ranges(4:6);                      % Sharp IRs pointing left
281     middleSensor = (ranges(3)+ranges(4))/2;         % make up a virtual middle sensor
282     s1 = vertcat(starboardSensors,middleSensor);    % add virtual middle to left array
283     sensorArray = vertcat(s1,portSensors);          % add right array
284     TF = isnan(sensorArray);                        % find any Nan values
285     sensorArray(TF)= 30;                            % replace NaN values
286         for iScanTransform =1:7
287             if(sensorArray(iScanTransform) < 7)     % if range is shorter than N vote to
288             sensorArray(iScanTransform)= -1.5;        % not turn in this sensor direction
289             else
290             sensorArray(iScanTransform)= 1;         % else can turn this way
291             end
292         end
293     wAvoid = sensorArray';                  % flip vertical ranges to horizontal wAvoid
294     wAvoid = flip(wAvoid);                  % flip array to match tp down boat coordinate system
295     vAvoid =[0 0 0 0 0 0 0];          % set better slow down velocity policy in your student arbiter
296     end
```

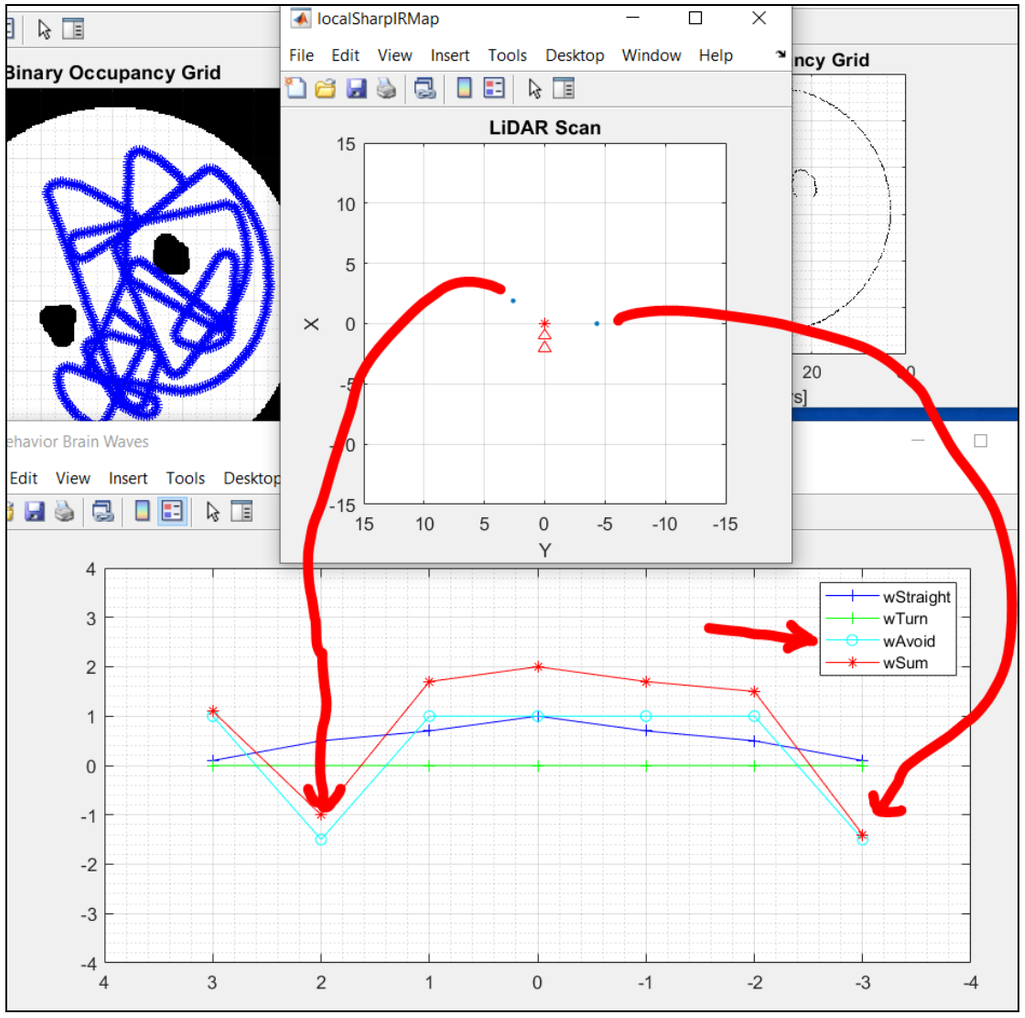
The first part of the code pads the 6 sharp IRs out to 7 by creating a virtual center IR that averages the ranges of the right and left front one. This gives the array and then the brainwave it generates symmetry around the Tugs main axis. Straight ahead is then zero wRef.

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 Tug 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).

The final part flips the command array so that it will match a top down view of the tug to make it easier for human operators to figure out what is going on.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

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 tug's brain, its behavior arbiter. Please create the code below.

```
function [vRef,wRef]=THINKTugArbiter(vStraight,wStraight,vTurn,wTurn,vAvoid,wAvoid, brainWaves)
% THINKTugArbiter takes in the desired outputs of the Robot Tugs
% 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 waveform
    [wMaxValue, wMaxIndex] = max(wSumOfWaves);    % Find peak of summed waveform

    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 tug 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 maximums 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 better version of this, perhaps arbitrating speed as well.

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

Moving on to the ACT section, it's mostly unchanged

```
163         end
164
165     % ACT--------------------------------------------------------------------------
166         % command robot actuators
167         ACT(tugX, tugY, vRef, wRef, rudderServo, propServo, testPool);
168
169      controlIndex = controlIndex + 1;   %increment control loop index
170      waitfor(r);                        % wait for loop cycle to complete
171
172  end
```

With the additional work of creating this fully fleshed out **ACT** function

Act Functions (store all Act related local functions here)

```
332  function ACT(tugX, tugY,vRef, wRef, rudderServo, propServo, fig_testTrack)
333  % ACT Increment the robot pose based on the derivative.
334  % inputs are curretn X and Y position of rover and figure to plot rover in.
335  % outputs are no  outputs
336  % Frank Olin 2021  Rev A
337         %Plot Tug location in pool
338         figure(fig_testTrack)
339         hold on
340         plot(tugX,tugY, 'b*');              % plot robot position in oval
341         hold off
342
343         % Drive Tug hardware serco motors
344         rudderAngle = wRef*15;    % convert angular rate wRef to rudder degrees
345         propSpeed = vRef;         % vref (-2 to 2 m/sec)
346         ok = TugServoControl(rudderServo, propServo, rudderAngle, propSpeed);
347
348  end
```

This new ACT function draws the Tug on the test pool plot and also uses a Fun-Robo supplied function (download from Canvas) called **TugServoControl** that will drive your Tugs rudder and propellor in real-life in a hardware-in-the loop simulation of the Tug actually motoring around the LPB pool. This is a THINK lab, so we want you to focus on that part of your Tug and so supply this function. But don't fret, you either learned about RC servo control on a Raspberry pi 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

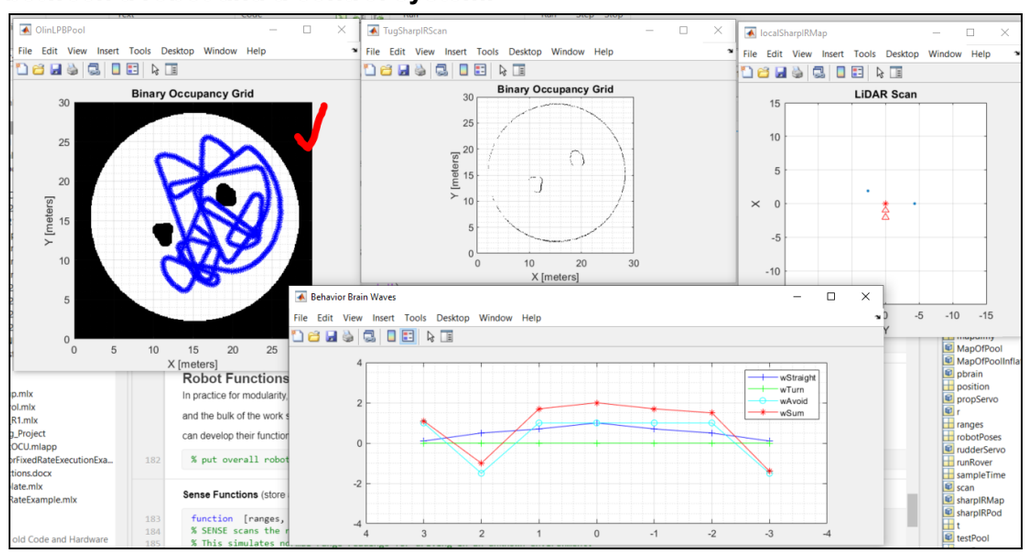
communication paths. This keeps systems form jamming when you want to run again.

```
174  % Stop program and clean up the connection to Raspberry Pi
175  % when no longer needed
176    writePosition(rudderServo, 90);      % always end servo at 0.5
177    writePosition(propServo, 90);        % always end servo at 0.5
178    clc
179    disp('Raspberry Pi program has ended');
```

```
Raspberry Pi program has ended
```

```
180    clear robotPi
181    beep              % play sytem sound to let user know program is ended
```

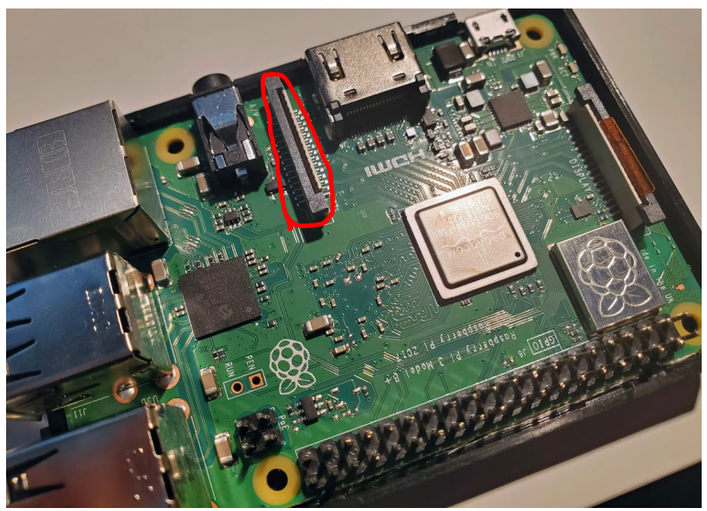
When you run your code, you should get a fully autonomous Tug steaming around the pool in a quasi random pattern, cleanly avoiding hitting walls and icebergs. Please note the robust iceberg handling compared with your previous control system program crashing straight into one. **This is a graphic depiction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



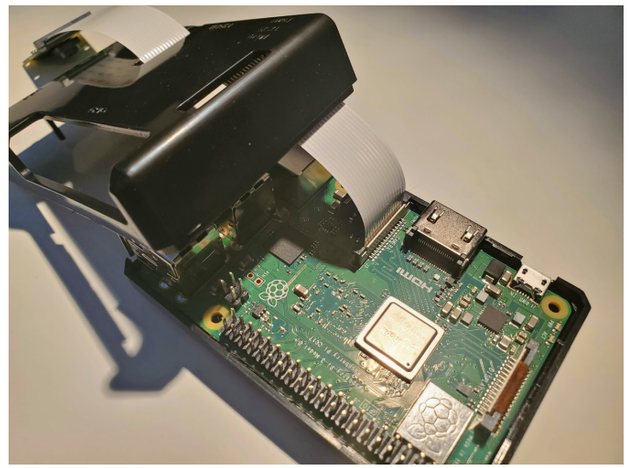
Brief walkthrough of PiCam Whale tracking code

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

To start, you will need to install your Pi-Cam if you haven't already. To do this, lift up the black tab on the port labeled "Camera" on your board (note that the "Display" port looks very similar - be careful not to plug your camera into this one or it won't work!)



Take note of the direction of the ribbon cable. The blue stripe (non-conductive side) should face towards the ports on the board. If you have a case, now would be a good time to feed the ribbon cable through. Insert the ribbon cable, and press down on the black clip to secure it. When installed correctly, the cable should not come out when gently tugged.



Installed Pi-Cam ribbon cable

Now that we have the Pi-Cam installed, 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 narwhal-tracking. A working example of the code referenced in this section can be found in the file **PiCamExample.mlx** on canvas.

To start, if the Raspberry pi is not yet connected, we'll need to connect to it

```
% Clear old vars
clear
% Connect to raspi
rpi = raspi()
```

After connecting to the Raspberry pi, we can use the cameraboard() function to connect to the Pi-Cam. Optionally, we can set the resolution here too.

```
% Note: use webcam function instead if using a USB camera
cam = cameraboard(rpi,'Resolution','1280x720')
```

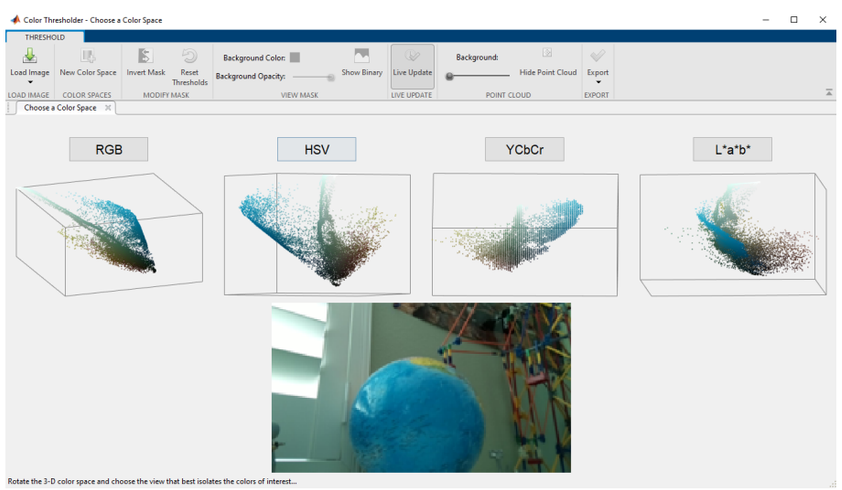
In order to capture an image to work with, we need to call Matlab's snapshot() function. We can use imshow() to display the image inline, which is helpful for debugging.

```
% Take a photo using the camera
img = snapshot(cam)

% Display image
figure, imshow(img)
```

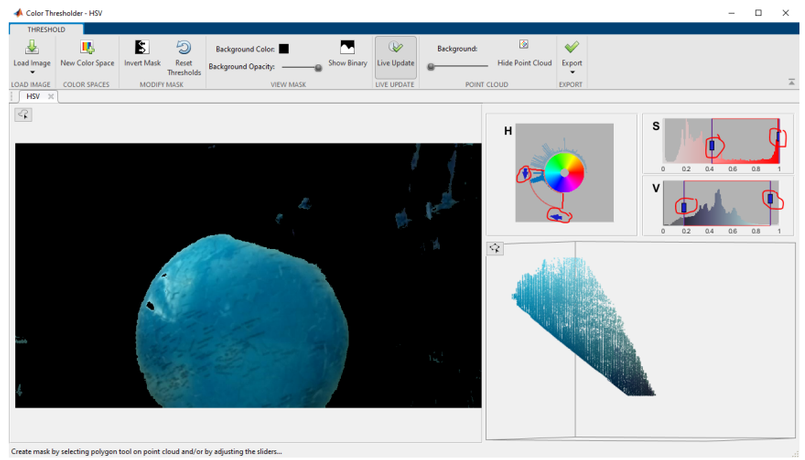
In order to select a colored target to track, hold the target in front of the camera and call the snapshot() function to update the img variable. Once you have a photo you're happy with, pass it into the colorThresholder function. This should open up a separate GUI window.

```
colorThresholder(img)
```

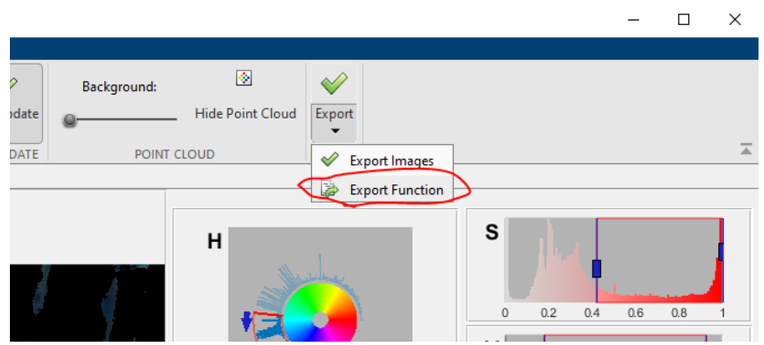
On the first page of the GUI, select a color space of your choice (HSV generally works pretty well, but any of these should work - feel free to experiment with different ones!)



On the next page, adjust the sliders until only the target color remains visible (it's ok if it's not perfect, but the better the sliders are tuned the better your target tracking will be!)



When you're done, click Export>Export Function



A new window will pop up - copy/paste the newly created createMask function at the bottom of the notebook (be sure to replace the existing createMask function). This function will apply the color thresholding settings you just created to any image that is passed into the function.

Now that we have a function to isolate the color of the target we want to track, let's write our whale tracking function! The code to do this is as follows:

```
function [targetX, targetY] = findWhale(frame, displayBool)
    % Use our auto-generated function to extract regions of the image that
    % are the color we want
    [BW, maskedRGBImage] = createMask(frame);

    % Create labels for all regions
    [L, num] = bwlabel(BW, 8);

    % Compute size of each continuous region, find largest region
    count_pixels_per_obj = sum(bsxfun(@eq,L(:),1:num));

    % Create new mask that only contains largest region
    [~,ind] = max(count_pixels_per_obj);
    biggest_blob = (L==ind);

    % Compute center of this region
    s = regionprops(biggest_blob,'centroid');
    centroids = cat(1,s.Centroid);

    % Assign center of largest region to our output variables
    targetX = centroids(1);
    targetY = centroids(2);

    % Display the results
    if displayBool
        figure,
        % Display original image
        subplot(311),imshow(frame)
        % Display color-masked image
        subplot(312),imshow(BW)
        % Display the largest blob with the center marked
        subplot(313),imshow(biggest_blob)
        hold on
        plot(centroids(:,1),centroids(:,2),'b*')
        hold off

        fprintf(['X-pixel value: %.4f \n'], targetX)
        fprintf(['Y-pixel value: %.4f \n'], targetY)
    end
end
```

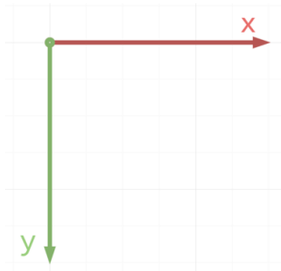
At a high level, the steps that this function walks through are

1. Use the createMask function from the previous step to create a masked image where only the selected color range is visible
2. For each "connected" region, create a unique label and compute the size of the region
3. Create a new mask that only contains the largest region
4. Compute the "center of mass" of the largest region
5. Return the x and y pixel values of the center of the region

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

```
% Choose whether to display frames for debugging
displayFrames = true;
[whaleX, whaleY] = findWhale(img, displayFrames);
```

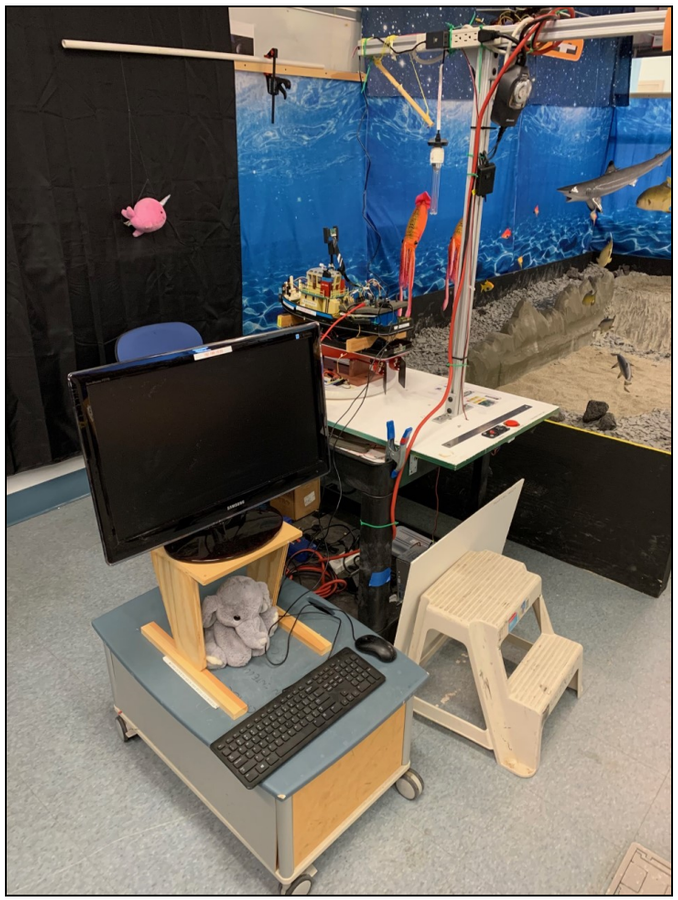
Aside: When using this function, it's important to note the coordinate system of the targetX and targetY values. In Matlab and most other image-processing software, the origin is located at the upper left of the image, with the x-axis pointing right and the y-axis pointing down. Keep this in mind if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



Matlab image coordinate system

Developing your Tug Hardware-in-the-Loop code

Adding in the actual physical tug hardware is fairly straightforward, you just need to control three motors; the RC motor directly linked to the twin rudders, the paired DC motors that spin the propellers and the turntable motor that makes the tug spin on the test cart.



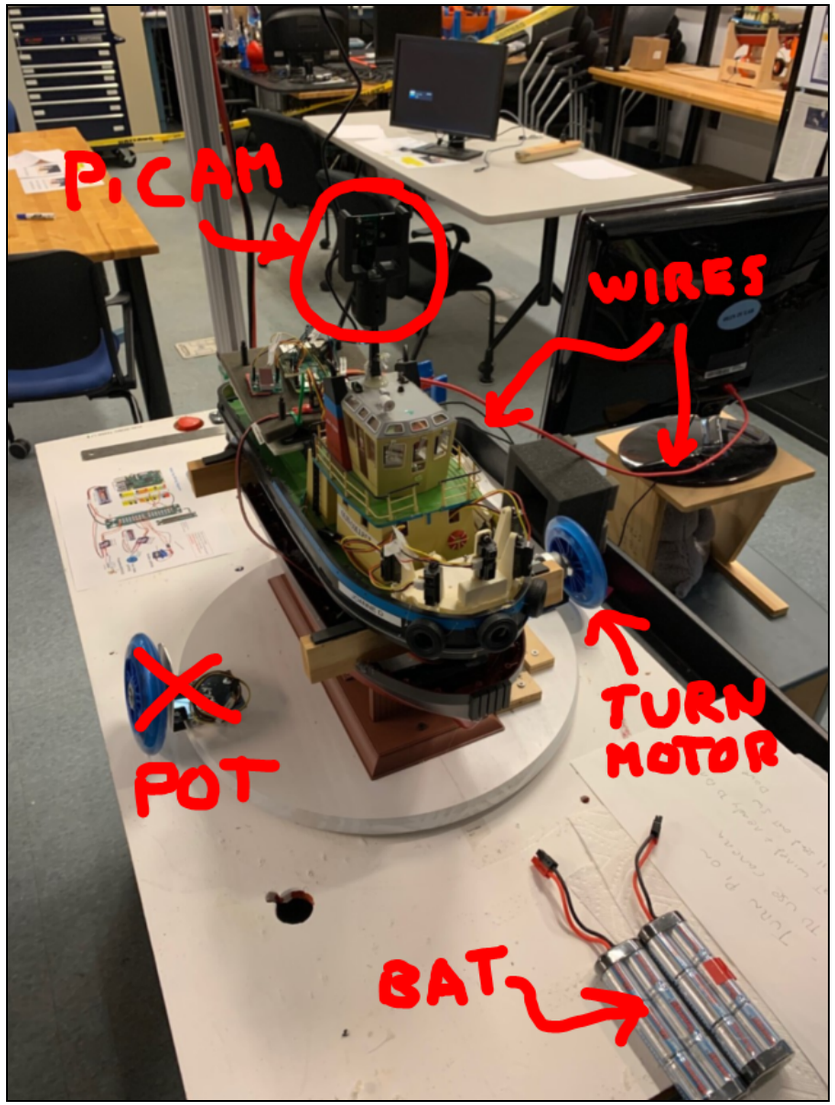
Please note: When turning the lab cart hardware on, a monitor, keyboard mouse and Raspberry Pi power are all plugged in together. YOU MUST UNPLUG Pi-cables IF YOU ARE GOING TO DRIVE TURNTABLE!

For convenience getting Pi up and developing software on it, the tugs Raspberry Pi is connected to a lot of land based hardware. Once Pi is up and running, you can collect camera images and run rudder and props all ok with this shore wiring in place, but if you drop the turntable drive wheel into contact with the cart, the tug will spin, tear out all wires and you will crap-trash the whole lab. Please don't do that!

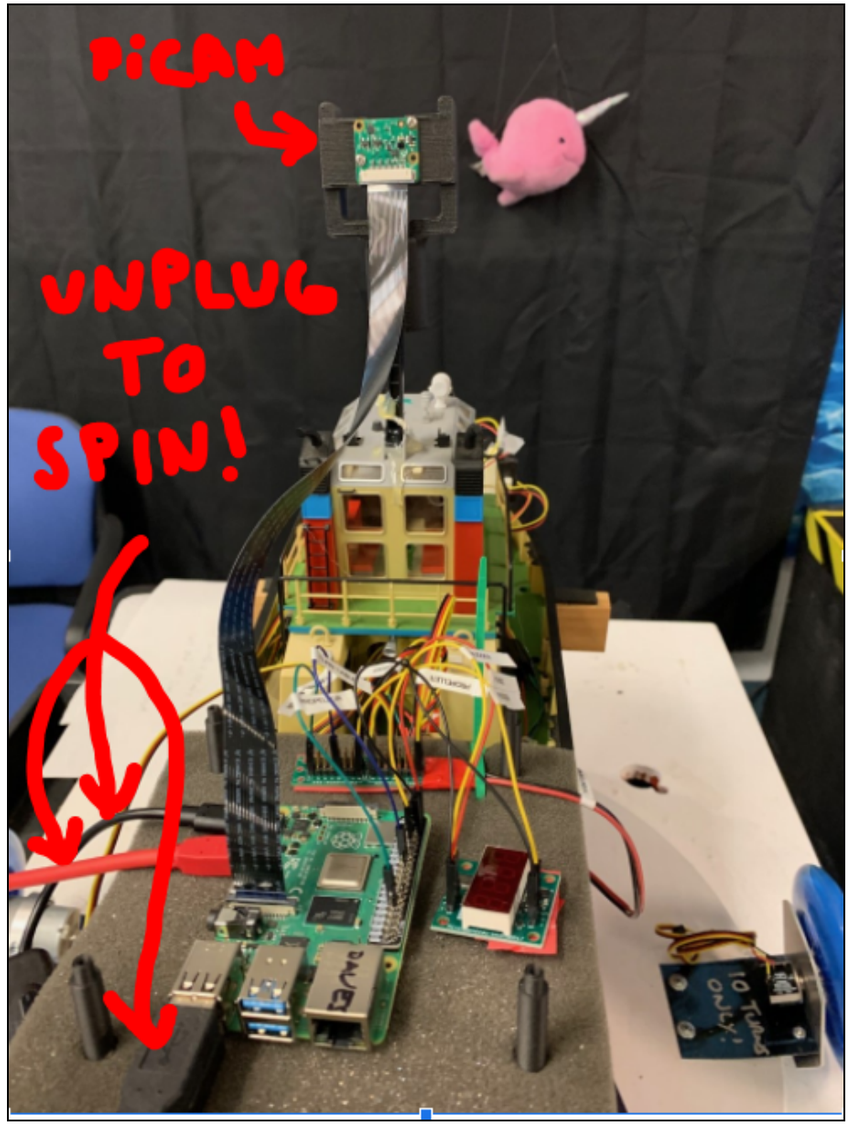
When you have most of the code in place, you can shut down, unplug everything and restart the test cart with the fresh on-tug batteries powering the Pi, drop the turntable drive wheel and then you can do whirling tug dances to your heart's content.

Please see course instructor or Ninja to be trained to carefully plug and unplug (with power off) the Pi and its shore support hardware, how to plug in and run it headlessly off the tug's 5 VDC battery pack and how to drop drive wheel:

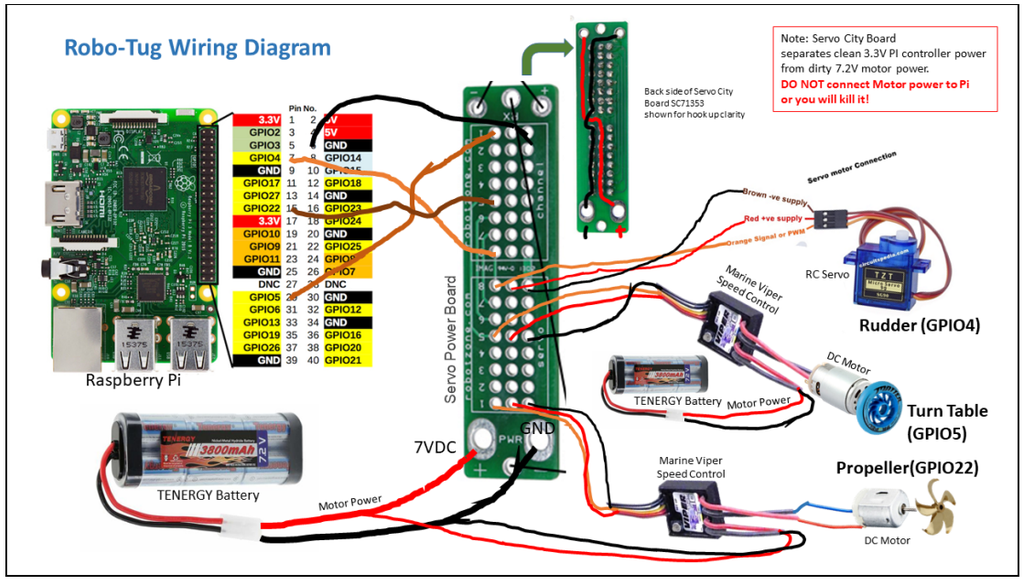
Taking a quick walk through the tug hardware-in-loop there is a forward facing Pi-Camera (see next section) up on top of its flying bridge mast:



The tug is oriented so that it can clearly see the purple narwhal you are tracking:



The Tug is controlled by a single Raspberry Pi 4 wired as shown in diagram below:

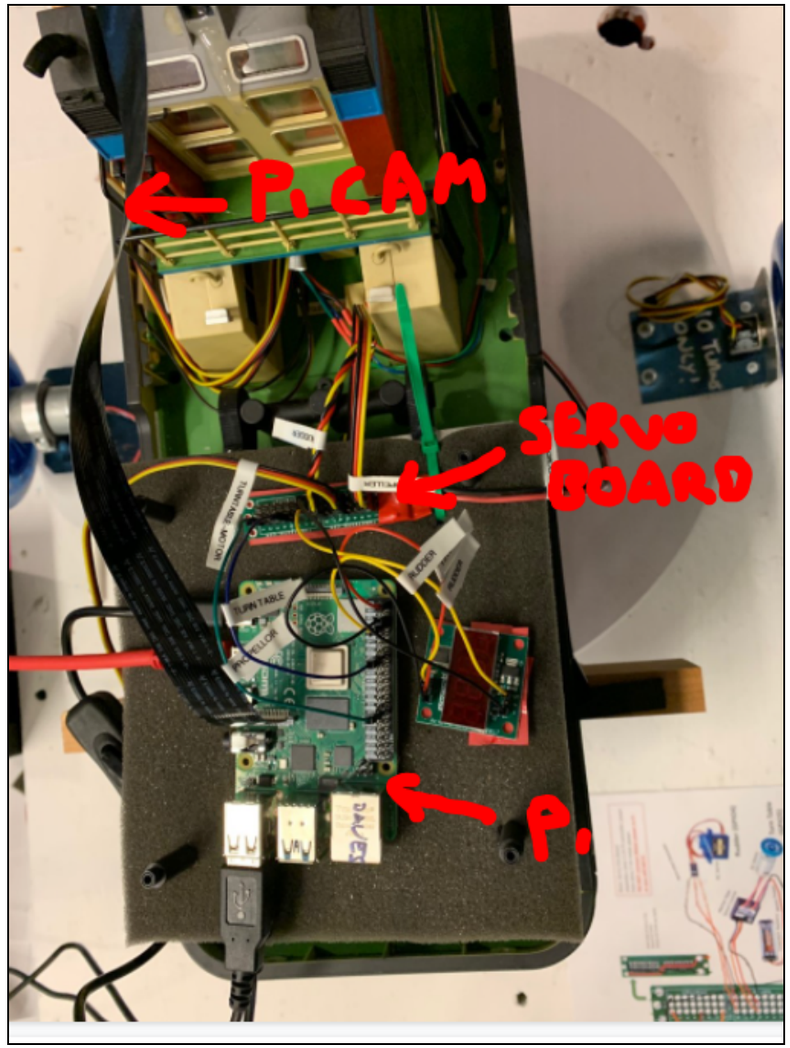


A full size copy of which is downloadable from the canvas website as well as printed out and attached to the lab cart itself.

The actual control system hardware for the Tug can be shown in the photo to the right. The Raspberry Pi can alternatively be connected to stationary (shore) power, monitor, keyboard and mouse for long time duration programming and then to a Tug based USB battery pack for simulated (revolving) operation.

Just don't don't forget to unplug the wires!

We will use three GPIO pins on the Pi to drive the three motors on the tug and a servocity power bus separation board to connect them:



ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

You will get to learn all about how to write control code for motors in the ACT lab. For now, we will give you the code as a canvas download and you can just cut and paste it into your simulation code in the correct spots to add the tug hardware to your loop. You can download two functions and a small piece of test code for this section from Canvas. Drop in your working directory. The first function **TugRaspPiSetupRevB**

TugServoControl_RevB.mlx | LiveMotorTest.mlx | TugRaspPiSetup_RevB.mlx

This function creates a Raspberry Pi object and configures the Raspberry Pi to work with the Think Labs robot tug boat sensors and servos.

It has no input arguments and returns a set of Pi objects to be used by supporting Sensor and Actuator functions. This function must be run first, before its Robot Tugboat famly functions will run.

D. Barrett Jan 2021 Revision B

```
function [robotPi,rudderServo,propServo, ttServo, blinkLED, cam] = TugRaspPiSetup_RevB()
% PiSetup creates and configures a Raspberry  Pi to be a simple robot
% controller. It requires no inputs and returns an Pi and Servo Pi objects
% D. Barrett 2021 Rev B

% Create a global raspberry PI object so that it can be used in functions
% Create a *raspi* object.
   robotPi = raspi('192.168.16.65');     % This IP address must match your current Pi address!

% There is a user LED on Raspberry Pi hardware that you can turn on and
% off. Execute the following command at the MATLAB prompt to turn the LED
% off and then turn it on again.
%
   blinkLED = robotPi.AvailableLEDs{1};

% Note: use the cameraboard webcam function instead if using a USB camera
%    cam = cameraboard(robotPi,'Resolution','1280x720');
% use Cam if using Tugs Picam
    cam = 0;

% create servo objects driving PWM GPIO pin 4, pin 5, pin 22
% MinPulseDuration: 1120 (microseconds)  center 1520 (microseconds)
% MaxPulseDuration: 1920 (microseconds)
% servo(mypi, pinNumber, Name,Value) creates a servo motor object
% with additional options specified by one or more Name, Value pair arg
  rudderServo = servo(robotPi, 4, 'MaxPulseDuration', 1925*10^-6, ...
      'MinPulseDuration', 1120*10^-6);
  propServo = servo(robotPi, 5, 'MaxPulseDuration', 1925*10^-6, ...
      'MinPulseDuration', 1120*10^-6);
  ttServo = servo(robotPi, 22, 'MaxPulseDuration', 1925*10^-6, ...
      'MinPulseDuration', 1120*10^-6);

% R-C Mode expects to start up with joystick centered
% In the RasPI MATLAB function servo position is 0-180 so 90 (1520ms) is centered
  writePosition(rudderServo, 90); % always start servo-command at center
  writePosition(propServo, 90); % always start servo-command at center
  writePosition(ttServo, 90); % always start servo-command at center
  pause(5.0);                 % wait for Pi to send stable pwm
end
```

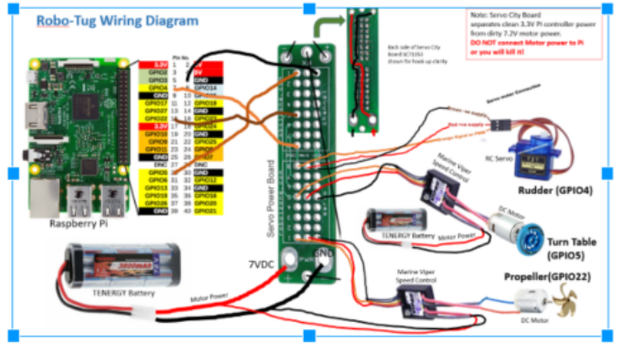
It just extends your original Raspberry Pi setup code to specifically connect to the Pi on the Tug (located at 192.168.16.65 see line 8)
and then creates three servo objects: **rudderServo**, **propServo** and **ttServo** (turntable servo) and then commands them to a center position which will stop both the propellor and turntable from moving and center the rudder with respect to the hull. You will need to replace your Simulated RaspPiSetup function with this one.

The next **TugServoControl_RevB** drives the rudder, turntable and propellers

TugServoControl_RevB.mlx | LiveMotorTest.mlx | TugRaspPiSetup_RevB.mlx

This function controls two RC servomotors on Robot Tug, the inputs are rudderAngle:the rudder in open loop position mode -45 to 45 degress and propSpeed: the propellor speed in open loop speed control from full rerverse to full forward,as well as the rudderServo and propServo Pi objects

Hook Pi pin IO4 to rudder Motor, IO5 to propellor Motor, IO22 to turntable motor, Ground Pi to Servo Ground.



Scale rudder Input command -45 to 45 to be 0-180 for rudder (90 is centered) Scale turntable to turn when rudder is not zero

Scale propellor input command full reverse (-2ft/sec) to full forward (2ft/sec) to be 0-180 for propellor (90 is stop)

Need to run TugRaspPiSetup function first to create a Raspberry-Pi and servo objects before using this function

D. Barrett Jan 2021 Revision B

```
function commandComplete = TugServoControl_RevB(rudderServo, propServo, ttServo, rudderAngle, propSpeed)
    %add code that hold rudder angle to 45 and speed to 2
    scaledRudderAngle = (rudderAngle+45)*2.0;
        scaledRudderAngle=min(max(scaledRudderAngle,-40),40);   % keep rudder in safe bounds
    writePosition(rudderServo, scaledRudderAngle);
    scaledPropSpeed = (propSpeed*45)+90;
    writePosition(propServo, scaledPropSpeed);
    scaledTurnTableSpeed = (rudderAngle+45)*2.0;
       scaledTurnTableSpeed=min(max(scaledTurnTableSpeed,-30),30);  % keep table speed in safe bounds
    writePosition(ttServo, scaledTurnTableSpeed);
    commandComplete = 1;
end
```

ENGR3390: Fundamentals of Robotics Tutorial (THINK - behaviors, arbitration, OCU) 2021c

And finally **LiveMotorTest**, to let you drive motors independently from your code

Live Editor - C:\Users\dbarrett\Dropbox\C_Olin Courses 2017\ENGR3390 Robotics\20XX steady state course material\course code\Thin

TugServoControl_RevB.mlx | LiveMotorTest.mlx | TugRaspPiSetup_RevB.mlx | +

LiveMotorTest.mlx runs Tugs three actuators for motor testing

This livescript lets operator run tugs three motors independantly and together

for hardware and code testing purposes

This code needs two functions placed in teh same directory that it runs in

TugRaspPiSetup_RevB.mlx and **TugServoControl_RevB.mlx**

D. Barrett Feb 25, 2021 Rev A

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

Set up robot control system (code that runs once)

```
testVariable = 1 ;      % put your run once code here
% Configure your Raspberry Pi to support Tug
[robotPi,rudderServo,propServo, ttServo, blinkLED, cam] =TugRaspPiSetup_RevB;
robotPi                 % print out pi specs to check connection
```

```
robotPi =
  raspi with properties:

         DeviceAddress: '192.168.16.65'
                  Port: 18734
             BoardName: 'Raspberry Pi 2 Model B'
        AvailableLEDs: {'led0'}
  AvailableDigitalPins: [4,5,6,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27]
  AvailableSPIChannels: {'CE0','CE1'}
     AvailableI2CBuses: {'i2c-1'}
      AvailableWebcams: {'mmal service 16.1 (platform:bcm2835-v412)'}
          I2CBusSpeed: 100000

  Supported peripherals
```

```
disp('****WARNING TUG MOTORS ACTIVE*******')
```

```
****WARNING TUG MOTORS ACTIVE*******
```

```
beep                % warning tug motors starting
beep                % warning tug motors starting
beep                % warning tug motors starting
```

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

```
controlFlag = 1;        % create a loop control
while (controlFlag == 1)
```

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

```
controlFlag = 1;        % create a loop control
while (controlFlag == 1)
    CONTROL_LOOP_INPUT()
    commandTugSpeed = input('Enter Tug Speed range -2 to 2 ft/s: ');
    commandRudder = input(' Enter RudderAngle range -45 to 45 deg: ');
    commandComplete = TugServoControl_RevB(rudderServo, propServo, ttServo, commandRudder, commandTugSp
    controlFlag = input('Run another test?  1 yes 0 stop:  ');
end
```

```
robot running, cntrl-c to stop
commandComplete = 1
robot running, cntrl-c to stop
commandComplete = 1
robot running, cntrl-c to stop
commandComplete = 1
```

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 Raspberry Pi
% when no longer needed
  writePosition(rudderServo, 90);      % always end servo at 0.5
  writePosition(propServo, 90);      % always end servo at 0.5
  writePosition(ttServo, 90);      % always end servo at 0.5
  clc
  disp('Raspberry Pi program has ended');
```

```
Raspberry Pi program has ended
```

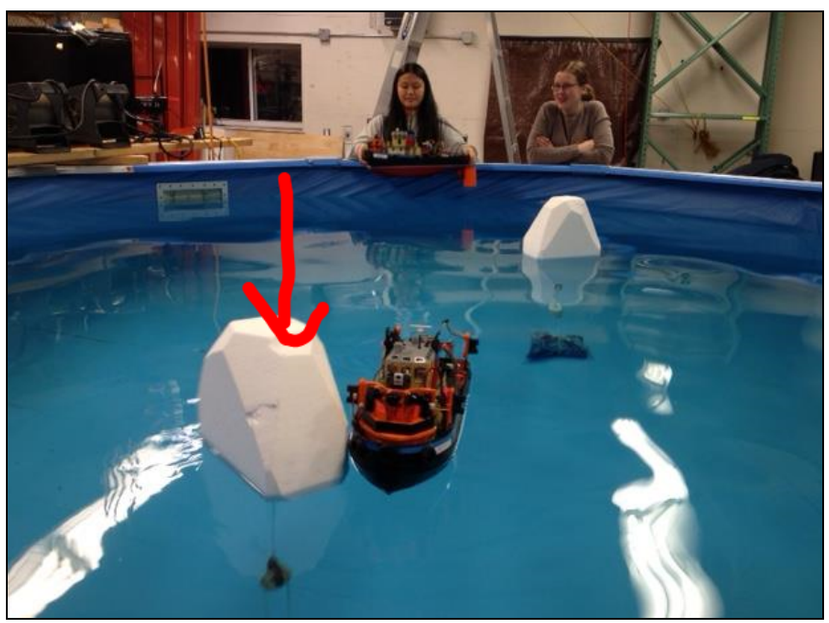
```
  clear robotPi
  beep              % play sytem sound to let user know program is ended
```

Run this program step by step, get an appreciation for driving tug motors and then go back and add live motor control to your Tug simulation code to make a full hardware in the loop simulation.

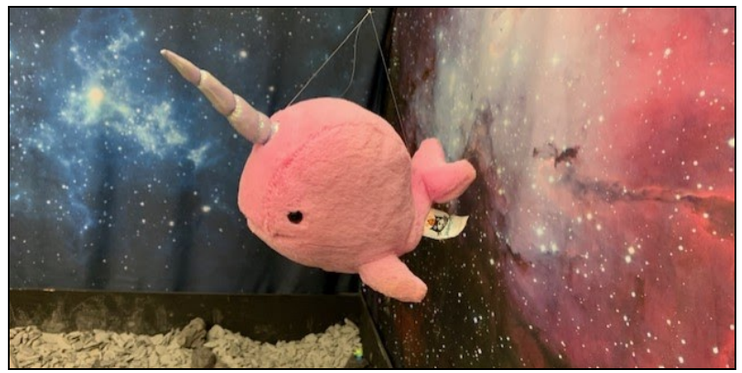
Developing your Whale Tracking Tug Demo

From the previous program, you can see how just three simple behaviors; go-straight, turn and avoid, let the Tug robustly and in the presence of dynamic obstacles, explore the pool looking for a Narwhal to photocapture. Your team's mission is to use that code as a starting template. Using the given Matlab functions for both SENSE (simulated Sharp IRs and real Pi-Cam) and ACT, controlling the Tugs actual rudder, turntable and propeller servos, your team should:

1) first create a new FSM behavior diagram to clearly describe your tugs behaviors. It's ok to go a bit wild here. Be daring! Maybe you should spiral out from the center of the tank? 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 tank walls and icebergs,



as well as be on the lookout for the purple Narwhal !



If and when you find narwhal (being waved about by Ninja on a pole), 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 arbiter to improve performance (and probably should do all given functions over 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 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!

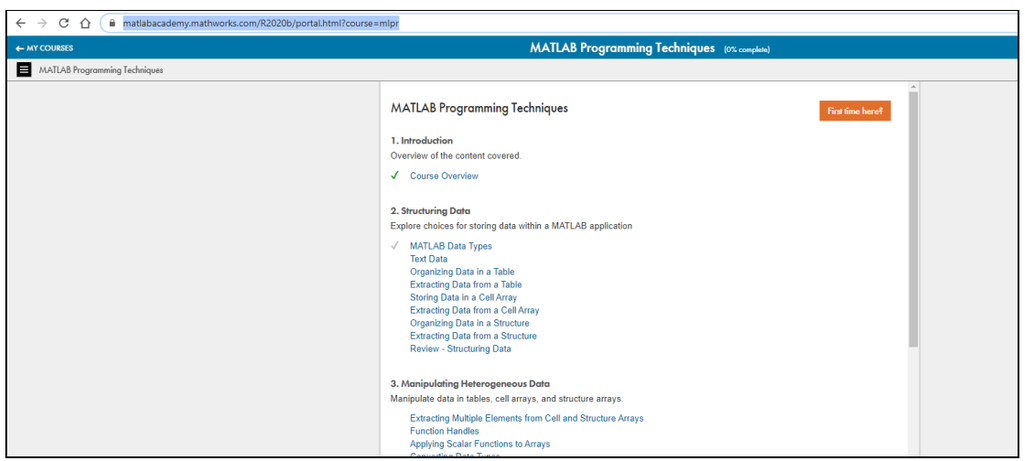
Two week demo: Please write this code, debug it, and be prepared to demo both the original control system and a full whale following Robot Tug.

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.

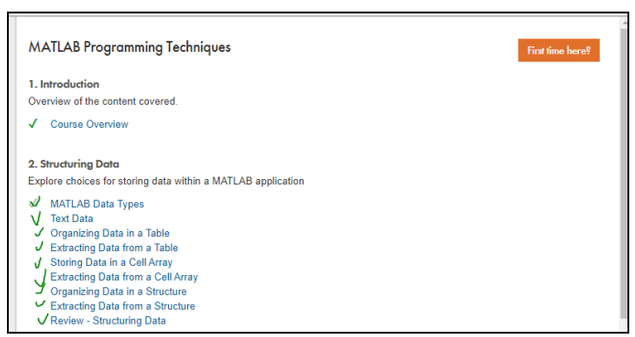
Please see a course instructor or Ninjas for MATLAB help as needed.

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