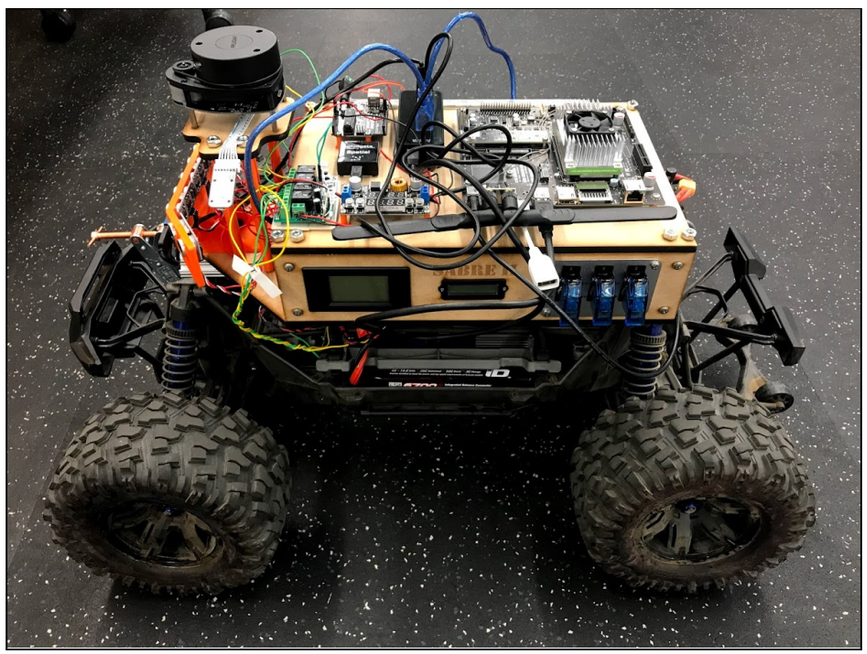
ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

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 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.

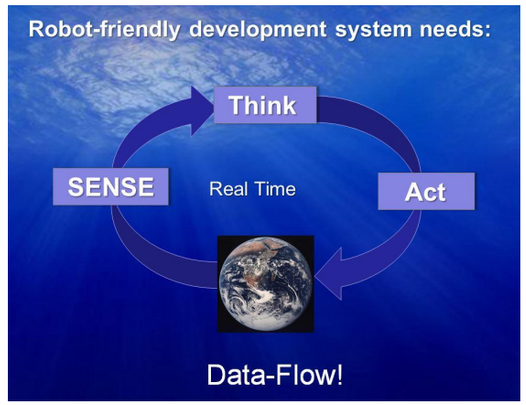


Figure 1 : Sense-Think-Act Control Loop

SENSE: involves adding sensors to a robot to answer three primary questions; “Where am I?”, “What is around me?” and “How am I?”. The art of robot perception is to develop your skills in choosing the appropriate sensor, learning to electrically hook it up into your robots existing circuitry, collecting data from it, filtering that data, metroligizing that data (converting it from volts to engineering units) and finally processing the data (i.e. perception) to answer one of the three fundamental questions presented above.

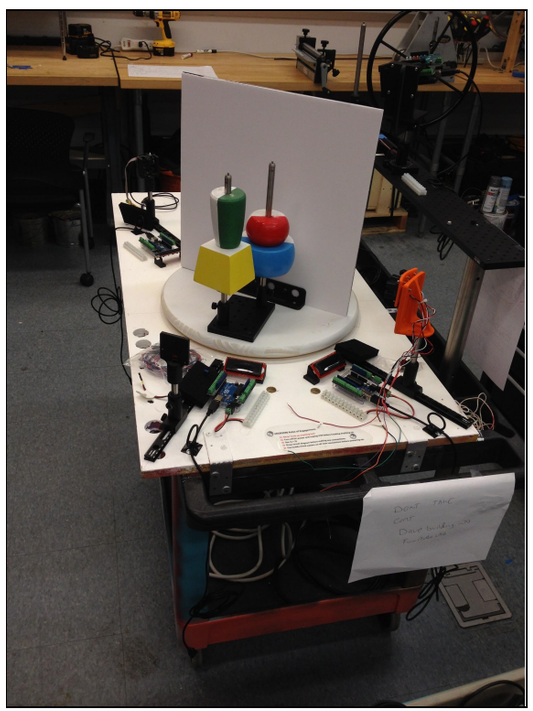
In this lab we will look at and give you hands-on experience with all of these goals on a carefully curated set of simple robotic sensors connected to your Arduino and Raspberry-Pi robot controller, as well as, in parallel, advancing your general engineering MATLAB programming skills. Please see the last section of this lab write up for your MATLAB homework in preparation for doing the hands-on work in the lab.

Please skip Raspberry-Pi sections of this lab in 2022.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Sense Lab

This lab contains 4 sensor test stations, each housing a commonly used perception sensor, all stations surrounding a rotating sensor target ring on which a set of different colored and shaped targets can be mounted. Perception sensors generally have two main system functions; **Find the target** or **Find the hole**. The first lets you move to or follow an object, the second lets you avoid objects so that your robot can pass through them.



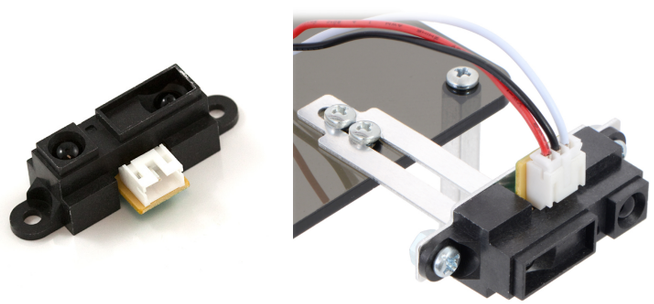
Sense Lab 1 Hardware

Sharp IR Range Sensors

The widely available Sharp Infrared Range sensors are the robot engineers low-cost, accurate distance sensor of choice. They are easy to hook up, easy to write MATLAB code for and easy to mount. This IR sensor is more economical than sonar rangefinders, yet it provides much better performance than many other range sensor alternatives.

Interfacing to most microcontrollers is straightforward: the single analog output can be connected to an analog-to-digital converter for taking distance measurements, or the output can be connected to a comparator for threshold detection. The detection range of this version is approximately 10 cm to 80 cm (4″ to 32″), depending on the model you are using.

The Sharp 0A41SK uses a 3-pin JST PH connector. These cables have 3-pin JST connectors on one end and are available with pre-crimped male pins, pre-crimped female pins, and with unterminated wires on the other end. It is also possible to solder three wires to the sensor where the connector pins are mounted (see the lower picture to the right). But please use connectors for long term viability. When looking at the back, the three connections from left to right are power (red), ground (black), and the output signal (white).



Sharp IR Range Sensor

Operation

The SHARP 2Y0A21 proximity sensor measures distance by shining a beam of infrared light and uses a phototransistor to measure the intensity of the light that bounces back.

Technical Specifications:

-Operating voltage: 4.5 V to 5.5 V
-Average current consumption: 30 mA (note: this sensor draws current in large, short bursts, and the manufacturer recommends putting a 10 µF capacitor or larger across power and ground close to the sensor to stabilize the power supply line)
-Distance measuring range: 4 cm to 30 cm
-Output type: analog voltage
-Output voltage differential over distance range: 1.9 V (typical)
-Update period: 38 ± 10 ms
-Size: 44.5 mm × 18.9 mm × 13.5 mm (1.75″ × 0.75″ × 0.53″)
-Weight: 3.5 g (0.12 oz)

Linearizing the Output

The relationship between the sensor's output voltage and the inverse of the measured distance is approximately linear over the sensor's usable range. The Sharp data sheet (also in Sense file on Canvas):

(https://www.pololu.com/file/0J713/GP2Y0A41SK0F.pdf)

contains a plot of analog output voltage as a function of the inverse of distance to a reflective object. You can use this plot to convert the sensor output voltage to an approximate distance by constructing a best-fit line or polygonal that relates the inverse of the output voltage (V) to distance (cm). In its simplest form, the linearizing equation can be that the distance to the reflective object is approximately equal to a constant scale factor (~27 V*cm) divided by the sensor's output voltage. Adding a constant distance offset and modifying the scale factor can improve the fit of this line.

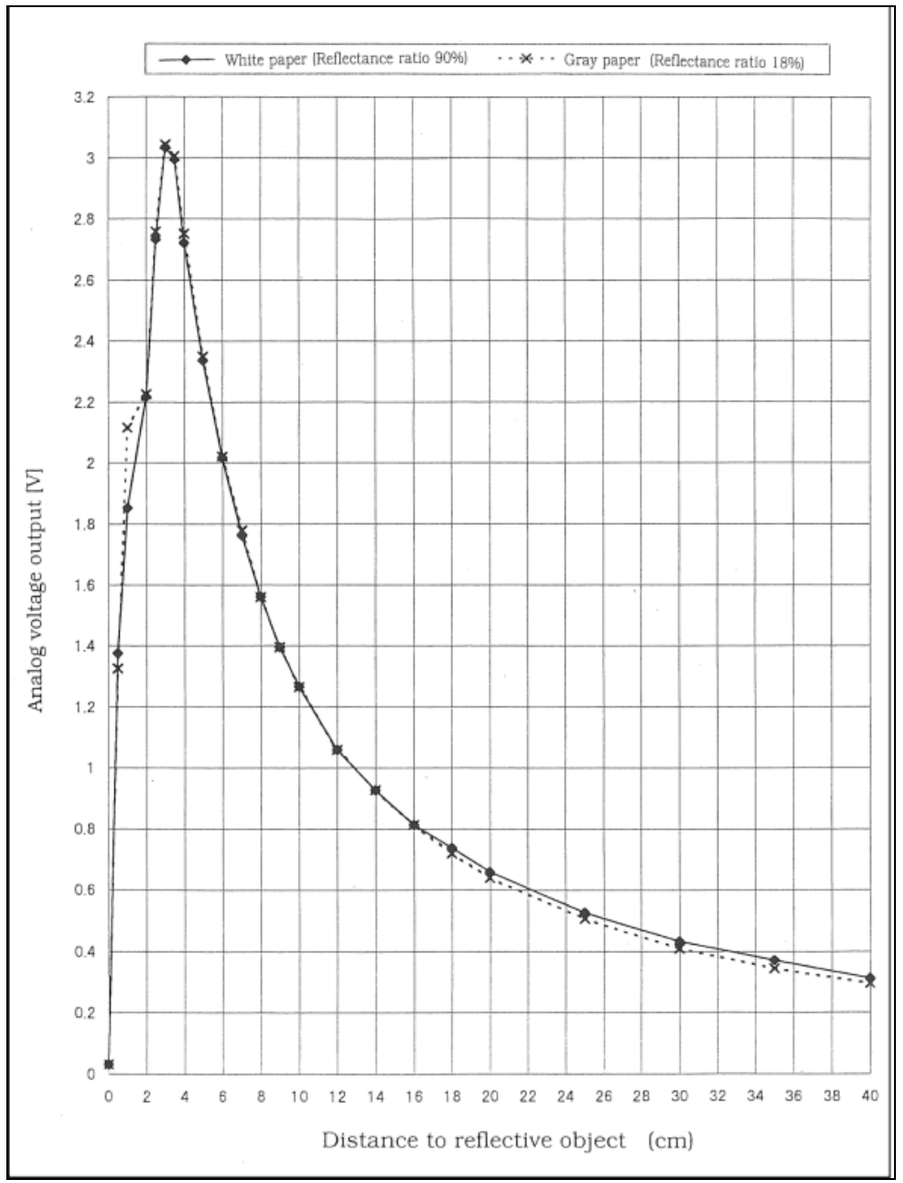
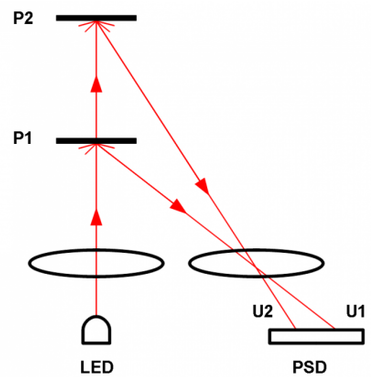


Figure 3: Analog Voltage vs. Distance

Theory

For measuring the distance to an object the Sharp Sensors are optical sensors using a triangulation measuring method. The sensors made by Sharp" have an IR LED equipped with a lens, which emits a narrow light beam. After reflecting from the object, the beam will be directed through the second lens on a position-sensitive photodetector (PSD). The conductivity of this PSD depends on the position where the beam falls. The conductivity is converted to voltage and if the voltage is digitalized by using analogue-digital converter, the distance can be calculated. The route of beams reflecting from different distance is presented below:



Sharp Sensor Optics

The output of distance sensors by "Sharp" is inversely proportional, this means that when the distance is growing the output is decreasing (decreasing is gradually slowing). Exact graph of the relation between distance and output is usually on the data-sheet of the sensor. All sensors have their specific measuring range where the measured results are creditable and this range depends on the type of the sensor. Maximum distance measured is restricted by two aspects: the amount of reflected light decreasing and inability of the PSD to register the small changes of the location of the reflected ray. When measuring objects which are too far, the output remains approximately the same as it is when measuring the objects at the maximum distance. Minimum distance is restricted due to the peculiarity of Sharp sensors, meaning the output starts to decrease (again) sharply at a close distance (depending on the model 4-20 cm). This means that one value of the output corresponds to two values of distance. This problem can be avoided by either mounting the sensor back that distance from the edge of the robot or using persistent tracking to ensure that the object is not too close to the sensor.

Linearizing output of sensor

Please see :

https://acroname.com/articles/linearizing-sharp-ranger-data

For a detailed description of how to linearize the Sharp's range output.

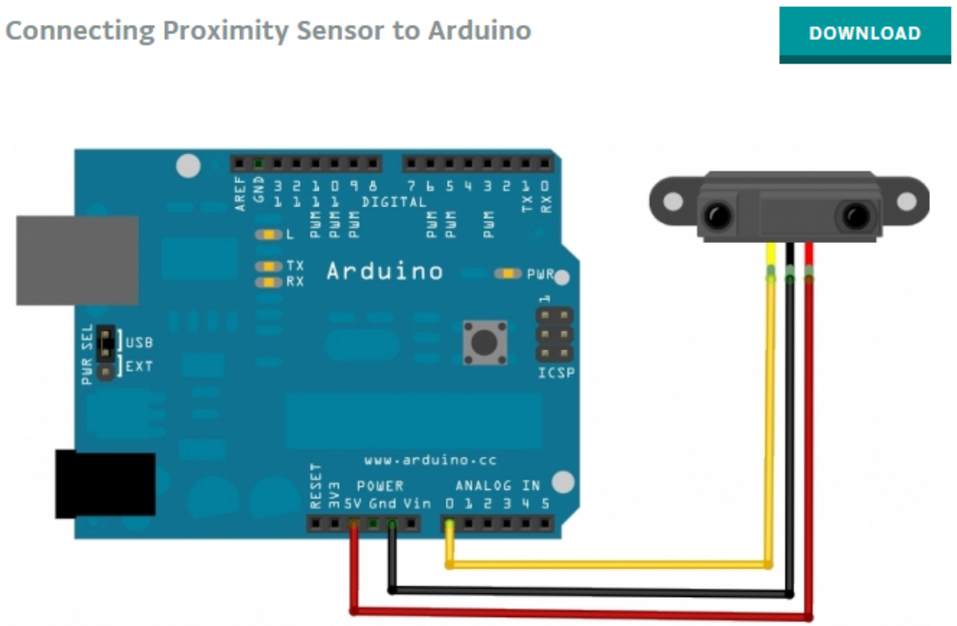
.

Alternative Sharp distance sensors

There is a variety of Sharp distance sensors to choose from, including the shorter-range (4 – 30 cm) GP2Y0A41SK0F and longer-range (20 – 150 cm) GP2Y0A02YK0F. These analog distance sensors have similar packages and identical pin-outs, making it easy to swap one version for another should your application requirements change. There is also the newer Sharp GP2Y0A60SZ analog distance sensor (10 – 150 cm), which outperforms the other analog Sharp distance sensors in almost all respects, offering a low minimum detection distance, high maximum detection distance, wide 3 V output voltage differential, high 60 Hz sampling rate, operation down to 2.7 V, and optional enable control, all in a smaller package.

Figure 5: Family of Sharp IR Range Sensors

Connecting Sharp IR Sensors to an Arduino is very straight forward.



You will simply connect the 5VDC (red) and GND (black) wires to the appropriate power and ground screw terminals on your Arduino and then connect the analog output (yellow or white) to one of your analog input pins. You can connect multiple Sharps at once, to build a multi-degree range perception sensor.

The Sharp IR test station mounted on Sense 1 lab looks like this:

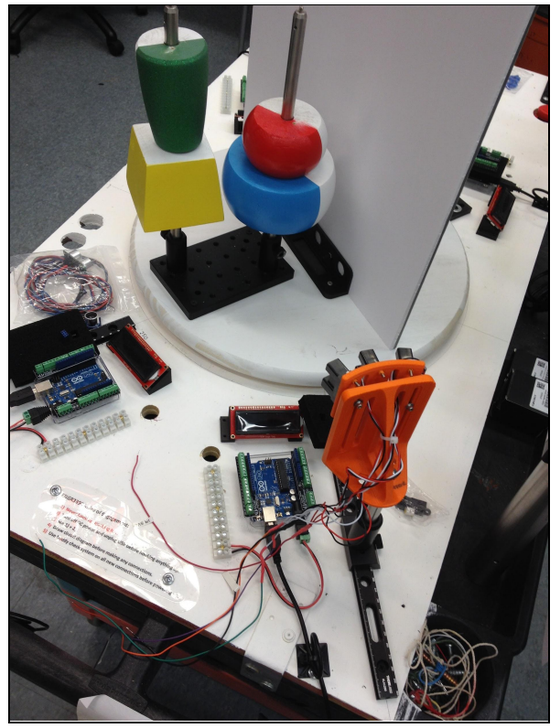
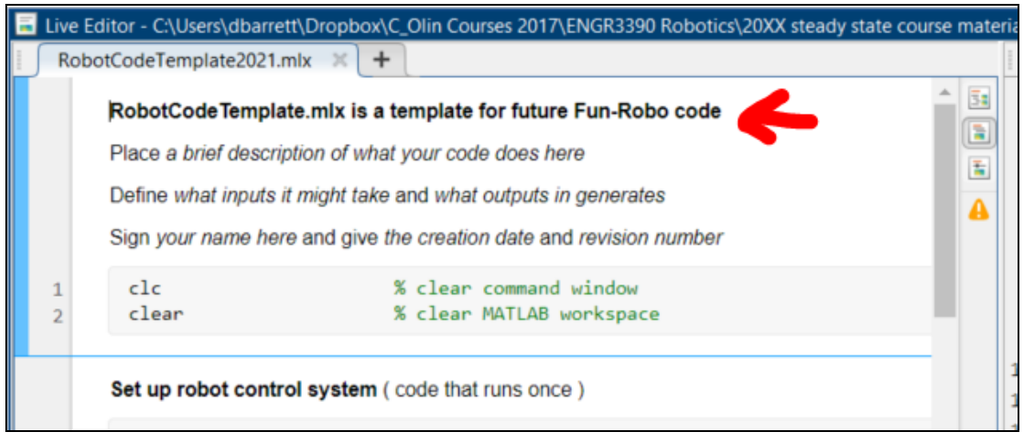


Figure: Sharp IR test station

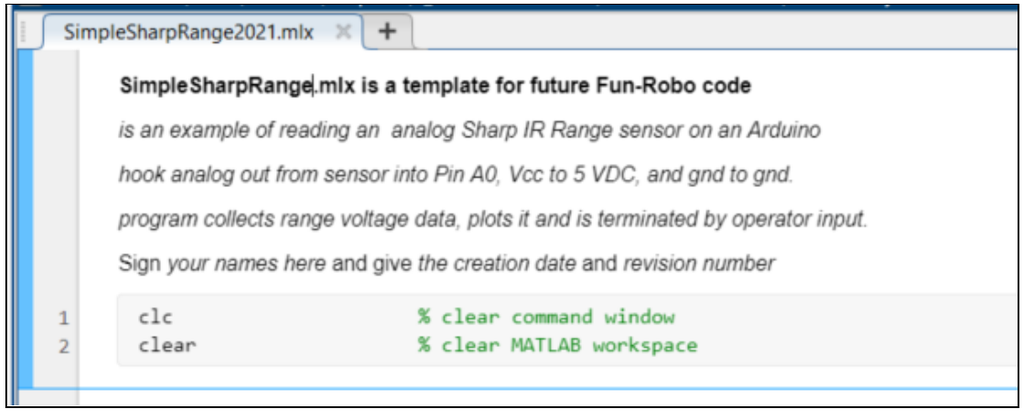
Getting the raw data into the Arduino is relatively easy, but you will quickly find that you are going to have to carefully calibrate each type of Sharp Range Sensor, you might need a different calibration curve for different color targets and you are going to need to find a way to weave all of that new data into a single function that you can add to your Matlab sensor code to find ranges. Once you have generated that SharpIR SENSE data collection code, you will need to linearize it, you may need to filter it and you will then need to use that filtered, linearized Sharp range data to perform two separate functions; **Find the target** (any of the targets on the rotating target ring) and **Find the hole** (find the spaces between targets). Finally you will

need to add a small piece of OCU user output to your Matlab Robot Code program that will print out “Target” or “Hole” as the turntable passes in front of the sensor array.

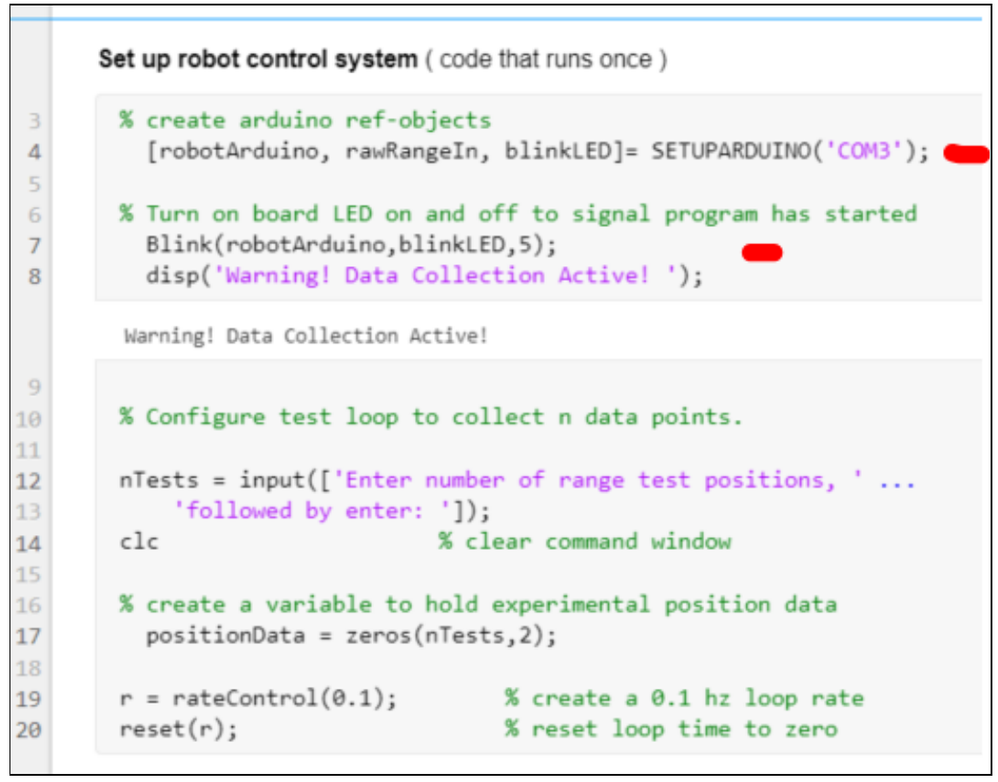
Let's take on each of these tasks sequentially. First open up a copy of your Matlab Robot Template:



And save as a new copy (in your lab team **Matlab Drive** folder) a Livescript called:



1). **To calibrate** an individual Sharp range sensor, start by removing all targets from the target ring, rotate Sharp head until the center Sharp is perpendicular to white target ring center plane. Your Sharp sensor is mounted on a precision linear slide that will let you move it in and out at fixed annotated intervals from the target plane. You will collect about 10 range data points, from white backdrop, from near to far at fixed intervals. Repeat this process for the black center plane, at exactly the same intervals. Plot both sets of data on a single graph. Save it for your lab report. What do you see? To help you get started, please add the following statements to your template:



ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

This **code that runs once** uses two functions, the first configures yoru Arduino, please enter it as shown:

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

```
function [robotArduino, rawRangeIn, blinkLED]= SETUPARDUINO(COMPORT)
% SETUPARDUINO creates and configures an arduino to be a simple robot
% controller. It requires which COM port your Arduino is attached to
% as its input and returns an Arduino object called robotArduino
% D. Barrett 2021 Rev A

% Create a global arduino object so that it can be used in functions
% a = arduino('setToYourComNumber','Uno','Libraries','Servo');
   robotArduino = arduino(COMPORT,'Uno','Libraries','Servo');

% configure pin 13 as a digitial-out LED
   blinkLED = 'D13';
   configurePin(robotArduino,blinkLED,'DigitalOutput');

% Configure A0 pin as an analog input
  rawRangeIn = 'A0';
  configurePin(robotArduino,rawRangeIn,'AnalogInput')

end
```

And it is simply configuring your Arduino as to what libraries to preload. It sets up pin **D13** to be your robot blinky light and it configures pin **A0** to be an analog input. By taking care of all of the plumbing to set up your Arduino, down here in a function, you both keep your main body of code clear and clean as well as make it very modular. If you choose to replace your Arduino with another controller, like a Raspberry Pi, you just need to redo this function, not the whole body of the code. This will be a real time savings throughout this course.

Next add a new **Blink** function. All robots need a blinky light to give the operator a visual clue something is happening. If you write one blinky light early on in your career, you can reuse it over and over for all sorts of indicator functions downstream. Please code as shown below.

```
function [] = Blink(a,LED, n)
  % Blink toggles Arduino a LED on and off to indicate program running
  % input n is number of blinks
  % no output is returned
  % dbarrett 1/14/20
    for bIndex = 1:n
      writeDigitalPin(a, LED, 0);
      pause(0.2);
      writeDigitalPin(a, LED, 1);
      pause(0.2);
    end
  end
```

Moving on to the main control loop:

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

```
controlFlag = 1;                % create a loop control
while (controlFlag < nTests+1)  % loop till ntests data captured

    % collect data from robot sensors
    rangeData = SENSE(robotArduino, rawRangeIn)
    THINK();                    % compute what robot should do next
    ACT();                      % command robot actuators

    % store experimental commanded versus actual data
    positionData(controlFlag,1)= input('Enter actual distance (cm): ');
    gtest= input('move Sharp to new range, type G, then hit ENTER ','s');
    positionData(controlFlag,2)= rangeData;
    Blink(robotArduino,blinkLED,1);
    waitfor(r);                 % wait for loop cycle to complete
    controlFlag = controlFlag+1; % increment loop
end
```

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Please enter the code as shown and then let us walk through it step by step.
All of the actual data collection takes place in the **SENSE** function:

Sense Functions (store all Sense related local functions here)

```
82  function rangeData = SENSE(robotArduino, rawRangeIn)
83      disp('Sense');
84      rangeData = readVoltage(robotArduino, rawRangeIn);
85  end
```

This function is straightforward, it reads the voltage on the analog pin specified by **rawRangeIn** and returns it as **rangeData**. You could wonder why we'd put this down in a function and not just up in the main code. It's a good question. In practice, we always will write highly modular code, both for clarity and to allow for future expansion. While this is currently a one line function, it's just returning an extremely non-linear voltage for range. What you will want to return is far more likely to be a linearized, metroligized range distance in centimeters. When you add all of that linearization and calibration code to this function it will be a quarter of a page long, and there is no way you want that upstairs in the control loop!

Next the **THINK** and **ACT** functions are just empty placeholders for future expandability:

Think Functions (store all Think related local functions here)

```
86  function THINK()
87     % null function, not much thinking to do here.
88  end
```

Act Functions (store all Act related local functions here)

```
89  function ACT()
90  % null function, not much acting to do here.
91  end
```

Moving on, the remaining procedural code asks the operator to measure actual range to target and enter it. Prompts them to move the sensor to a new range and then repeats the full process for **nTest** cycles.

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

```
21  controlFlag = 1;                % create a loop control
22  while (controlFlag < nTests+1)  % loop till ntests data captured
23
24      % collect data from robot sensors
25      rangeData = SENSE(robotArduino, rawRangeIn)
26      THINK();                    % compute what robot should do next
27      ACT();                      % command robot actuators
28
29      % store experimental commanded versus actual data
30      positionData(controlFlag,1)= input('Enter actual distance (cm): ');
31      gtest= input('move Sharp to new range, type G, then hit ENTER ','s');
32      positionData(controlFlag,2)= rangeData;
33      Blink(robotArduino,blinkLED,1);
34      waitfor(r);                 % wait for loop cycle to complete
35      controlFlag = controlFlag+1; % increment loop
36  end
```

The output of the livescript will look something like this:

```
Sense
rangeData = 0.8895

Sense
rangeData = 1.1144

Sense
rangeData = 1.2170

Sense
rangeData = 1.2805

Sense
rangeData = 1.5982

Sense
rangeData = 1.7791

Sense
rangeData = 2.0968

Sense
```

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

After collecting your data, you will want to process and see it, so please add the following section to make a pretty plot:

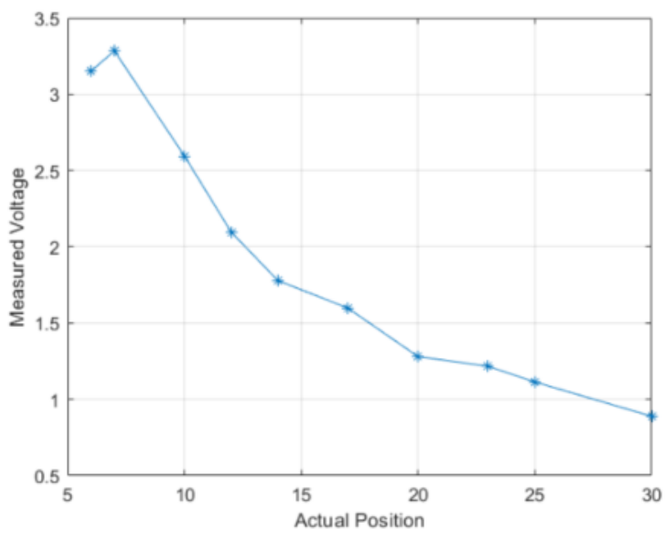
Mission data processing

For many robot applications, you will need to post-process the data collected after the mission. Here we will plot the measured versus actual range positions.

```
37  % Plot and store commanded position data vs. actual position
38    plot(positionData(:,1), positionData(:,2), '-*')
39    xlabel('Actual Position')
40    ylabel ('Measured Voltage')
41    grid
```

Which will generate a nice Sharp voltage versus range plot for your sensor:

```
40    ylabel ('Measured Voltage')
41    grid
```



Wrapping it all up, please add a short piece of cleanup code to shut down your Arduino cleanly. Please note: If you don't shut down embedded controllers like Arduinos and Raspberry Pi's they will run indefinitely. Overnight, for months and years into the future. This can make your life really complicated and drive your teammates crazy, so please shut down cleanly:

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.

```
43    % Stop program and clean up the connection to Arduino
44  % when no longer needed
45
46    clc
47    disp('Arduino program has ended');
```

```
Arduino program has ended
```

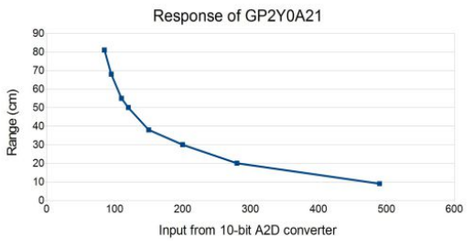
```
48    clear robotArduino
49    beep              % play sytem sound to let user know program is ended
```

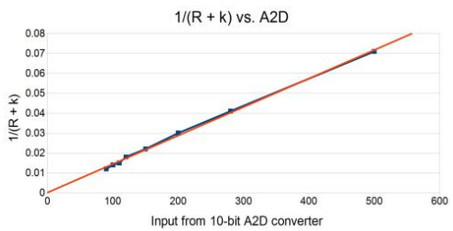
Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

With this code complete, you've done the hard part of sensing, namely getting the raw data in from the environment around the Robot. The next steps are to linearize it and perhaps filter it so that it is clean enough to work with. Then you can calibrate it so that your sense functions produce ranges in centimeters that your future robot brain can work with and not in volts (which it can't !). Please work with your pair programming partner and take on the next steps.

2). **Linearize**. With this raw data and the linearization background algorithm information given to you above, develop a MATLAB function that will convert the raw sensor data, collected in volts, into useful range data in cm (or inches). Write a simple MATLAB function that when called by the main program range0 = readSharp(0), range1 = readSharp(1), etc.); that reads the raw sensor voltage, applies your calibration/linearization function to it and returns range in engineering units, either cms or inches. Hint: You may need to come up with some clever interpolation way to incorporate the difference between a white target and a black one. See Ninjas or instructors for suggestions if you need a bit of help here.

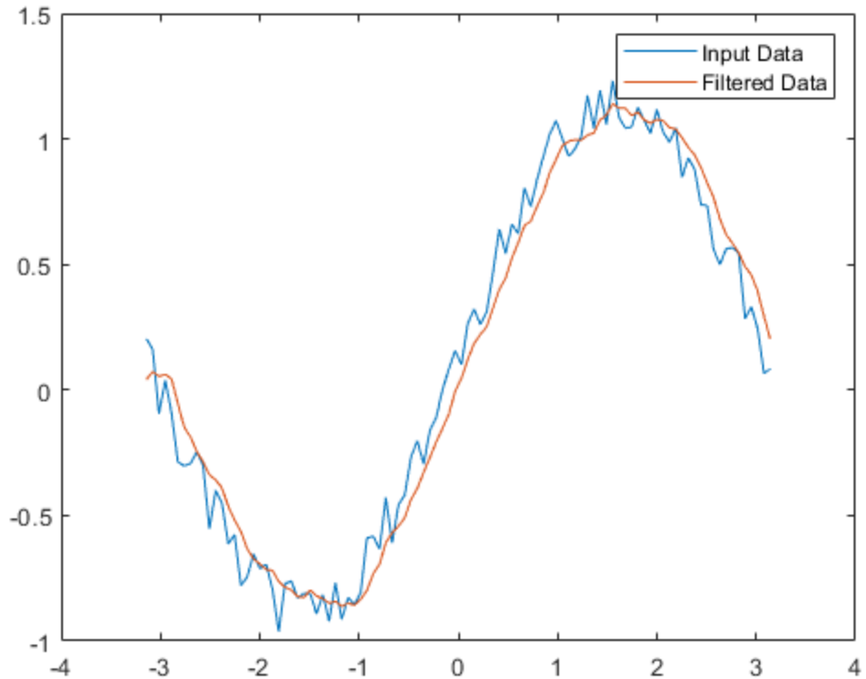




3). **Filtering?** Next, write a MATLAB script to collect 10 range samples at a single distance from white backplane target about one second apart. Record them in a rangeDataOvertime variable matrix along with the sample time of each measurement and plot them. Does the data vary with time? If so, you may need to develop a low-pass software **filter** to provide clean data to the perception code that will consume it. You could add this software filter to your readSharp(n) function as well.

For filter background/help, see:
https://www.megunolink.com/articles/3-methods-filter-noisy-arduino-measurements/

And MATLAB has extensive filtering functions already built in. As a starting point see:
https://www.mathworks.com/help/matlab/ref/filter.html



4). **Check sensor data processing robustness**. Your next step is to determine how your SharpIR functions work on colored and oddly shaped targets. Collect 10 data sets of actual range and measured range data from the red sphere, yellow trapezoid, etc. Save data into a MATLAB variable and plot. In an ideal world the graph of actual distance versus measured distance would be a clean 45 degree line for any color or surface. If it's not, go back and add additional code to your function to deal with any color or surface issues that arise. You may want to alter your function to return both a range and a confidence value. See Ninjas for help here, if you get bogged down.

Having got your Sharp range sensors calibrated, linearized, filtered and solid, you can move on to using the range data from all three sensors mounted on the perception head to take on your labs main perception functions,

5). **Find Target Or Hole**. Using your existing MATLAB code, write a simple sharp IR range sensor script that prints out something like:

"hole...hole...hole...target...hole...hole"

for about ten seconds as the target turntable passes in front of the sensor head. As a starting point, slowly rotate a variety of targets past the sensor head at a set of distances and record the data you collect as they go by. Can you see each target? When do you first see them? Can you tell the difference between them? Extra points here for being able to use the full sensor head to tell which object is passing your sensing station. More extra points for porting this code into a MATLAB APP stand alone application.

Given the final two week demo goal below, modify your existing Robot Control code to do the following demo functionality.

Final Sharp range sensor perception demo: *:* Course Ninjas will place some small collection of given targets on the center target ring and set your sensor head at a midway range from the ring. As the ring slowly rotates by your Sharp range sensor head, your code should find each target, (and for extra points) correctly identify which target it is and give a good range estimate to its centroid, printing all of this back to your OCU laptop terminal in real time. Ninjas will then position the white back plane perpendicular to the sensor headset sensor head at a fixed distance and slowly move obstacles (the existing targets) through the field of view of your sensors by hand. Your code should read sensors, perform perception on the data stream and send either "Open hole" or "Obstacle, range, bearing" to the OCU terminal. Your team gets massive applause if each of all 5 Obstacles are detected.

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

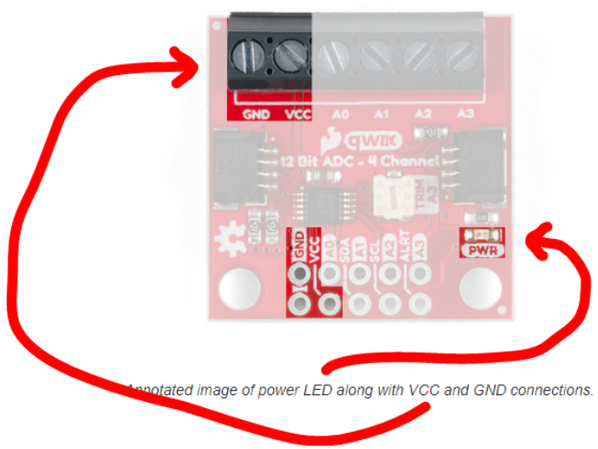
Sharp IR-Range Sensors on Raspberry-Pi (Skip in 2022)

The Raspberry Pi has many awesome talents, but directly reading the output of Analog sensors isn't one of them. The Pi doesn't have analog input ports like an Arduino. To overcome this shortcoming we are going to give you a i2C bus Analog Input device and a pre-written Matlab function to call it. With that support, you can easily do a light edit of your Arduino code to port it to your Raspberry Pi.

First off, please shut power to your Raspberry Pi, attach qwiic pHAT to it, read through this section and then wire in Sharp Ir range sensor as described below:

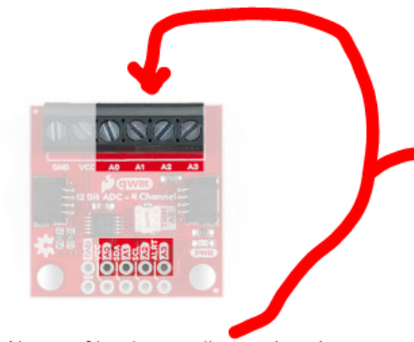
Qwiic 12-Bit ADC Hardware Hookup Guide:

Power; There is a power status LED to help make sure that your Qwiic ADC is getting power. You can power the board either through the polarized Qwiic connector system or the breakout pins (3.3V and GND) provided. This Qwiic system is meant to use 3.3V, be sure that you are NOT using another voltage when using the Qwiic system.

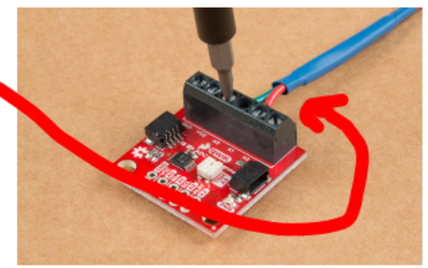


Annotated image of power LED along with VCC and GND connections.

There are four input measurement channels for the ADS1015, labeled AIN0-3, accessible through the screw pin terminals shown below. With the Qwiic system, the absolute minimum and maximum voltage for these inputs is -.3V and 3.6V, respectively.



Annotated image of input connections on board. Click to enlarge.



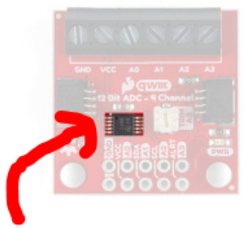
Example of attaching inputs through screw terminals. Click to enlarge.

Depending on the settings for the multiplexer (MUX) in the ADS1015, the analog voltage readings (or conversions on the ADC) will be either for 4 single-ended inputs or 2 differential pairs.

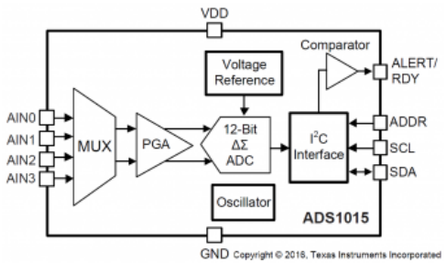
Pin Label	Configuration Options
AIN_0	Single-Ended Differential Inputs: AIN1 or AIN3 w/ Programmable Comparator
AIN_1	Single-Ended Differential Ref. w/ Programmable Comparator Differential Input: AIN3 w/ Programmable Comparator
AIN_2	Single-Ended Differential Input: AIN3 w/ Programmable Comparator
AIN_3	Single-Ended Differential Ref. w/ Programmable Comparator On-board 10kΩ Potentiometer

FUN ROBO

The ADS1015 ADC is a low-power 12-bit analog-to-digital converter (ADC), which includes a built-in integrated voltage reference and oscillator. At the core of its operation, the ADC uses a switched-capacitor input stage and a delta-sigma (ΔΣ) modulator to determine the differential between AINP (positive analog input) and AINN (negative analog input). Once the conversion is completed, the digital output is accessible over the I2C bus from the internal conversion register.



Annotated image of ADS1015 on board. Click to enlarge.



Functional block diagram from datasheet. Click for more details.

The ADS1015 is a powerful tool with multiple configuration settings, set by the Config Register, for the analog voltage readings (or conversions). In the following sections, we will cover the general operation of the ADS1015. For exact details of the various configuration settings, please refer to the manufacturer datasheet. The operational characteristics of the ADS1015 are summarized in the table to right.

Characteristic	Description
Operating Voltage (V_{DD})	2.0V to 5.5V (**Default on Qwiic System: 3.3V**)
Operating Temperature	-40°C to 125°C
Operation Modes	Single-Shot (**Default**), Continuous-Conversion, and Duty Cycling
Analog Inputs	Measurement Type: Single-Ended or Differential Input Voltage Range: **GND** to V_{DD} (*see **Caution** note, below*) Maximum Voltage Measurement: *Smallest of* V_{DD} or **FSR** Full Scale Range (FSR): ±.256V to ±6.114V (**Default: 2.048V**)
Resolution	12-bit (Differential) or 11-bit (Single-Ended) LSB size: 0.125mV - 3mV (**Default: 1 mV** *based on FSR*)
Sample Rate	128 Hz to 3.3 kHz (**Default: 1600SPS**)
Current Consumption (*Typical*)	Operating: 150µA to 200µA Power-Down: 0.5µA to 2µA
I²C Address	**0x48** (**Default**), 0x49, 0x4A, or 0x4B

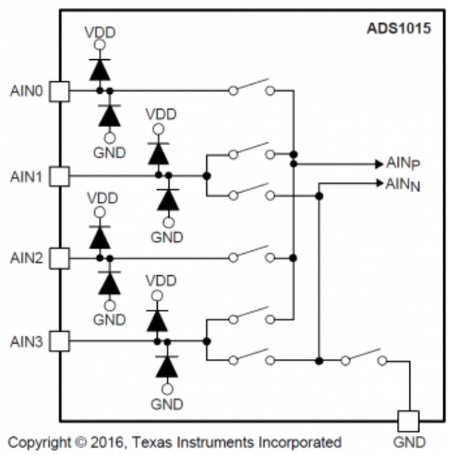
⚡**Caution:** The absolute, analog input voltage range is **(GND - .3V) to (V_{DD} + .3V)**; anything slightly higher/lower may damage the ADC chip. If the voltages on the input pins can potentially violate these conditions, as specified by the datasheet, use external Schottky diodes and series resistors to limit the input current to safe values.

The ADS1015 has 2 different conversion modes: single-shot and continuous-conversion with the ability to support duty cycling. Through these modes, the ADS1015 is able to optimize its performance between low power consumption and high data rates. We will use Single shot

Single-Shot

By default, the ADS1015 operates in single-shot mode. In single-shot mode, the ADC only powers up for ~25µs to convert and store the analog voltage measurement in the conversion register before powering down. The ADS1015 only powers up again for data retrieval. The power consumption in this configuration is the lowest, but it is dependent on the frequency at which data is converted and read.

There are four input measurement channels for the ADS1015, labeled AIN0-3. The input multiplexer controls which of those channels operates as the AINP (positive analog input) and AINN (negative analog input) to the ADC. Depending on the input MUX configuration, the voltage measurements will be either on single-ended inputs or as differential pairs.



Operational digram of multiplexer from datasheet.

Although it is listed as a 12-bit ADC, the ADS1015 operates as an 11-bit ADC when used with single-ended (individual) inputs. The 12th bit only comes into play in differential mode, as a sign (+ or -) indicator for the digital output. This allows the digital output to represent the full positive and negative range of the FSR (see table and figure below).

Table 3. Input Signal Versus Ideal Output Code

INPUT SIGNAL $V_{IN} = (V_{AINP} - V_{AINN})$	IDEAL OUTPUT CODE(1)(1)
≥ +FS $(2^{11} - 1)/2^{11}$	7FF0h
$+FS/2^{11}$	0010h
0	0000h
$-FS/2^{11}$	FFF0h
≤ –FS	8000h

(1) Excludes the effects of noise, INL, offset, and gain errors.

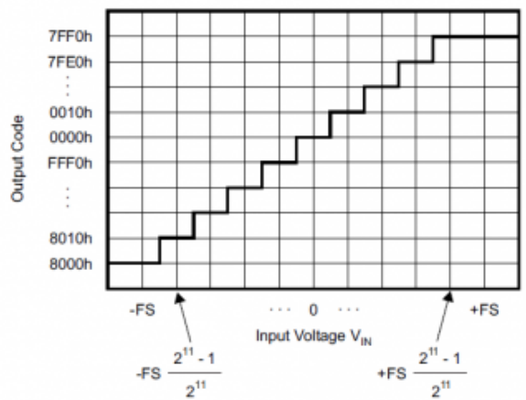


Figure 18. Code Transition Diagram

NOTE

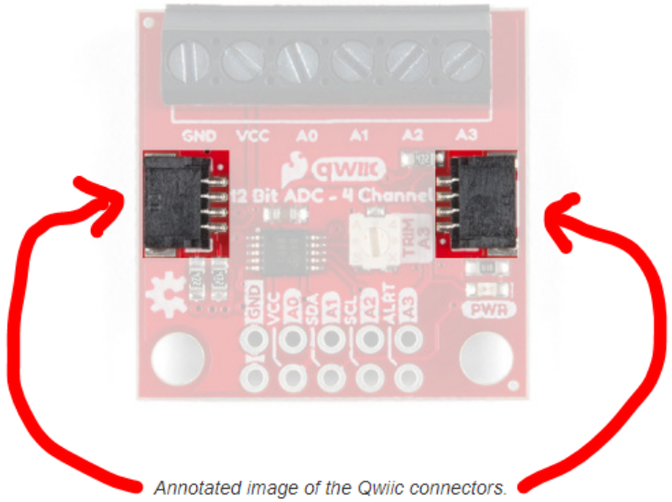
Single-ended signal measurements, where V_{AINN} = 0 V and V_{AINP} = 0 V to +FS, only use the positive code range from 0000h to 7FF0h. However, because of device offset, the ADS101x can still output negative codes in case V_{AINP} is close to 0 V.

Qwiic or I2C

The ADS1015 has 4 available I2C addresses, which are set by the address pin, ADDR. On the Qwiic ADC, the default slave address of the ADS1015 is 0x48 (HEX) of 7-bit addressing, following I2C protocol. The ADS1015 does have an additional general call address that can be used to reset all internal registers and power down the ADS1015 (see datasheet).

Default I2C Slave Address: 0x48

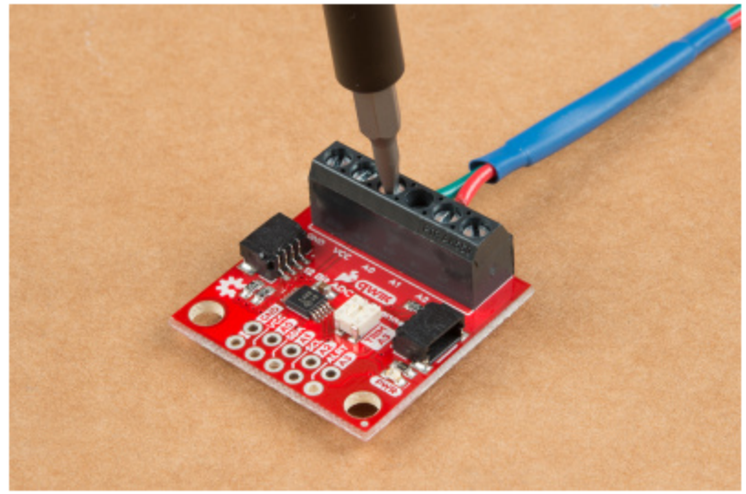
The simplest way to use the Qwiic ADC is through the Qwiic connect system. The connectors are polarized for the I2C connection and power. (*They are tied to their corresponding breakout pins.)



Annotated image of the Qwiic connectors.

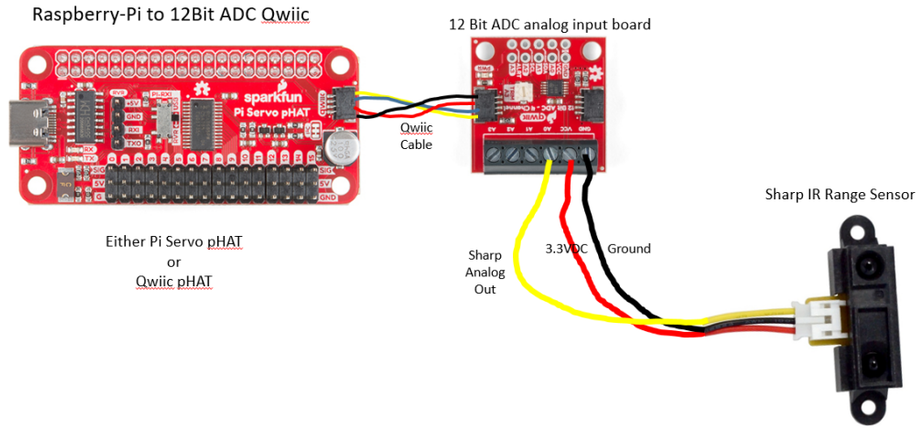
With the Qwiic connector system, assembling the hardware is fairly simple. For the examples below, all you need to do is connect your Qwiic ADC to Qwiic enabled microcontroller with a Qwiic cable. Otherwise, you can use the I2C pins, if you don't have a Qwiic connector on your microcontroller board. Just be aware of your input voltage and any logic level shifting you may need to do.

Additionally, you can connect your input voltages to the available inputs on the screw terminals and or use the breakout pins on the board. Make sure the connections are fully inserted and that you are ground looping your inputs. Ground looping can be done by connecting the ground of your input to the ground of the ADC.



Example of attaching inputs through screw terminals.

Please wire your Sharp IR range sensor in as shown:



Then, please download a copy of the **ads1015.m** function, needed to talk with the 12bit ADC from canvas. Typing **help ads1015** at the command line will bring up a detailed description of the function:

```
>> help ads1015
 ads1015 Analog-to-Digital converter.

  adc = ads1015(rpi, bus) creates a ads1015 ADC object attached to the
  specified I2C bus. The first parameter, rpi, is a raspi object. The
  I2C address of the ads1015 ADC defaults to '0x48'.

  adc = ads1015(rpi, bus, address) creates a ads1015 ADC object
  attached to the specified I2C bus and I2C address. Use this form if
  the ADDR pin is used to change the I2C address of the ads1015 from
  the default '0x48' to something else.

  [voltage, rawData] = readVoltage(adc, AINp) reads the single-ended voltage
measurement from AINp input port. Also returns the contents of conversion
  register, optionally in rawData
  Accepted values for AINp are 0,1,2,3.

  [voltage, rawData] = readVoltage(adc, AINp, AINn) reads the differential
voltage measurement between AINp and AINn input ports. Also returns the
  contents of conversion register, optionally in rawData
  Supported differential measurement pairs: (0, 1), (0, 3), (1, 3), (2, 3).


  Customizable properties of ads1015 given below:

  The "OperatingMode" property of the ads1015 ADC object determines
power consumption, speed and accuracy. The default OperatingMode is
  'single-shot' meaning that the ads1015 performs a single analog to
  digital conversion upon request and goes to power save mode. In
  continuous mode, the device performs continuous conversions.

  The "SamplesPerSecond" property sets the conversion rate.

  The "VoltageScale" property of the ads1015 ADC object determines the
  setting of the Programmable Gain Amplifier (PGA) value applied before
  analog to digital conversion. See table below to correlate the
  input voltage scale with the PGA value:

  VoltageScale | PGA Value
  -------------------------
    6.144  |   2/3
    4.096  |   1
    2.048  |   2
    1.024  |   4
    0.512  |   8
    0.256  |   16

  NOTE: Do not apply voltages exceeding VDD+0.3V to any input pin.
```

There's a lot of information here, don't panic, it's mostly for background, the actual code and hook up is surprisingly easy.

Finally, open up your Arduino Sharp Ir code and make a copy of it, rename it **SimplePiSharpRange** and then add the following changes to it..

SimplePiSharpRange2021.mlx

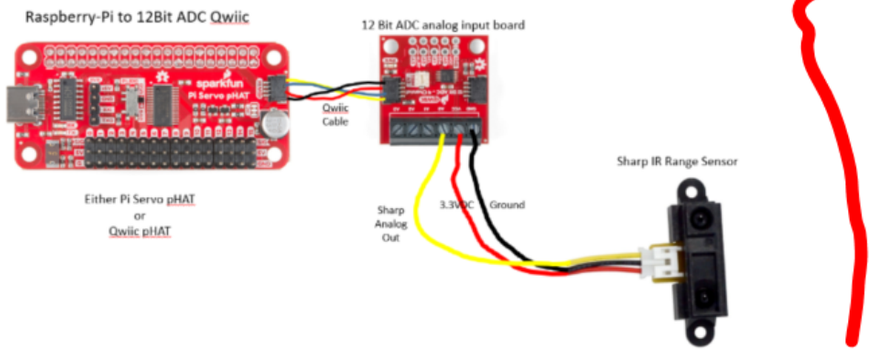
SimplePiSharpRange.mlx is a template for future Fun-Robo code

is an example of reading an analog Sharp IR Range sensor on a Raspberry Pi

uisng a Qwicc 12bit ADC board. Must have ***ads1015.m*** *function in your working directory*

hook analog out from sharp sensor into Pin A0, Vcc to 3.3 VDC, and gnd to gnd.

program collects range voltage data, plots it and is terminated by operator input.



At end of programming session, please don't forget to shutdown your Pi cleanly by using the follwing at the command line:

```
>>h = robotPi
>>h.execute('sudo shutdown -h now')
```

Sign *your names here* and give *the creation date* and *revision number*

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

Next we just need to swap in a few lines of Raspberry Pi code to replace the Arduino code. The bulk of the code will remain unchanged, but your calibration curve will differ by a lot because it will be set on the Raspberry Pis 3.3VDC power bus rather than on the Arduinos 5.9VDC one.

Modify the code that runs once to look like this:

Set up robot control system (code that runs once)

```
% Create a global raspberry PI object so that it can be used in functions
% Create a *raspi* object. You must be on Olin-Robots WIFI network and use your
% pis correct IP address, replace address in code below with yours

robotPi = raspi('192.168.10.141')

% create SparkFun Qwiic Analog Input adc device that is present on i2c bus 1 at address 0x49

adcDevice = ads1015(robotPi,'i2c-1','0x48')

disp('Warning! Data Collection Active! ');

% Configure test loop to collect n data points.

nTests = input(['Enter number of range test positions, ' ...
    'followed by enter: ']);
clc                    % clear command window

% create a variable to hold experimental position data
  positionData = zeros(nTests,2);

r = rateControl(0.1);        % create a 0.1 hz loop rate
reset(r);                    % reset loop time to zero
```

You will just need to replace the Arduino code in two spots. The two controllers are pretty similar from Matlab's point of view, and you can easily swap code back and forth between them as needed. This is one of the powers of using a higher level language like Matlab to do your robot coding in. Conceptually you will be able to port it to a new robot controller somewhere in the future if needed.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Next go to the part that loops and make the following changes:

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

```
controlFlag = 1;                % create a loop control
while (controlFlag < nTests+1)  % loop till ntests data captured

    % collect data from robot sensors
    rangeData = SENSE(adcDevice);
    THINK();                    % compute what robot should do next
    ACT();                      % command robot actuators
    beep                        % play system sound

    % store experimental commanded versus actual data
    positionData(controlFlag,1)= input('Enter actual distance (cm): ');
    gtest= input('move Sharp to new range, type G, then hit ENTER ','s');
    positionData(controlFlag,2)= rangeData;
    waitfor(r);                 % wait for loop cycle to complete
    controlFlag = controlFlag+1; % increment loop
end
```

Upgrade the **Sense** function:

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

```
  %place genratl functions here
```

Sense Functions (store all Sense related local functions here)

```
function rangeData = SENSE(adcDevice)
    disp('Sense');
    rangeData = adcDevice.readVoltage(0)  % read ADC Pin 0 voltage
end
```

Think Functions (store all Think related local functions here)

```
function THINK()
   % null function, not much thinking to do here.
end
```

Act Functions (store all Act related local functions here)

```
function ACT()
% null function, not much acting to do here.
end
```

And redo the clean up shutdown section for a Raspberry Pi as shown:

Clean shut down

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

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

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

```
% Stop program and clean up the connection to WebCam
% when no longer needed

  clc
  clear robotPi            % clear connection to Pi
  disp('Pi Program Done');
  beep        % play system sound to let user know program is ended
```

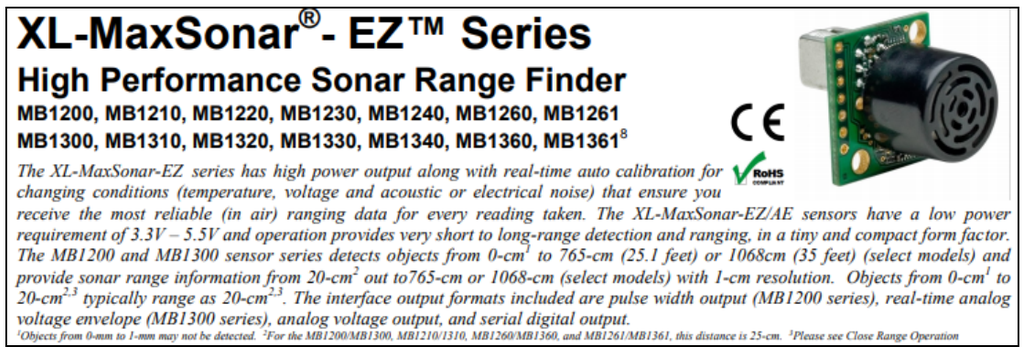
Robot Functions (store this codes local functions here)

Once you have yoru code all working, go back and re-measure the Sharp Ir calibration curve using the same procedure used with the Arduino. It should have the same overall shape but lower absolute voltages. Your final project will use Sharps and raspberry Pi, so store this data for later use.

Final Sharp range sensor perception demo: *:* Course Ninjas will place some small collection of given targets on the center target ring and set your sensor head at a midway range from the ring. As the ring slowly rotates by your Sharp range sensor head, your code should find each target, (and for extra points) correctly identify which target it is and give a good range estimate to its centroid, printing all of this back to your OCU laptop terminal in real time. Ninjas will then position the white back plane perpendicular to the sensor headset senor head at a fixed distance and slowly move obstacles (the existing targets) through the field of view of your sensors by hand. Your code should read sensors, perform perception on the data stream and send either “Open hole” or “Obstacle, range, bearing” to the OCU terminal. Your team gets massive applause if each of all 5 Obstacles are detected.

Sonar Range Sensors

Sonar range sensors work by broadcasting a spherical pressure wave front from a small piezo transducer (an ultrasonic ping) out toward targets and then measuring the time elapsed from that ping to when a return echo (pong) is heard. During this time the wavefront goes out, hits objects and bounces back. Because sound travels much slower than light, a perceptible pause can occur between measurements.



Sense Lab MB1340 Sonar Sensor

Because the ping wave front is much broader than something like a narrow Sharp Infrared beam, the wave can hit multiple targets and generate multiple return echoes. And, if you remember your 1st year physics, waves can reflect, deflect and bend around corners, causing all sorts of odd data readings.

This often can make sonar sensors complex to use, but in other cases, like attempting to sense transparent objects (glass) or objects in bright sunlight (brighter than any simple IR sensor can overcome) sonar becomes a very useful sensor to have in your toolkit. It is often used in conjunction with Sharp IR class sensors to use the best of both optical and sonar sensing technologies.

We will use one of the narrowest beam, shortest range sonar sensors in this lab: **The Ultrasonic Sensor - MB1340-000** from maxbotix sonars.

Let's look at the **MB1340**. This economical sensor provides 20cm to 765cm of non-contact measurement functionality with a resolution of 1cm. And so it has a lot longer range than the Sharp IR, but the beam is a lot wider and it has lower resolution. Each MB1340 module includes an ultrasonic transmitter, a receiver and a control circuit.

There are seven pins that you need to worry about on the MB1340:

Pin 1 -BW-Leave open (or high) for serial output on the Pin 5 output. When Pin 1 is held low the Pin 5 output sends a pulse (instead of serial data), suitable for low noise chaining.

Pin 2 -PW– For the MB1200 (EZ) sensor series, this pin outputs a pulse width representation of range. To calculate distance, use the scale factor of 58uS per cm. For the MB1300 (AE) sensor series, this pin outputs the analog voltage envelope of the acoustic wave form. The output allows the user to process the raw waveform of the sensor. (if you want to get fancy about sonar data processing and track multiple returns.

Pin 3- AN– For the 7.6 meter sensors (all sensors except for MB1260, MB1261, MB1360, and MB1361), this pin outputs analog voltage with a scaling factor of (Vcc/1024) per cm. A supply of 5V yields ~4.9mV/cm., and 3.3V yields ~3.2mV/cm. Hardware limits the maximum reported range on this output to ~700cm at 5V and ~600cm at 3.3V. The output is buffered and corresponds to the most recent range data.For the 10 meter sensors (MB1260, MB1261, MB1360, MB1361), this pin outputs analog voltage with a scaling factor of (Vcc/1024) per 2 cm. A supply of 5V yields ~4.9mV/2cm., and 3.3V yields ~3.2mV/2cm. The output is buffered and corresponds to the most recent range data.

Pin 4 -RX– This pin is internally pulled high. The XL-MaxSonar-EZ sensors will continually measure range and output if the pin is left unconnected or held high. If held low the will stop ranging. Bring high 20uS or more for range reading.

Pin 5 -TX- When Pin 1 is open or held high, the Pin 5 output delivers asynchronous serial with an RS232 format, except voltages are 0-Vcc. The output is an ASCII capital “R”, followed by three ASCII character digits representing the range in centimeters up to a maximum of 765, followed by a carriage return (ASCII 13). The baud rate is 9600, 8 bits, no parity, with one stop bit. Although the voltage of 0-Vcc is outside the RS232 standard,

most RS232 devices have sufficient margin to read 0-Vcc serial data. If standard voltage level RS232 is desired, invert, and connect an RS232 converter such as a MAX232. When Pin 1 is held low, the Pin 5 output sends a single pulse, suitable for low noise chaining (no serial data).

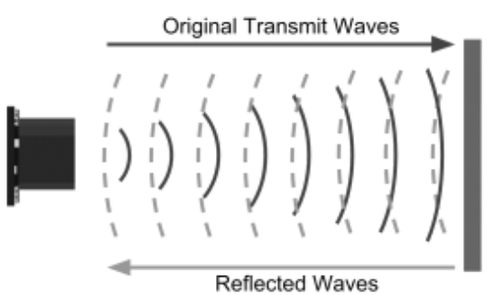
Pin 6 -+5V- Vcc – Operates on 3.3V - 5V. The average (and peak) current draw for 3.3V operation is 2.1mA (50mApeak) and at 5V operation is 3.4mA (100mA peak) respectively. Peak current is used during sonar pulse transmit. Please reference page 4 for minimum operating voltage versus temperature information.

Pin 7 -GND- Return for the DC power supply. GND (& V+) must be ripple and noise free for best operation. Trig (Trigger), Echo (Receive), and GND (Ground). You will find this sensor very easy to set up and use for your next range-finding project.

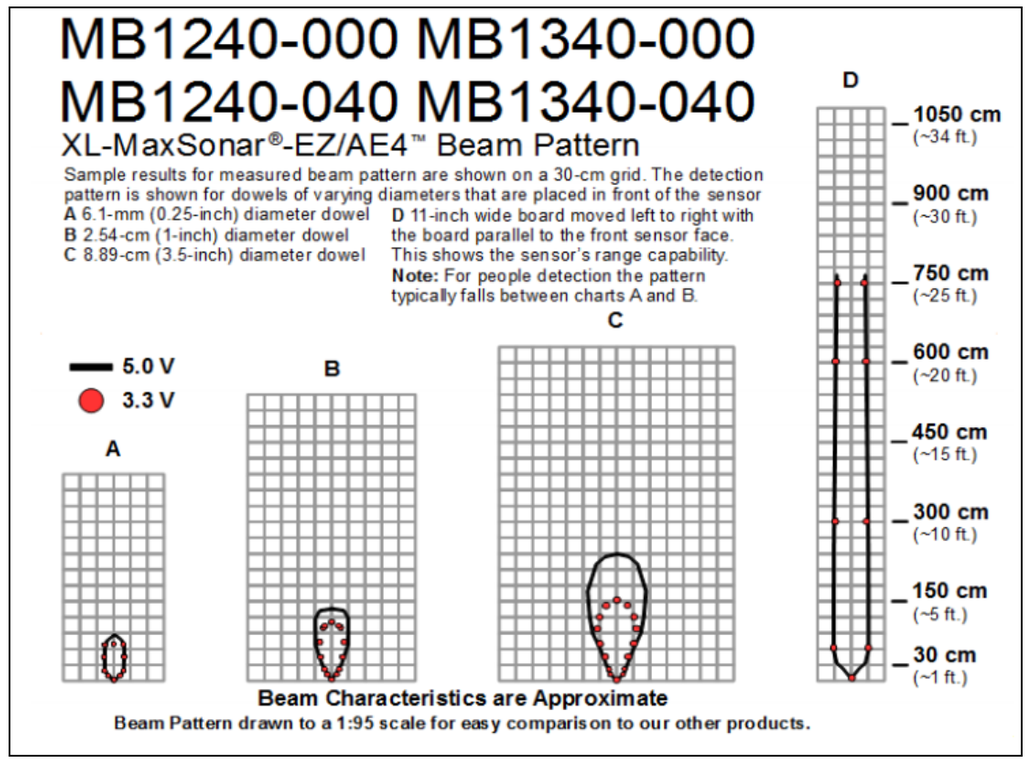
Its data sheet is here:

https://www.maxbotix.com/documents/XL-MaxSonar-EZ_Datasheet.pdf

The MB-1340 ultrasonic sensor uses sonar to determine the distance to an object like bats do. It offers excellent non-contact range detection with high accuracy and stable readings in an easy-to-use package. Its operation is not affected by sunlight or black material like sharp rangefinders are (although acoustically soft materials like cloth can be difficult to detect). And it can detect glass and transparent objects (like much of the corridor railings at Olin). How Does it Work? The ultrasonic sensor uses sonar to determine the distance to an object. Here's what happens: the transmitter sends a signal: a high-frequency sound, when the signal finds an object, it is reflected and the transmitter receives it.



Please keep in mind that although the MB1340 has the narrowest beam width in its family, but it is still pretty wide. 30cms is roughly a foot, so it's sonar beam width is around 2 feet wide. This makes it great for area coverage, but it's not a narrow laser beam by any stretch.



The time between the transmission and reception of the signal allows us to know the distance to an object. This is possible because we know the sound velocity in the air. But keep in mind that all of the wave physics still apply and that wave may not bounce back cleanly as is shown in the above illustration. It may bounce back at an angle, it might deform around and edge and then bounce back, it might hit many obstacles and bounce back at different times from all of them. It can get quite complicated, very fast.

There is a specific phenomenon called multi-pathing that haunts sonar operators where the outgoing beam reflects off multiple surfaces and creates multiple range echoes back at the transducer. This is a strong argument to mount your sonar high enough up off the ground plane so that it doesn't cause kickback echoes.

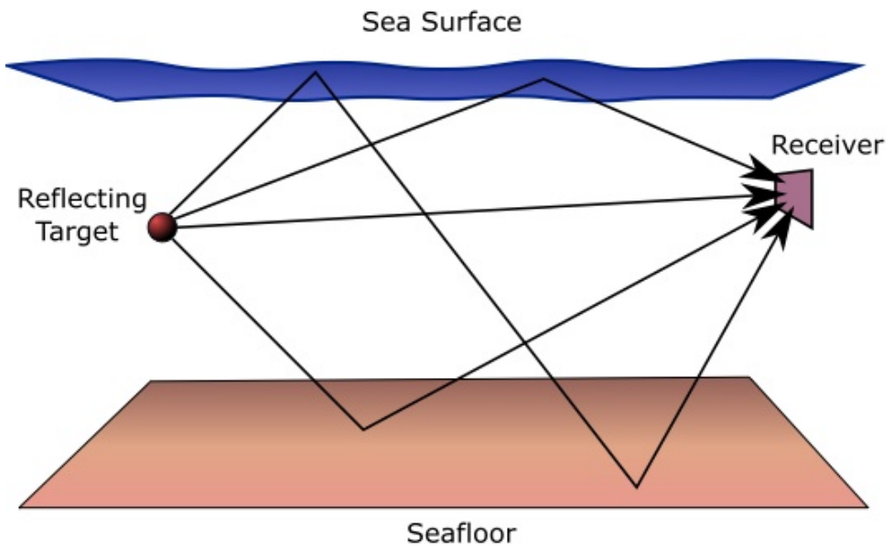


Figure: Example of multipath sonar environment

This sensor is really popular among the Arduino tinkerers. It's easy to hook up and gives great wide area coverage. When used in tandem with an array of Sharp IR sensors, you can build a really effective, multi-material sensor suite. In this lab you will use the ultrasonic sensor to range the distance to an object, to find holes and targets.

The goal of this lab is to help you understand how this sensor works. Then, you can use this example in your own projects.

The manufacturer provides a quick start guide here:

https://www.maxbotix.com/ultrasonic-sensor-hrlv%E2%80%91maxsonar%E2%80%91ez-guide-158

Arduino and ultrasonic sensors are very popular for integrating when designing solutions for many applications in robotics and automation. The MaxBotix ultrasonic sensors that interface with the Arduino platform make it easy for users to implement the needed ranging capabilities no matter the need. There are also ultrasonic sensors for Arduino with RS232, analog voltage, pulse width or I2C sensor outputs.

There is an extremely detailed guide of how to connect your MaxSonar sensor to an Arduino here:

https://www.maxbotix.com/Arduino-Ultrasonic-Sensors-085/

Request that you read it through carefully. An abbreviated set of high points are shown here. First the wiring diagram:

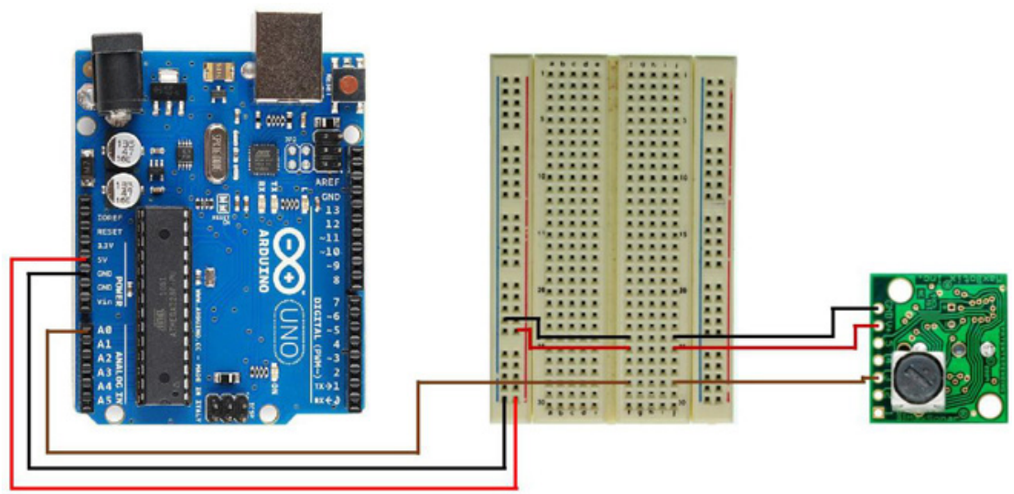
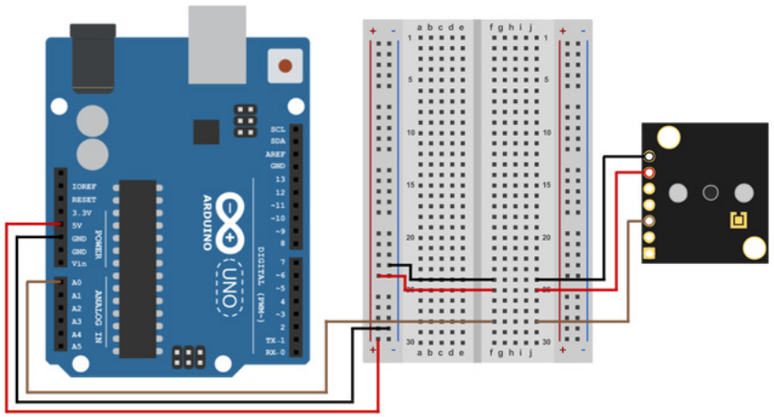


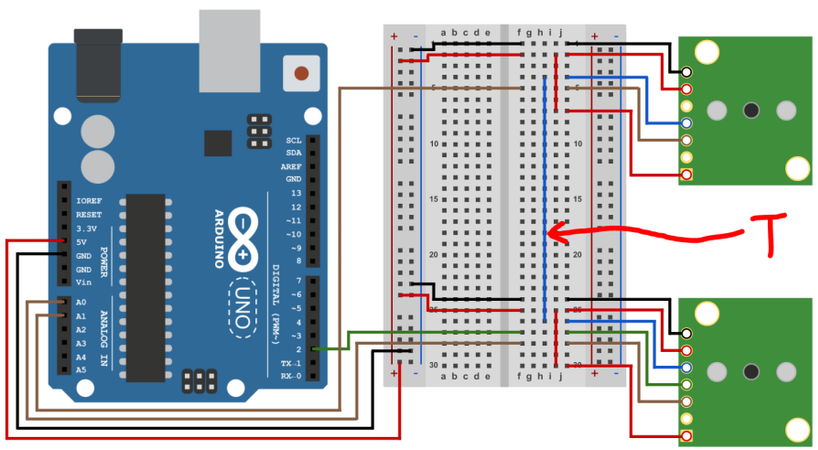
Figure : Arduino MaxSonar wiring diagram (analog input)

As is typical for this class of sensor, red wire is 5VDC, black is ground and here brown is the analog output of the board.

Please read through the detailed information above and then wire your sonar sensors from the Sonar test station, into your Arduino's analog inputs



With Multiple sonars, you need to add a wire so they fire sequentially:



The Sonar test station mounted on Sense 1 lab looks like this:

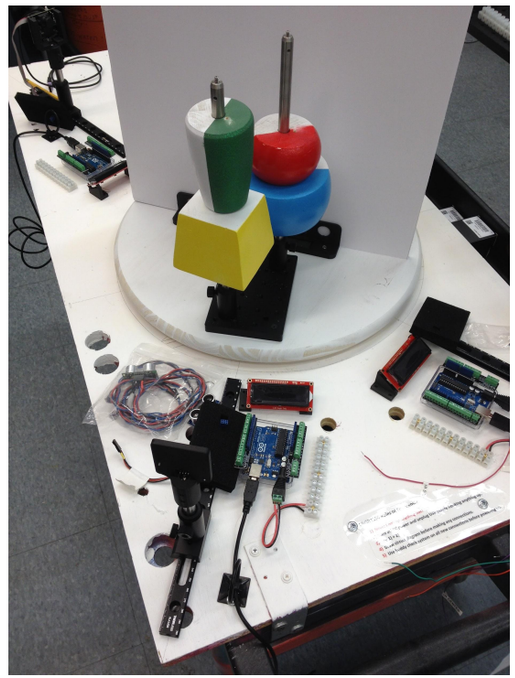


Figure: Sonar sensor test station

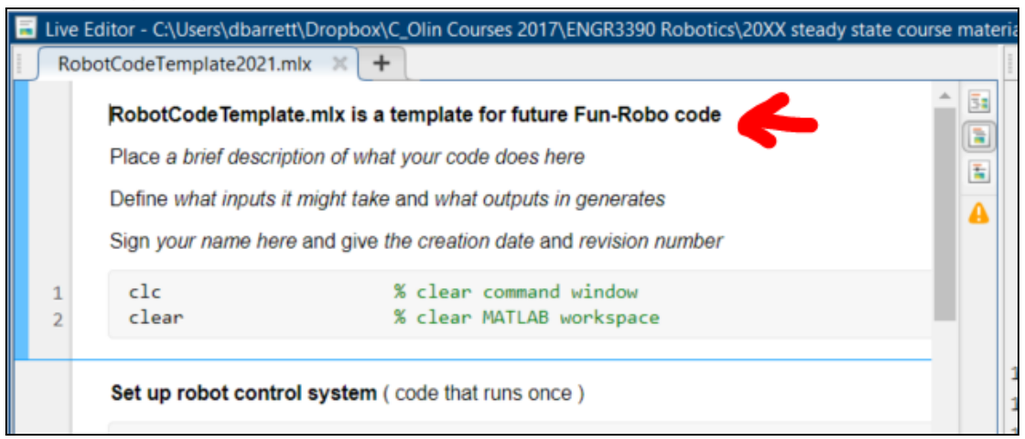
Getting the raw sonar data into the Arduino is relatively easy, but you will quickly find that you are going to have to carefully calibrate each type of sonar sensor, you might need a different calibration curve for different shape targets and you are going to need to find a way to weave all of that new data into a single function that you can use downstream to find ranges. Once you have generated that SENSE function, you will need to calibrate it, and probably filter it. As your final goal you will want to use the filtered, linearized Sonar range data to perform two separate functions; **Find the target** (any of the targets on the rotating target ring) and **Find the hole** (find the spaces between targets). Finally you will need to add a small piece of code to the

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

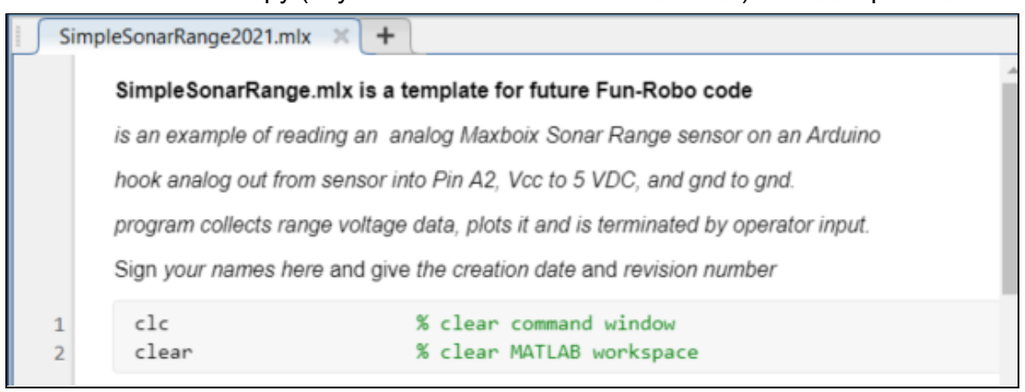
OCU user output section of your MATLAB script program that will print out “Target” or “Hole”.

Let's take on each task sequentially.

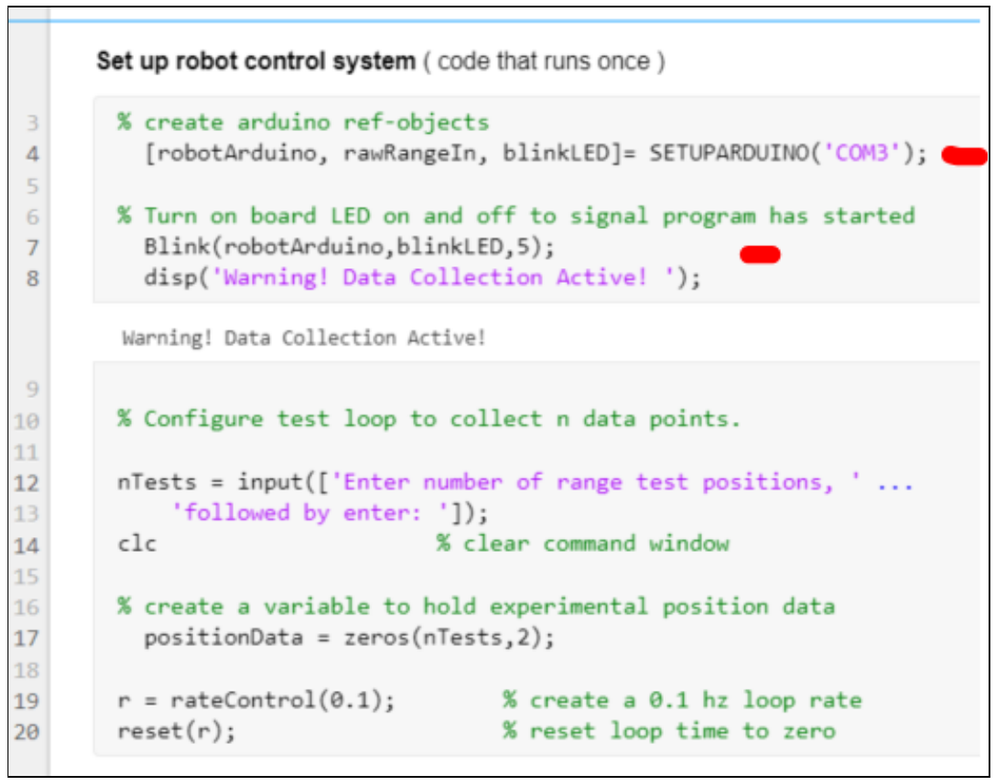
Let's take on each of these tasks sequentially. First open up a copy of your Matlab Robot Template:



And save as a new copy (in your lab team **Matlab Drive** folder) a Livescript called:



1). **To calibrate** an individual Sharp sonar sensor, start by removing all targets from the target ring, rotate sonar head until the center sonar is perpendicular to white target ring center plane. Your sonar sensor is mounted on a precision linear slide that will let you move it in and out at fixed annotated intervals from the target plane. You will collect about 10 range data points, from white backdrop, from near to far at fixed intervals. Repeat this process for the black center plane, at exactly the same intervals. Plot both sets of data on a single graph. Save it for your lab report. What do you see? To help you get started, please add the following statements to your template:



ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

This **code that runs once** uses two functions, the first configures your Arduino, please enter it as shown:

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

```
function [robotArduino, rawRangeIn, blinkLED]= SETUPARDUINO(COMPORT)
% SETUPARDUINO creates and configures an arduino to be a simple robot
% controller. It requires which COM port your Arduino is attached to
% as its input and returns an Arduino object called robotArduino
% D. Barrett 2021 Rev A

% Create a global arduino object so that it can be used in functions
% a = arduino('setToYourComNumber','Uno','Libraries','Servo');
   robotArduino = arduino(COMPORT,'Uno','Libraries','Servo');

% configure pin 13 as a digitial-out LED
   blinkLED = 'D13';
   configurePin(robotArduino,blinkLED,'DigitalOutput');

% Configure A0 pin as an analog input
  rawRangeIn = 'A2';
  configurePin(robotArduino,rawRangeIn,'AnalogInput')

end
```

And it is simply configuring your Arduino as to what libraries to preload. It sets up pin **D13** to be your robot blinky light and it configures pin **A2** to be an analog input. By taking care of all of the plumbing to set up your Arduino, down here in a function, you both keep your main body of code clear and clean as well as make it very modular. If you choose to replace your Arduino with another controller, like a Raspberry Pi, you just need to redo this function, not the whole body of the code. This will be a real time savings throughout this course.

Next add a new **Blink** function. All robots need a blinky light to give the operator a visual clue something is happening. If you write one blinky light early on in your career, you can reuse it over and over for all sorts of indicator functions downstream. Please code as shown below.

```
function [] = Blink(a,LED, n)
  % Blink toggles Arduino a LED on and off to indicate program running
  % input n is number of blinks
  % no output is returned
  % dbarrett 1/14/20
    for bIndex = 1:n
      writeDigitalPin(a, LED, 0);
      pause(0.2);
      writeDigitalPin(a, LED, 1);
      pause(0.2);
    end
  end
```

Moving on to the main control loop:

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

```
controlFlag = 1;                % create a loop control
while (controlFlag < nTests+1)  % loop till ntests data captured

    % collect data from robot sensors
    rangeData = SENSE(robotArduino, rawRangeIn)
    THINK();                    % compute what robot should do next
    ACT();                      % command robot actuators

    % store experimental commanded versus actual data
    positionData(controlFlag,1)= input('Enter actual distance (cm): ');
    gtest= input('move Sonar to new range, type G, then hit ENTER ','s');
    positionData(controlFlag,2)= rangeData;
    Blink(robotArduino,blinkLED,1);
    waitfor(r);                 % wait for loop cycle to complete
    controlFlag = controlFlag+1; % increment loop
end
```

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Please enter the code as shown and then let us walk through it step by step.
All of the actual data collection takes place in the **SENSE** function:

Sense Functions (store all Sense related local functions here)

```
82  function rangeData = SENSE(robotArduino, rawRangeIn)
83      disp('Sense');
84      rangeData = readVoltage(robotArduino, rawRangeIn);
85  end
```

This function is straightforward, it reads the voltage on the analog pin specified by **rawRangeIn** and returns it as **rangeData**. You could wonder why we'd put this down in a function and not just up in main code. It's a good question. In practice, we always will write highly modular code, both for clarity and to allow for future expansion. While this is currently a one line function, it's just returning an extremely non-linear voltage for range. What you will want to return is far more likely to be a linearized, metroligized range distance in centimeters. When you add all of that linearization and calibration code to this function it will be a quarter of a page long, and there is no way you want that upstairs in the control loop!

Next the **THINK** and **ACT** functions, at the moment, are just empty placeholders for future expandability:

Think Functions (store all Think related local functions here)

```
86  function THINK()
87     % null function, not much thinking to do here.
88  end
```

Act Functions (store all Act related local functions here)

```
89  function ACT()
90  % null function, not much acting to do here.
91  end
```

Moving on, the remaining procedural code asks the operator to measure actual range to target and enter it. Prompts them to move the sensor to a new range and then repeats the full process for **nTest** cycles.

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

```
21  controlFlag = 1;                % create a loop control
22  while (controlFlag < nTests+1)  % loop till ntests data captured
23
24      % collect data from robot sensors
25      rangeData = SENSE(robotArduino, rawRangeIn)
26      THINK();                    % compute what robot should do next
27      ACT();                      % command robot actuators
28
29      % store experimental commanded versus actual data
30      positionData(controlFlag,1)= input('Enter actual distance (cm): ');
31      gtest= input('move Sonar to new range, type G, then hit ENTER ','s');
32      positionData(controlFlag,2)= rangeData;
33      Blink(robotArduino,blinkLED,1);
34      waitfor(r);                 % wait for loop cycle to complete
35      controlFlag = controlFlag+1; % increment loop
36  end
```

The output of the livescript will look something like this:

```
Sense
rangeData = 0.8895

Sense
rangeData = 1.1144

Sense
rangeData = 1.2170

Sense
rangeData = 1.2805

Sense
rangeData = 1.5982

Sense
rangeData = 1.7791

Sense
rangeData = 2.0968

Sense
```

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

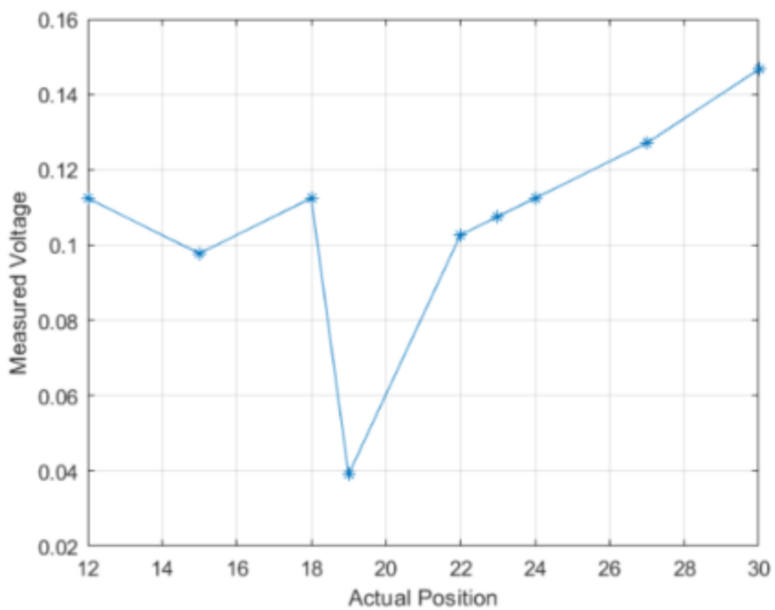
After collecting your data, you will want to process and see it, so please add the following section to make a pretty plot:

Mission data processing

For many robot applications, you will need to post-process the data collected after the mission. Here we will plot the measured versus actual range positions.

```
% Plot and store commanded position data vs. actual position
  plot(positionData(:,1), positionData(:,2), '-*')
  xlabel('Actual Position')
  ylabel ('Measured Voltage')
  grid
```

Which will generate a nice Sonar voltage versus range plot for your sensor:



Wrapping it all up, please add a short piece of cleanup code to shut down your Arduino cleanly. Please note: If you don't shut down embedded controllers like Arduinos and Raspberry Pi's they will run indefinitely. Overnight, for months and years into the future. This can make your life really complicated and drive your teammates crazy, so please shut down cleanly:

Clean shut down

finally, with most embeded robot controllers, its good practice to put all actualtors into a safe position and then release all control objects and shut down all communication paths. This keeps systems form jamming when you want to run again.

```
 % Stop program and clean up the connection to Arduino
% when no longer needed

  clc
  disp('Arduino program has ended');
```

```
Arduino program has ended
```

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

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

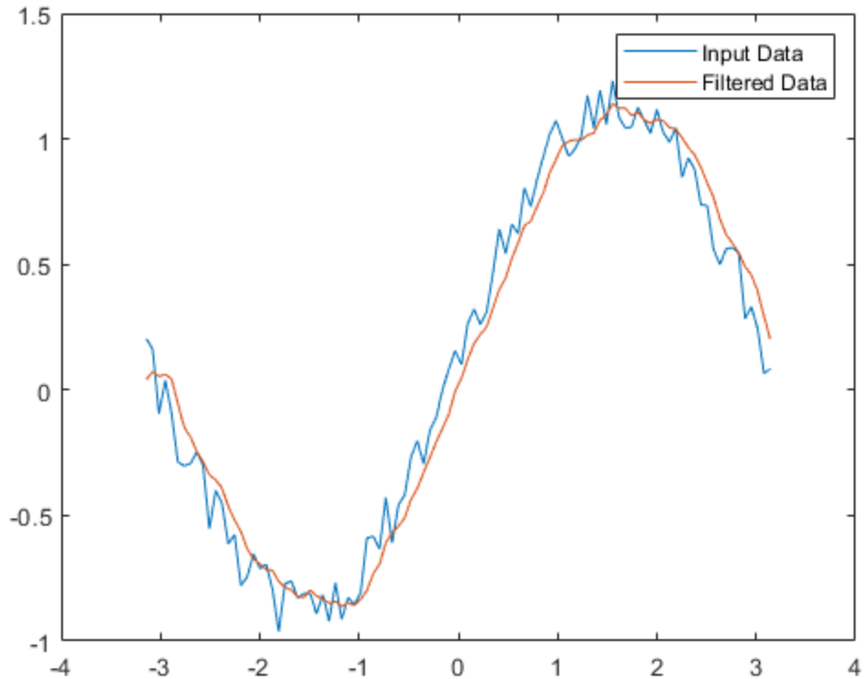
With this code complete, you've done the hard part of sensing, namely getting the raw data in from the environment around the Robot. The next steps are to linearize it and perhaps filter it so that it is clean enough to work with. Then you can calibrate it so that your sense functions produce ranges in centimeters that your future robot brain can work with and not in volts (which it can't !). Please work with your pair programming partner and take on the next steps.

2). **Linearize**. With this raw data and the linearization background algorithm information given to you in canvas, develop a MATLAB function that will convert the raw sensor data, collected in volts, into useful range data in cm (or inches). Write a simple MATLAB function that when called by the main program range0 = readSharp(0), range1 = readSharp(1), etc.); that reads the raw sensor voltage, applies your calibration/linearization function to it and returns range in engineering units, either cms or inches. Hint: You may need to come up with some clever interpolation way to incorporate the difference between a white target and a black one. See Ninjas or instructors for suggestions if you need a bit of help here.

3). **Filtering?** Next, write a MATLAB script to collect 10 range samples at a single distance from white backplane target about one second apart. Record them in a rangeDataOvertime variable matrix along with the sample time of each measurement and plot them. Does the data vary with time? If so, you may need to develop a low-pass software **filter** to provide clean data to the perception code that will consume it. You could add this software filter to your readSharp(n) function as well.

For filter background/help, see:
https://www.megunolink.com/articles/3-methods-filter-noisy-arduino-measurements/

And MATLAB has extensive filtering functions already built in. As a starting point see:
https://www.mathworks.com/help/matlab/ref/filter.html



4). **Check sensor data processing robustness**. Your next step is to determine how your SharpIR functions work on colored and oddly shaped targets. Collect 10 data sets of actual range and measured range data from the red sphere, yellow trapezoid, etc. Save data into a MATLAB variable and plot. In an ideal world the graph of actual distance versus measured distance would be a clean 45 degree line for any color or surface. If it's not, go back and add additional code to your function to deal with any color or surface issues that arise. You may want to alter your function to return both a range and a confidence value. See Ninjas for help here, if you get bogged down.

Having got your Sharp range sensors calibrated, linearized, filtered and solid, you can move on to using the range data from all three sensors mounted on the perception head to take on your labs main perception functions,

5). **Find Target Or Hole**. Using your existing MATLAB code, write a simple sharp IR range sensor script that prints out something like:

"hole...hole...hole...target...hole...hole"

for about ten seconds as the target turntable passes in front of the sensor head. As a starting point, slowly rotate a variety of targets past the sensor head at a set of distances and record the data you collect as they go by. Can you see each target? When do you first see them? Can you tell the difference between them? Extra points here for being able to use the full sensor head to tell which object is passing your sensing station. More extra points for porting this code into a MATLAB APP stand alone application.

Given the final two week demo goal below, modify your existing Robot Control code to do the following demo functionality.

Final Sonar range sensor perception demo: *:* Course Ninjas will place some small collection of given targets on the center target ring and set your sensor head at a midway range from the ring. As the ring slowly rotates by your Sonar range sensor head, your code should find each target, (and for extra points) correctly identify which target it is and give a good range estimate to its centroid, printing all of this back to your OCU laptop terminal in real time. Ninjas will then position the white back plane perpendicular to the sensor headset senor head at a fixed distance and slowly move obstacles (the existing targets) through the field of view of your sensors by hand. Your code should read sensors, perform perception on the data stream and send either "Open hole" or "Obstacle, range, bearing" to the OCU terminal. Your team gets massive applause if each of all 5 Obstacles are detected.

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

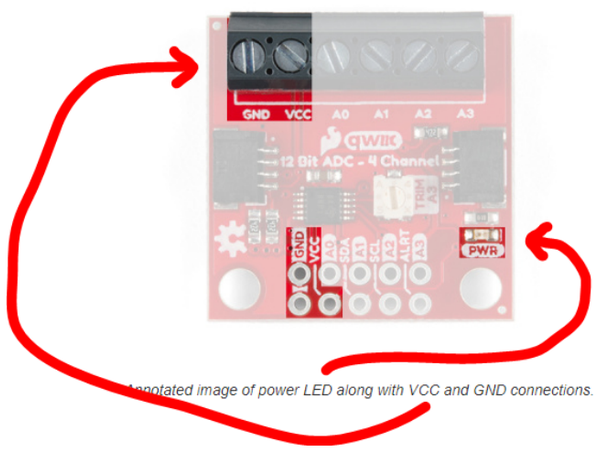
Sonar Range Sensors with a Raspberry-Pi (Skip2022)

The Raspberry Pi has many awesome talents, but directly reading the output of Analog sensors isn't one of them. The Pi doesn't have analog input ports like an Arduino. To overcome this shortcoming we are going to give you a i2C bus Analog Input device and a pre-written Matlab function to call it. With that support, you can easily do a light edit of your Arduino code to port it to your Raspberry Pi.

First off, please shut power to your Raspberry Pi, attach qwiic pHAT to it, read through this section and then wire in Sharp Ir range sensor as described below:

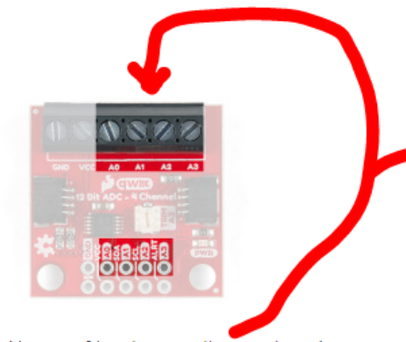
Qwiic 12-Bit ADC Hardware Hookup Guide:

Power; There is a power status LED to help make sure that your Qwiic ADC is getting power. You can power the board either through the polarized Qwiic connector system or the breakout pins (3.3V and GND) provided. This Qwiic system is meant to use 3.3V, be sure that you are NOT using another voltage when using the Qwiic system.

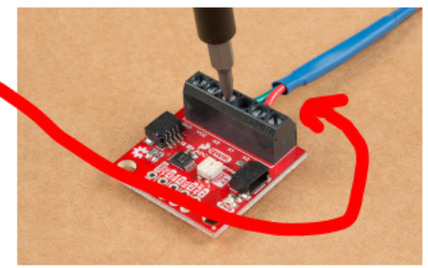


Annotated image of power LED along with VCC and GND connections.

There are four input measurement channels for the ADS1015, labeled AIN0-3, accessible through the screw pin terminals shown below. With the Qwiic system, the absolute minimum and maximum voltage for these inputs is -.3V and 3.6V, respectively.



Annotated image of input connections on board. Click to enlarge.

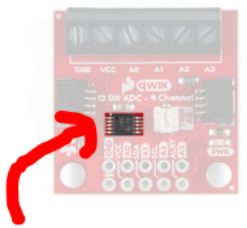


Example of attaching inputs through screw terminals. Click to enlarge.

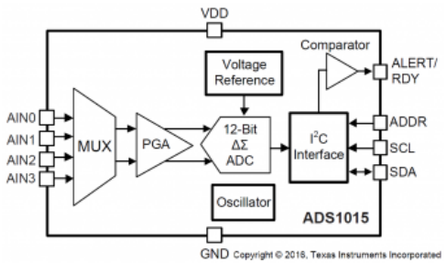
Depending on the settings for the multiplexer (MUX) in the ADS1015, the analog voltage readings (or conversions on the ADC) will be either for 4 single-ended inputs or 2 differential pairs.

Pin Label	Configuration Options
AIN_0	Single-Ended Differential Inputs: AIN1 or AIN3 w/ Programmable Comparator
AIN_1	Single-Ended Differential Ref. w/ Programmable Comparator Differential Input: AIN3 w/ Programmable Comparator
AIN_2	Single-Ended Differential Input: AIN3 w/ Programmable Comparator
AIN_3	Single-Ended Differential Ref. w/ Programmable Comparator On-board 10kΩ Potentiometer

The ADS1015 ADC is a low-power 12-bit analog-to-digital converter (ADC), which includes a built-in integrated voltage reference and oscillator. At the core of its operation, the ADC uses a switched-capacitor input stage and a delta-sigma (ΔΣ) modulator to determine the differential between AINP (positive analog input) and AINN (negative analog input). Once the conversion is completed, the digital output is accessible over the I2C bus from the internal conversion register.



Annotated image of ADS1015 on board. Click to enlarge.



Functional block diagram from datasheet. Click for more details.

The ADS1015 is a powerful tool with multiple configuration settings, set by the Config Register, for the analog voltage readings (or conversions). In the following sections, we will cover the general operation of the ADS1015. For exact details of the various configuration settings, please refer to the manufacturer datasheet. The operational characteristics of the ADS1015 are summarized in the table to right.

Characteristic	Description
Operating Voltage (V_{DD})	2.0V to 5.5V (**Default on Qwiic System: 3.3V**)
Operating Temperature	-40°C to 125°C
Operation Modes	Single-Shot (**Default**), Continuous-Conversion, and Duty Cycling
Analog Inputs	Measurement Type: Single-Ended or Differential Input Voltage Range: **GND** to V_{DD} (*see **Caution** note, below*) Maximum Voltage Measurement: *Smallest of* V_{DD} or **FSR** Full Scale Range (FSR): ±.256V to ±6.114V (**Default: 2.048V**)
Resolution	12-bit (Differential) or 11-bit (Single-Ended) LSB size: 0.125mV - 3mV (**Default: 1 mV** *based on FSR*)
Sample Rate	128 Hz to 3.3 kHz (**Default: 1600SPS**)
Current Consumption (*Typical*)	Operating: 150µA to 200µA Power-Down: 0.5µA to 2µA
I²C Address	**0x48** (**Default**), 0x49, 0x4A, or 0x4B

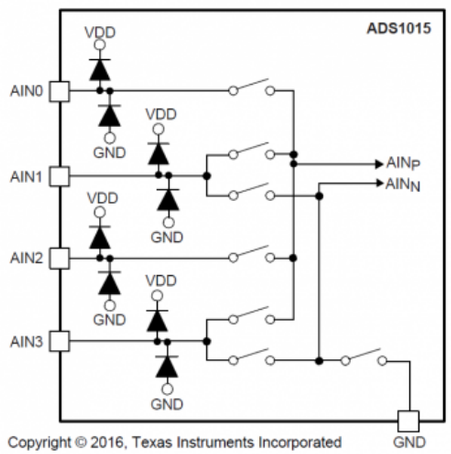
⚡**Caution:** The absolute, analog input voltage range is **(GND - .3V) to (V_{DD} + .3V)**; anything slightly higher/lower may damage the ADC chip. If the voltages on the input pins can potentially violate these conditions, as specified by the datasheet, use external Schottky diodes and series resistors to limit the input current to safe values.

The ADS1015 has 2 different conversion modes: single-shot and continuous-conversion with the ability to support duty cycling. Through these modes, the ADS1015 is able to optimize its performance between low power consumption and high data rates. We will use Single shot

Single-Shot

By default, the ADS1015 operates in single-shot mode. In single-shot mode, the ADC only powers up for ~25µs to convert and store the analog voltage measurement in the conversion register before powering down. The ADS1015 only powers up again for data retrieval. The power consumption in this configuration is the lowest, but it is dependent on the frequency at which data is converted and read.

There are four input measurement channels for the ADS1015, labeled AIN0-3. The input multiplexer controls which of those channels operates as the AINP (positive analog input) and AINN (negative analog input) to the ADC. Depending on the input MUX configuration, the voltage measurements will be either on single-ended inputs or as differential pairs.



Operational digram of multiplexer from datasheet.

Although it is listed as a 12-bit ADC, the ADS1015 operates as an 11-bit ADC when used with single-ended (individual) inputs. The 12th bit only comes into play in differential mode, as a sign (+ or -) indicator for the digital output. This allows the digital output to represent the full positive and negative range of the FSR (see table and figure below).

Table 3. Input Signal Versus Ideal Output Code

INPUT SIGNAL $V_{IN} = (V_{AINP} - V_{AINN})$	IDEAL OUTPUT CODE(1)(1)
≥ +FS $(2^{11} - 1)/2^{11}$	7FF0h
$+FS/2^{11}$	0010h
0	0000h
$-FS/2^{11}$	FFF0h
≤ –FS	8000h

(1) Excludes the effects of noise, INL, offset, and gain errors.

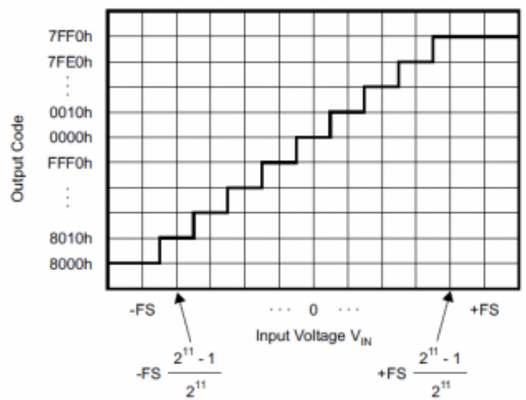


Figure 18. Code Transition Diagram

NOTE

Single-ended signal measurements, where V_{AINN} = 0 V and V_{AINP} = 0 V to +FS, only use the positive code range from 0000h to 7FF0h. However, because of device offset, the ADS101x can still output negative codes in case V_{AINP} is close to 0 V.

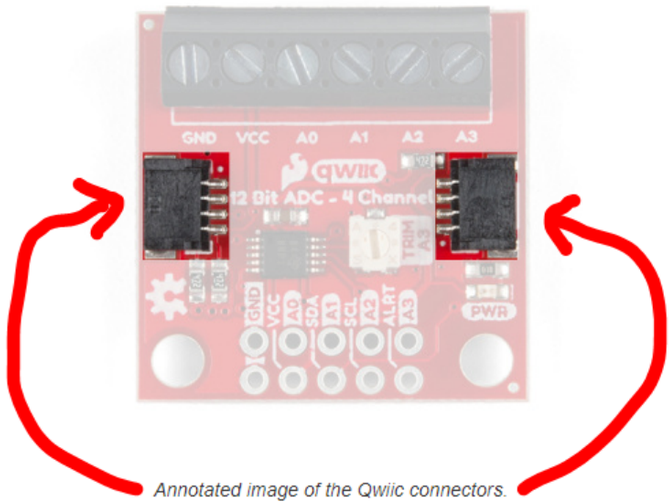
Qwiic or I2C

The ADS1015 has 4 available I2C addresses, which are set by the address pin, ADDR. On the Qwiic ADC, the default slave address of the ADS1015 is 0x48 (HEX) of 7-bit addressing, following I2C protocol. The ADS1015 does have an additional general call address that can be used to reset all internal registers and power down the ADS1015 (see datasheet).

Default I2C Slave Address: 0x48

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

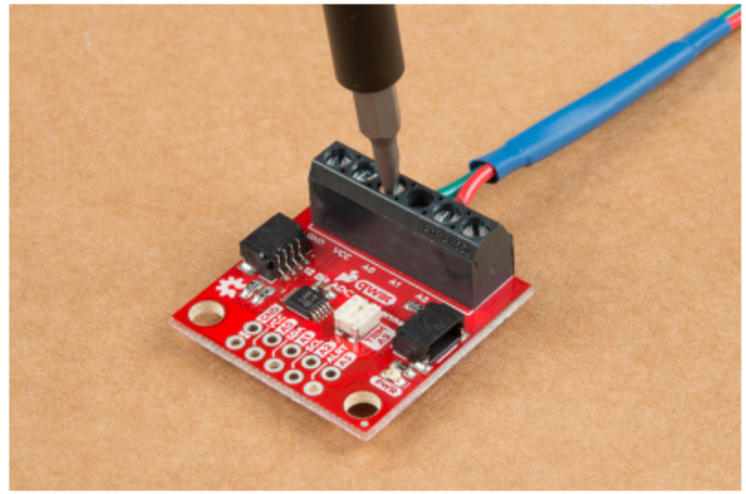
The simplest way to use the Qwiic ADC is through the Qwiic connect system. The connectors are polarized for the I2C connection and power. (*They are tied to their corresponding breakout pins.)



Annotated image of the Qwiic connectors.

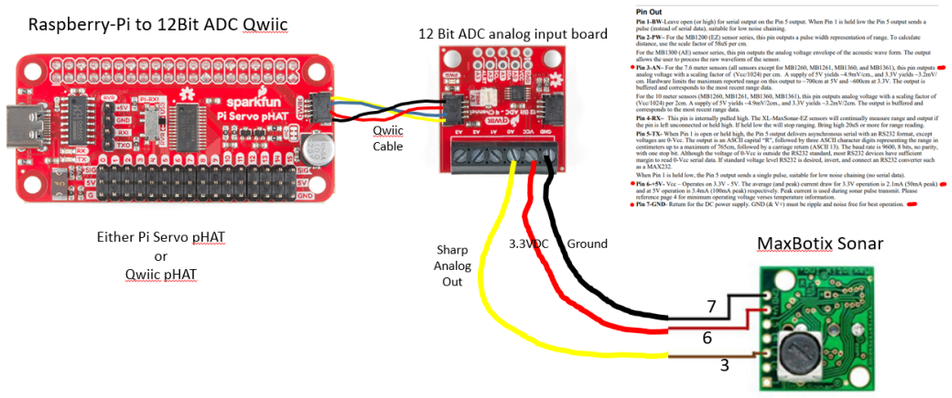
With the Qwiic connector system, assembling the hardware is fairly simple. For the examples below, all you need to do is connect your Qwiic ADC to Qwiic enabled microcontroller with a Qwiic cable. Otherwise, you can use the I2C pins, if you don't have a Qwiic connector on your microcontroller board. Just be aware of your input voltage and any logic level shifting you may need to do.

Additionally, you can connect your input voltages to the available inputs on the screw terminals and or use the breakout pins on the board. Make sure the connections are fully inserted and that you are ground looping your inputs. Ground looping can be done by connecting the ground of your input to the ground of the ADC.



Example of attaching inputs through screw terminals.

Please wire your Sharp IR range sensor in as shown:



Then, please download a copy of the **ads1015.m** function, needed to talk with the 12bit ADC from canvas. Typing **help ads1015** at the command line will bring up a detailed description of the function:

>> help ads1015

ads1015 Analog-to-Digital converter.

adc = ads1015(rpi, bus) creates a ads1015 ADC object attached to the specified I2C bus. The first parameter, rpi, is a raspi object. The I2C address of the ads1015 ADC defaults to '0x48'.

adc = ads1015(rpi, bus, address) creates a ads1015 ADC object attached to the specified I2C bus and I2C address. Use this form if the ADDR pin is used to change the I2C address of the ads1015 from the default '0x48' to something else.

[voltage, rawData] = readVoltage(adc, AINp) reads the single-ended voltage measurement from AINp input port. Also returns the contents of conversion register, optionally in rawData
Accepted values for AINp are 0,1,2,3.

[voltage, rawData] = readVoltage(adc, AINp, AINn) reads the differential voltage measurement between AINp and AINn input ports. Also returns the contents of conversion register, optionally in rawData
Supported differential measurement pairs: (0, 1), (0, 3), (1, 3), (2, 3).

Customizable properties of ads1015 given below:

The "OperatingMode" property of the ads1015 ADC object determines power consumption, speed and accuracy. The default OperatingMode is 'single-shot' meaning that the ads1015 performs a single analog to digital conversion upon request and goes to power save mode. In continuous mode, the device performs continuous conversions.

The "SamplesPerSecond" property sets the conversion rate.

The "VoltageScale" property of the ads1015 ADC object determines the setting of the Programmable Gain Amplifier (PGA) value applied before analog to digital conversion. See table below to correlate the input voltage scale with the PGA value:

VoltageScale	PGA Value
6.144	2/3
4.096	1
2.048	2
1.024	4
0.512	8
0.256	16

NOTE: Do not apply voltages exceeding VDD+0.3V to any input pin.

There's a lot of information here, don't panic, it's mostly for background, the actual code and hook up is surprisingly easy.

Finally, open up your Arduino Sharp Ir code and make a copy of it, rename it **SimplePiSharpRange** and then add the following changes to it..

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

SimplePiSonarRange2021.mlx

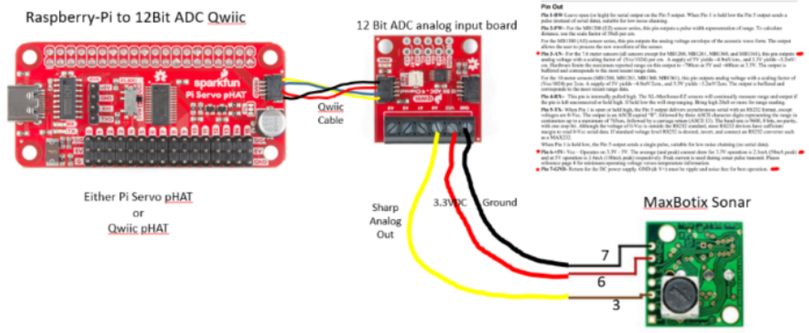
SimplePiSonarRange.mlx is a template for future Fun-Robo code

is an example of reading an analog Sharp IR Range sensor on a Raspberry Pi

uisng a Qwicc 12bit ADC board. Must have ***ads1015.m*** *function in your working directory*

hook analog out from sharp sensor into Pin A0, Vcc to 3.3 VDC, and gnd to gnd.

program collects range voltage data, plots it and is terminated by operator input.



At end of programming session, please don't forget to shutdown your Pi cleanly by using the follwing at the command line:

```
>>h = robotPi
>>h.execute('sudo shutdown -h now')
```

Sign *your names here* and give *the creation date* and *revision number*

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

Next we just need to swap in a few lines of Raspberry Pi code to replace the Arduino code. The bulk of the code will remain unchanged, but your calibration curve will differ by a lot because it will be set on the Raspberry Pis 3.3VDC power bus rather than on the Arduinos 5.9VDC one.

Modify the code that runs once to look like this:

Set up robot control system (code that runs once)

```
% Create a global raspberry PI object so that it can be used in functions
% Create a *raspi* object. You must be on Olin-Robots WIFI network and use your
% pis correct IP address, replace address in code below with yours

robotPi = raspi('192.168.10.141')

% create SparkFun Qwiic Analog Input adc device that is present on i2c bus 1 at address 0x49

adcDevice = ads1015(robotPi,'i2c-1','0x48')

disp('Warning! Data Collection Active! ');

% Configure test loop to collect n data points.

nTests = input(['Enter number of range test positions, ' ...
    'followed by enter: ']);
clc                     % clear command window

% create a variable to hold experimental position data
  positionData = zeros(nTests,2);

r = rateControl(0.1);       % create a 0.1 hz loop rate
reset(r);                   % reset loop time to zero
```

You will just need to replace the Arduino code in two spots. The two controllers are pretty similar from Matlab's point of view, and you can easily swap code back and forth between them as needed. This is one of the powers of using a higher level language like Matlab to do your robot coding in. Conceptually you will be able to port it to a new robot controller somewhere in the future if needed.

Next go to the part that loops and make the following changes:

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

```
26  controlFlag = 1;                % create a loop control
27  while (controlFlag < nTests+1)  % loop till ntests data captured
28
29      % collect data from robot sensors
30      rangeData = SENSE(adcDevice);
31      THINK();                    % compute what robot should do next
32      ACT();                      % command robot actuators
33      beep                        % play system sound
34
35      % store experimental commanded versus actual data
36      positionData(controlFlag,1)= input('Enter actual distance (cm): ');
37      gtest= input('move Sharp to new range, type G, then hit ENTER ','s');
38      positionData(controlFlag,2)= rangeData;
39      waitfor(r);                 % wait for loop cycle to complete
40      controlFlag = controlFlag+1; % increment loop
41  end
```

Upgrade the **Sense** function:

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bulk of the work should be done by functions

```
55    %place genratl functions here
56
```

Sense Functions (store all Sense related local functions here)

```
57  function rangeData = SENSE(adcDevice)
58      disp('Sense');
59      rangeData = adcDevice.readVoltage(0)  % read ADC Pin 0 voltage
60  end
```

Think Functions (store all Think related local functions here)

```
61  function THINK()
62     % null function, not much thinking to do here.
63  end
```

Act Functions (store all Act related local functions here)

```
64  function ACT()
65  % null function, not much acting to do here.
66  end
```

And redo the clean up shutdown section for a Raspberry Pi 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.

```
48  % Stop program and clean up the connection to WebCam
49  % when no longer needed
50
51    clc
52    clear robotPi            % clear connection to Pi
53    disp('Pi Program Done');
54    beep        % play system sound to let user know program is ended
```

Robot Functions (store this codes local functions here)

Once you have yoru code all working, go back and re-measure the Sonar calibration curve using the same procedure used with the Arduino. It should have the same overall shape but lower absolute voltages. Your final project will use Sonars and raspberry Pi, so store this data for later use.

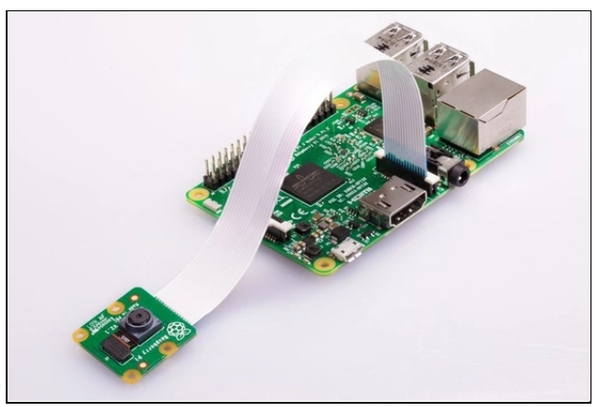
Final Sharp range sensor perception demo: *:* Course Ninjas will place some small collection of given targets on the center target ring and set your sensor head at a midway range from the ring. As the ring slowly rotates by your Sonar range sensor head, your code should find each target, (and for extra points) correctly identify which target it is and give a good range estimate to its centroid, printing all of this back to your OCU laptop terminal in real time. Ninjas will then position the white back plane perpendicular to the sensor headset senor head at a fixed distance and slowly move obstacles (the existing targets) through the field of view of your sensors by hand. Your code should read sensors, perform perception on the data stream and send either “Open hole” or “Obstacle, range, bearing” to the OCU terminal. Your team gets massive applause if each of all 5 Obstacles are detected.

Pi Camera V2 Target Tracking Sensors (skip 2022)

The Raspberry Pi Camera Module v2 replaced the original Camera Module in April 2016. The v2 Camera Module has a Sony IMX219 8-megapixel sensor (compared to the 5-megapixel OmniVision OV5647 sensor of the original camera).

The Camera Module can be used to take high-definition video, as well as stills photographs. It's easy to use for beginners, but has plenty to offer advanced users if you're looking to expand your knowledge. There are lots of examples online of people using it for time-lapse, slow-motion, and other video cleverness. You can also use the libraries bundled with the camera to create effects.

You can read all the gory details about IMX219 and the Exmor R back-illuminated sensor architecture on Sony's website, but suffice to say this is more than just a resolution upgrade: it's a leap forward in image quality, colour fidelity, and low-light performance. It supports 1080p30, 720p60 and VGA90 video modes, as well as still capture. It attaches via a 15cm ribbon cable to the CSI port on the Raspberry Pi. The camera works with all models of Raspberry Pi 1, 2, 3 and 4. It can be accessed through the MMAL and V4L APIs, and there are numerous third-party libraries built for it.

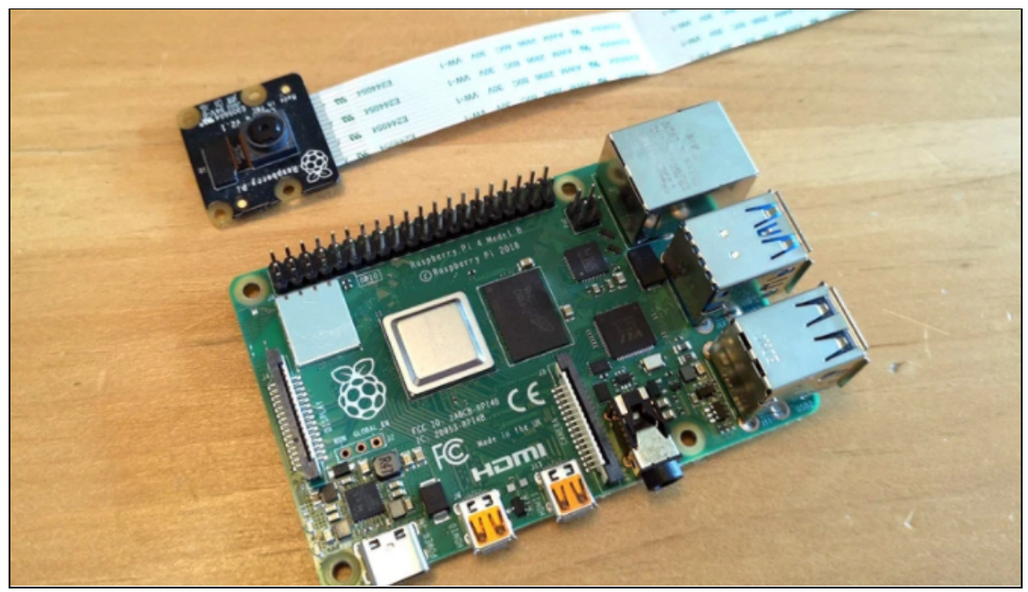
! 

How to install and use the Raspberry Pi Camera Module

In this step by step guide we will explain how to install the Raspberry Pi Camera Module , along with how to take pictures and videos with it.

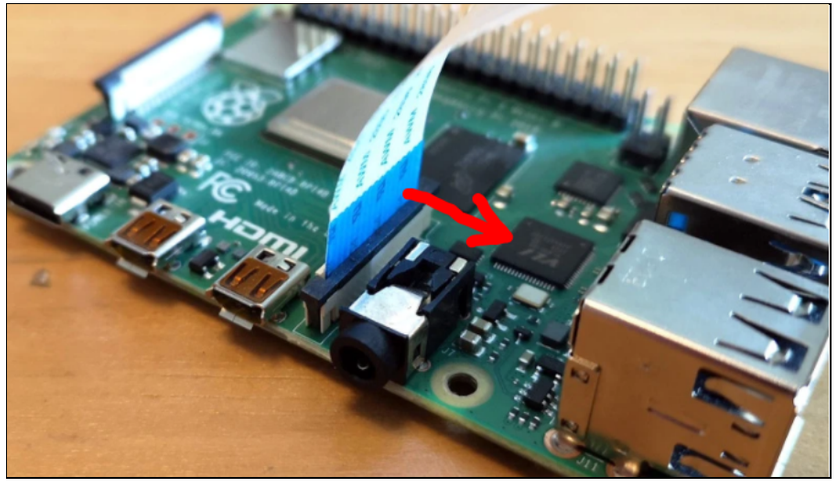
Before you take your Camera Module out of the box, be aware that it can be damaged by static electricity. Make sure you have discharged yourself by touching an earthed object (e.g. a radiator, PC Chassis or similar). Please make sure your Pi is off and unplugged from power supply.

This tutorial shows the NoIR Camera Module (great for night photography projects). The standard Camera Module is green. They are installed and work in the same way.

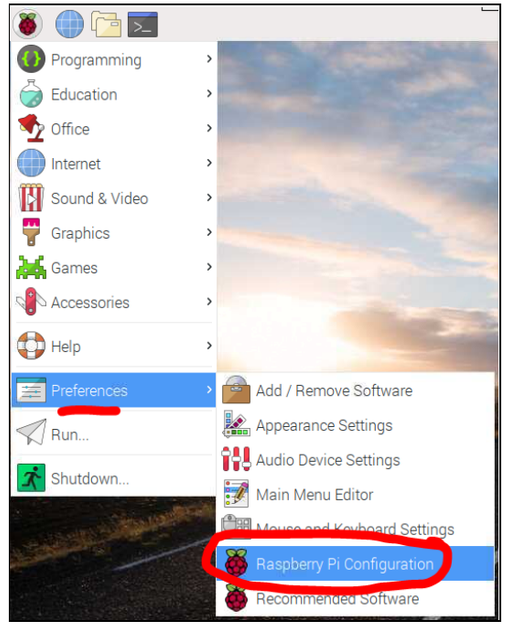


Install the Raspberry Pi Camera module by inserting the cable into the Raspberry Pi camera port. The cable slots into the connector situated between the USB and micro-HDMI ports, with the silver connectors facing the micro-HDMI ports.

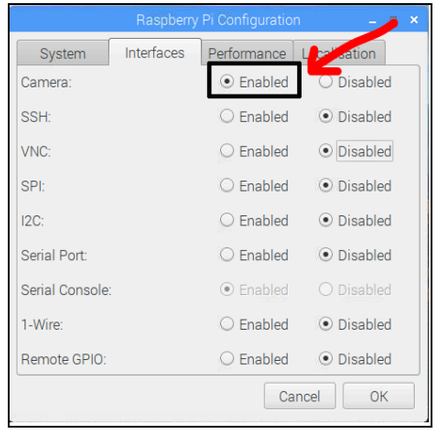
If in doubt, just make sure the blue part of the cable is facing the USB ports on the Raspberry Pi:



Start up your Raspberry Pi. Go to the main menu and open the Raspberry Pi Configuration tool.

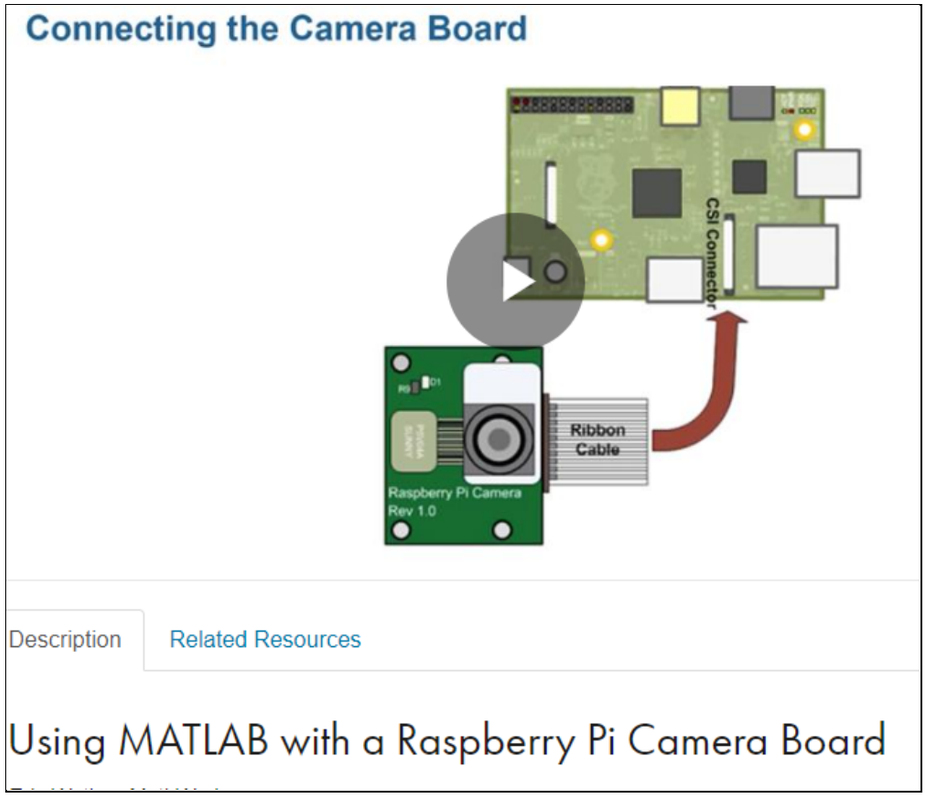


Select the Interfaces tab and ensure that the camera is **Enabled**:



Reboot your Raspberry Pi.

Once your Raspberry Pi has rebooted, please watch this short 6 minute Matlab tutorial about how to connect to it inside Matlab:



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

Vision as a Sensor

If you want your robot to perform a task such as picking up an object, chasing a ball, locating a charging station, etc., and you want a single sensor to help accomplish all of these tasks, then vision is your sensor. Vision (image) sensors are useful because they are so flexible. With the right algorithm, an image sensor can sense or detect practically anything. But there are two drawbacks with image sensors: 1) they output lots of data, dozens of megabytes per second, and 2) processing this amount of data can overwhelm many processors. And if the processor can keep up with the data, much of its processing power won't be available for other tasks.

Purple Narwhals (and other things)

Your Pi-Cam will use a color-based filtering algorithm to detect objects. Color-based filtering methods are popular because they are fast, efficient, and relatively robust. Most of us are familiar with RGB (red, green, and blue) to represent colors. Matlab calculates the color (hue) and saturation of each RGB pixel from the image sensor and uses these as the primary filtering parameters. The hue of an object remains largely unchanged with changes in lighting and exposure. Changes in lighting and exposure can have a frustrating effect on color filtering algorithms, causing them to break. You will build a filtering algorithm that is robust when it comes to lighting and exposure changes.

.**MATLAB Image Processing Background**

Images are represented as grids, or matrices, of picture elements (called pixels). In MATLAB an image typically is represented as a matrix in which each element corresponds to a pixel in the image. Each element that represents a particular pixel stores the color for that pixel. There are two basic ways that the color can be represented:

- True color, or RGB, in which the three color components are stored (red, green, and blue, in that order).
- Index into a colormap: the value stored is an integer that refers to a row in a matrix called a colormap. The colormap stores the red, green, and blue components in three separate columns.

For an image that has m n pixels, the true color matrix would be a three dimensional matrix with the size m n 3. The first two dimensions represent the coordinates of the pixel. The third index is the color component; (:,:,1) is the red, (:,:,2) is the green, and (:,:,3) is the blue component.

The indexed representation instead would be an m n matrix of integers, each of which is an index into a colormap matrix that is the size p 3 (where p is the number of colors available in that particular colormap). Each row in the colormap has three numbers representing one color: first the red, then green, then blue components, as we have seen before. For example,

[1 0 0] is red
[0 1 0] is green
[0 0 1] is blue

The format of calling the image function is: **image(mat)** where the matrix mat is a matrix that represents the colors in an m n image (m n pixels in the image). If the matrix has the size m n, then each element is an index into the current colormap.

The function **imread** can read in an image file, for example a JPEG (.jpg) file. The function reads color images into a three-dimensional matrix.

For Example:

```
>> myimage1 = imread('Fishing_1.JPG');
>> size(myimage1)
ans =
1536 2048 3
```

There is a lot to learn to begin using this powerful set of image processing tools well. For now we will have you focus on some simple ones and leave plenty of room for expansion if you get interested in the Computer Vision space.

MATLAB has a very deep and quite wide collection of image processing functions and can pretty much do any robot sensing application you might be interested in.

Next, let's write your first Pi image processing program.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Please start with your Robot Control Template, save a copy as **SimplePICameraTutorial** and then enter the code below, section by section and we will walk you through collecting and inspecting your first webcam images.Please modify as shown:

SimplePiCameraTutorial2021RevB.mlx

SimplePiCameraTutorial.mlx is a template for future Fun-Robo Raspberry-Pi camera code

This is a simple color object tracking Pi Camera Tutorial for your robots.

This example shows how to use the snapshot function to acquire live images from PI Cameras. To use it you need to first have Loaded the Raspberry Pi Matlab tools onto your laptop. See Tutorial 1

Because of a bug in standard Matlab snapshot.m function, please use snapshotCustom.m function stored in same directory as this code

At end of programming session, please don't forget to shutdown your Pi cleanly by using the follwing at the command line:

```
h = robotPi
h.execute('sudo shutdown -h now')
```

Sign *your names here* and give *the creation date* and *RevB*

```
clc                    % clear command window
clear                  % clear MATLAB workspace
imaqreset              % clear all image objects from memory
```

Set up robot control system (code that runs once)

```
% Create a global raspberry PI object so that it can be used in functions
% Create a *raspi* object. You must be on Olin-Robots network and use your
% pis correct IP address, rplace address in code below with yours
   robotPi = raspi('192.168.10.141')
```

```
robotPi =
  raspi with properties:

        DeviceAddress: '192.168.10.141'
                 Port: 18734
            BoardName: 'Raspberry Pi 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: {'bcm2835-isp (platform:bcm2835-isp)','mmal service 16.1 (platform:bcm2835-v412)
           I2CBusSpeed: 100000

  Supported peripherals
```

Please note: you must use your Pis IP address, not the one shown.

Then start to create the code that runs once.

The next lines of code connect your Pi Cam:

```
% To create a connection to the V2 Pi Camera
robotCam = cameraboard(robotPi,'Resolution','1280x720')
```

```
robotCam =
  cameraboard with properties:

                    Name: Camera Board
              Resolution: '1280x720'      (View available resolutions)
                 Quality: 10              (1 to 100)
                Rotation: 0               (0, 90, 180 or 270)
          HorizontalFlip: 0
            VerticalFlip: 0
               FrameRate: 30              (2 to 90)
               Recording: 0

  Picture settings
              Brightness: 50              (0 to 100)
                Contrast: 0               (-100 to 100)
              Saturation: 0               (-100 to 100)
               Sharpness: 0               (-100 to 100)

  Exposure and AWB
            ExposureMode: 'auto'          (View available exposure modes)
   ExposureCompensation: 0                (-10 to 10)
                AWBMode: 'auto'          (View available AWB modes)
           MeteringMode: 'average'       (View available metering modes)

  Effects
             ImageEffect: 'none'          (View available image effects)
     VideoStabilization: 'off'
                    ROI: [0.00 0.00 1.00 1.00] (0.0 to 1.0 [top, left, width, height])
```

```
% Fix auto exposure problem , set eposure to work in lab lighting
% set whitebalance to manual too. Auto exposure and white balance drive
% computer vison algorythms crazy by constantly changing
SetUpPiCamera(robotCam);
```

You can control many features of your Pi Camera by directly setting teh picture settings, Exposure mode, and effects.

Because auto exposure and auto whitebalance change how the cameras respond to lighting on a random basis, we will go in and set these two properties to try and get a bit more stability for the system. You will do this in the function **SetUpPiCamera.** Please enter code as shown:

Sense Functions (store all Sense related local functions here)

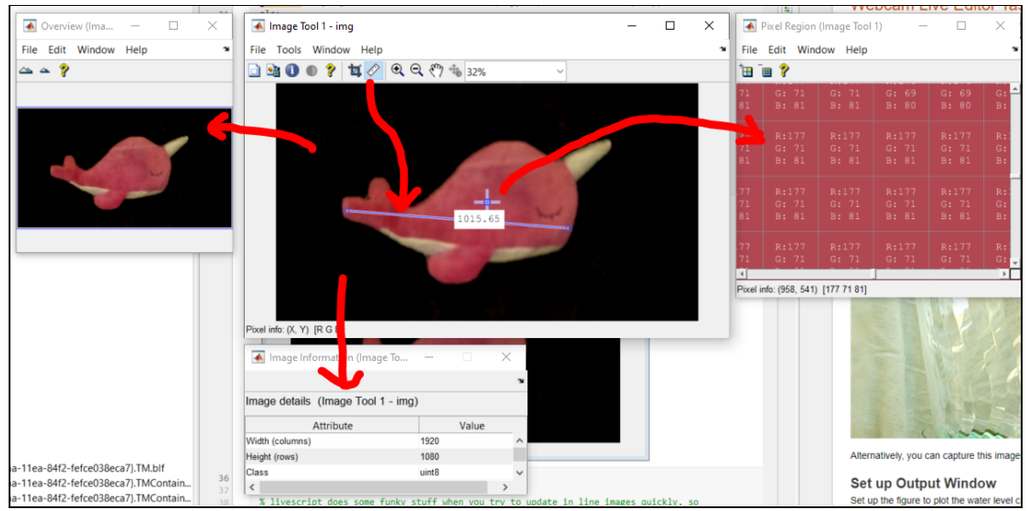
```
71  function []= SetUpPiCamera(robotCam)
72      % creates and configures an Pi V2 Camera to be a simple robot
73      % vision system. It requires a standard Pi V2 cmaera attached to
74      % your Raspberry piand takes the picam object name as sole input
75      % You need to set your cameras unique parameters to optimize picture
76      % D. Barrett 2021 Rev A
77
78      % Fix  exposure set it to respond quickly, set auto whitebalance to off
79      robotCam.ExposureMode = 'sports';
80      robotCam.AWBMode = 'fluorescent';
81  end
82
```

Returning to the main code that runs once, add the following part to take a snapshot of an image and then load it into the **imTool** for inspection.

```
14  % Acquire a Frame
15  img = snapshotCustom(robotCam);
16
17  % Display the frame in the imaqtool window
18  % imaqtool launches an interactive GUI to allow you to explore, configure,
19  % and acquire data from your installed and supported image acquisition devices.
20  imtool(img);
21
22  % explore image with tool
23  gCamTest= input('experiment with tool, type G, then hit ENTER ','s');
24  clc;
```

Please note: We are using the **snapshotCustom** function because the Matlab supplied Snapshot function has a bug in it that does not update new frames for 5 frame calls. Just don't use the standard **snapshot** till they fix it. Make sure that **snapshotCustom** is in the same folder as this code.

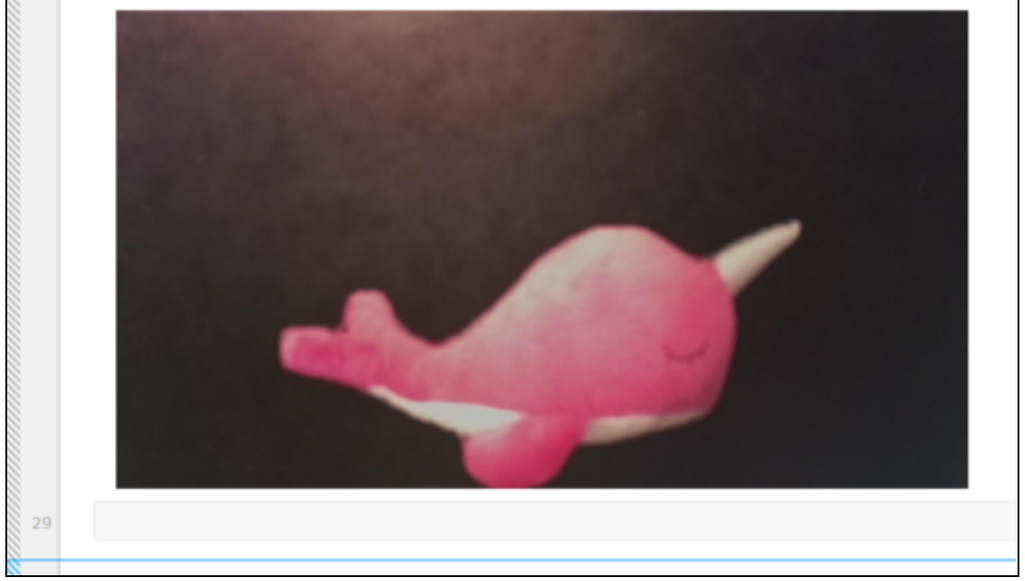
The **imtool** lets you probe the actual RGB pixel values in a region, measure objects on the screen to tell how many pixels they span and the size and content of the image.



These tools let you carefully evaluate an image so that you can tune an object tracking filter later with that knowledge. Play around with the tools until you are familiar with them then type in **G** (for go) followed by the enter key.

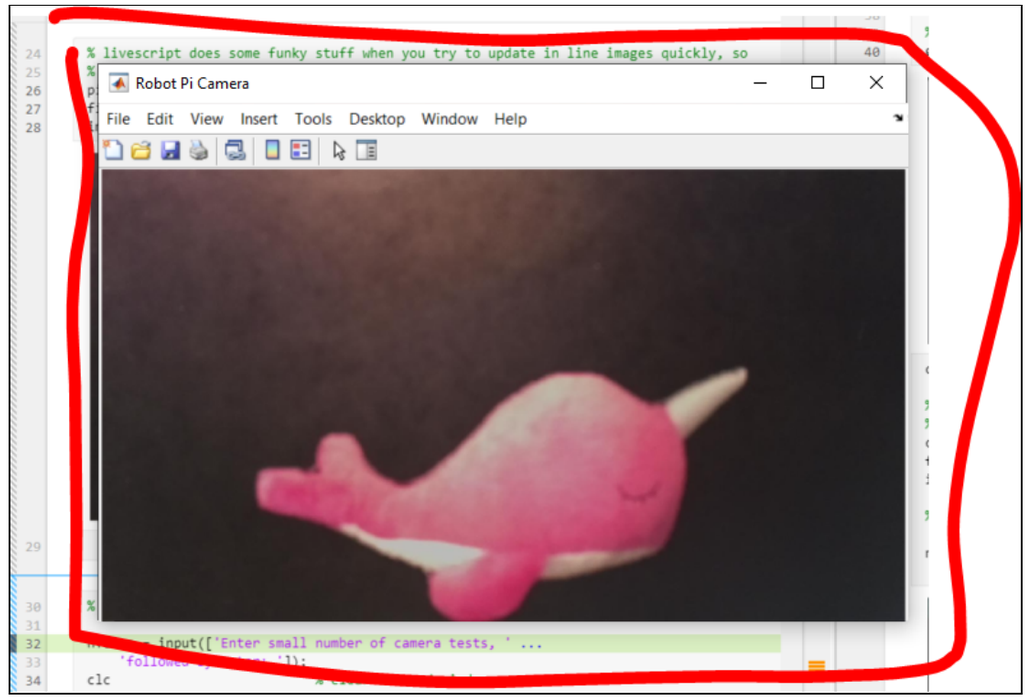
The next set of code creates an external figure to drop images in:

```
% livescript does some funky stuff when you try to update in line images quickly, so
% create a free floating figure for images to deal with livescript update problem
piCamWindow = figure('name','Robot Pi Camera','NumberTitle','off','Visible','on');
figure(piCamWindow)                              % go to camWindow for imshow
imshow(img,'Border','tight')
```



Because of some program oddness in the Java engine behind Live Scripts, it sometimes does weird things with in-line figures in the Matlab code window. To overcome this and to also give you a semi-permanent record of the image after the script completes, this code will create an external stand alone figure for use later in the code.

The figure looks like this:



Next there is a little piece of code that sets up the number of experimental test to do:

```
% Configure test loop to collect n data points.

nTests = input(['Enter small number of camera tests, ' ...
    'followed by enter: ']);
clc                             % clear command window

r = rateControl(0.1);           % create a 0.1 hz loop rate
reset(r);                       % reset loop time to zero
```

Followed by the robot control loop:

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

```
38  controlFlag = 1;
39  % create a loop control
40  while (controlFlag < nTests+1)   % loop till ntests data captured
41
42      piIMG = SENSE(robotCam);         % capture a Pi camera image
43      figure(piCamWindow)              % go to camWindow for imshow
44      imshow(piIMG,'Border','tight')
45
46      THINK();                        % compute what robot should do next
47      ACT();                          % command robot actuators
48
49      % pause to allow you to move objects
50      camtest= input('move object to new position, type G, then hit ENTER ','s');
51      clc;
52
53      waitfor(r);                  % wait for loop cycle to complete
54      controlFlag = controlFlag+1; % increment loop
55  end
```

In this example code, the loop is pretty simple.the THINK and ACT functions are empty placeholders. The SENSE function pulls an image off the USB webcam and there's a little logic to pause the loop to let you move targets between images.

Taking a look at the SENSE function:

```
81  function [piRobotImage] = SENSE(robotCam)
82      % This function aquires a single image from the PiCamera
83      % and displays it in a stand alone figure
84      % D. Barrett 2021 Rev A
85     piRobotImage = snapshotCustom(robotCam);     % take one image
86
87  end
```

It just takes a snapshot of what is in front of the USB camera and returns that as a **robotImage**. The THINK and ACT functions are empty placeholders waiting for future work. It's a good practice for both modularity and clear code structure to populate your code with placeholder functions to be filled in later.

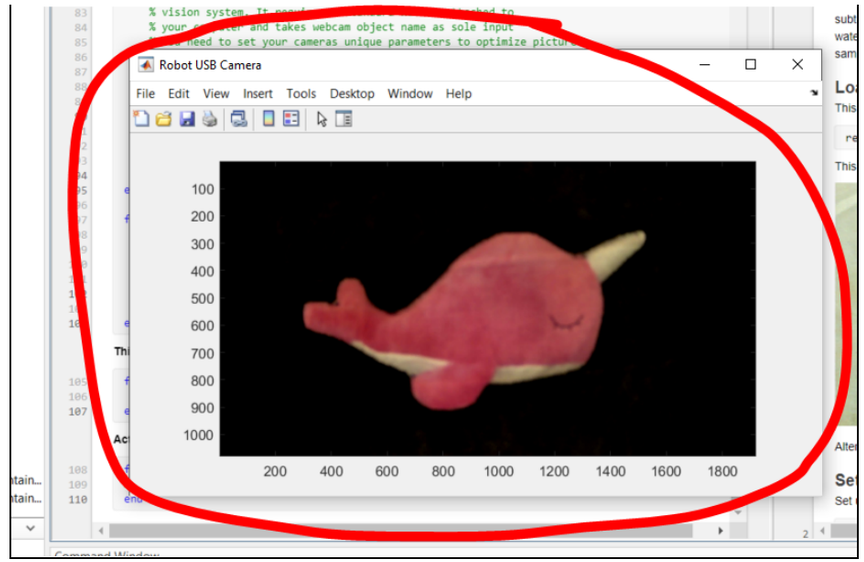
Think Functions (store all Think related local functions here)

```
90  function THINK()
91     % null function, not much thinking to do here.
92  end
```

Act Functions (store all Act related local functions here)

```
93  function ACT()
94  % null function, not much acting to do here.
95  end
```

Running the code will take a snapshot and display it both within the livescript and in a free standing figure called **Robot Pi Camera**



Finally to wrap up your first image procession script add a placeholder section in which to do further processing and a clean shutdown section, that releases the webcam as shown below:

Mission data processing

For many robot applications, you will need to post-process the data collected after the mission. Here we will plot the measured versus actual range positions.

```
% Add post processing code here, mmight be good to downoad images to a
% MATLAB drive location where you can work on processing them latter
```

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 WebCam
% when no longer needed

  clc
  clear robotCam          % connection is no longer needed, clear the cam variable.
  clear robotPi           % clear connection to Pi
  disp('SimplePiCameraTutorial Done');
  beep        % play system sound to let user know program is ended
```

It's very important to release your Raspberry Pi and its Pi cam, so that other applications can communicate with them when you need to.

Run the program a number of times. Modify it a bit to let you save images as files (image.jpg) in your team matlab drive directory. Try a few different targets and against both the black and white backgrounds. When you are working in computer vision, you want to have a lot of trial data image files to try out new algorithms on, without needing to go to the lab and fire up the whole test bench.

After collecting images, it would be good to be able to find and identify multiple different colored targets. Please view this short video tutorial on how to do so:

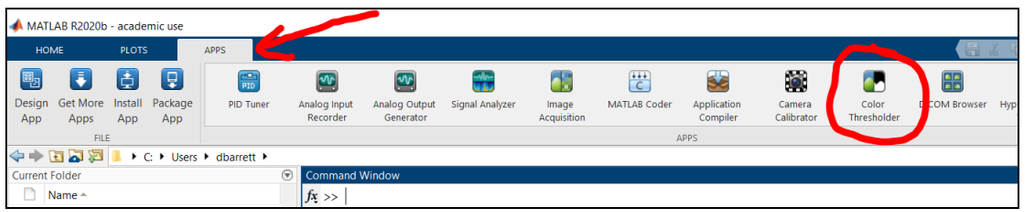
https://www.mathworks.com/videos/color-based-segmentation-with-live-image-acquisition-68771.html

And then, drawing heavily from the tools shown in the above tutorial and with the guide below, let's write a MATLAB script that lets you find and identify the multiple colored targets on the test station cart.

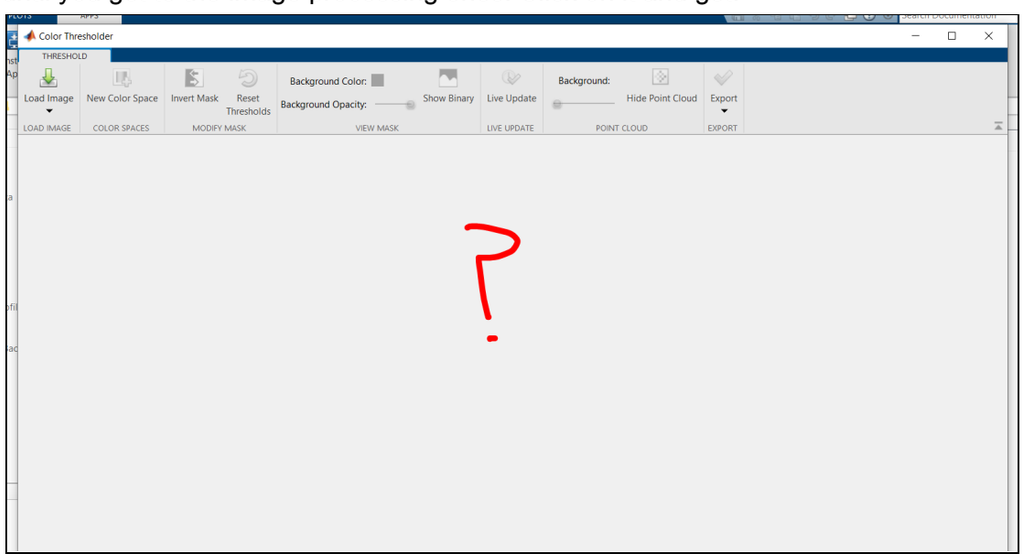
ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Using your existing simple pi camera Tutorial as a code starter, we are going to use one of MATLABS built in computer vision Apps to create a color filter that will segment out and find just objects of the particular color that you choose. It will generate a really cool function that will take your input image and reliably find just those parts of it that have the color you specified.

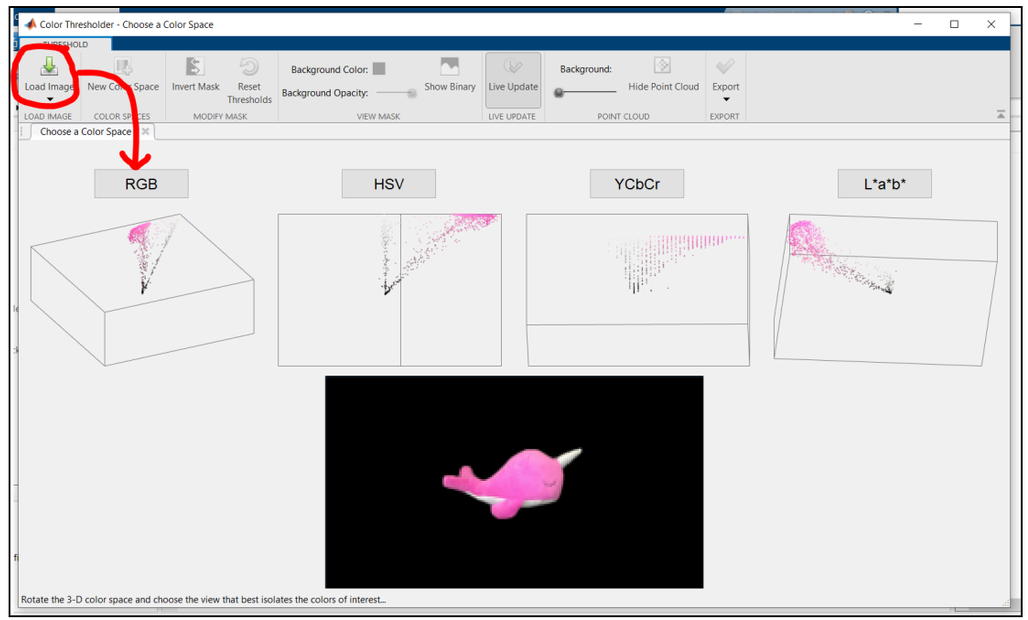
You might want to use this tool multiple times to build several color filters, one for each potential target. Go to **APPS tab** then **ColorThresholder**



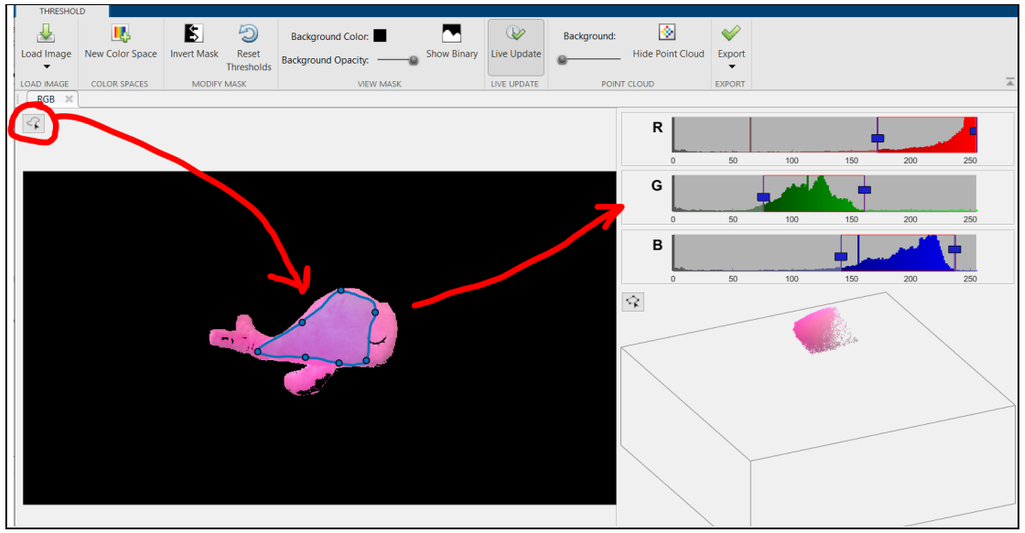
There are a lot of APPS in 2020b, you may need to scan down the full page of APPS until you get to the image processing ones. Click on it and get:



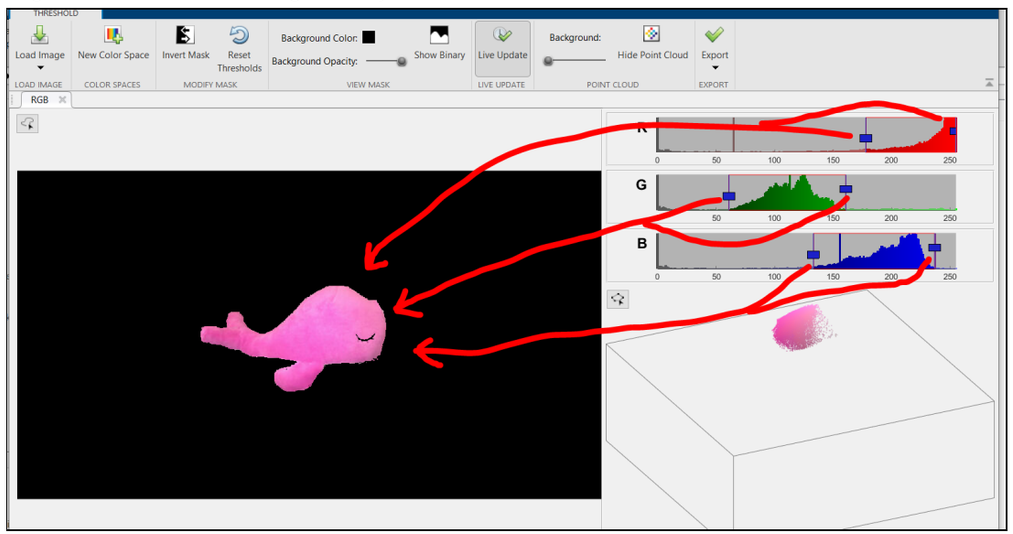
Click on **Load Image** and load one of your saved images from the previous section of this tutorial, it will appear in the center work space:



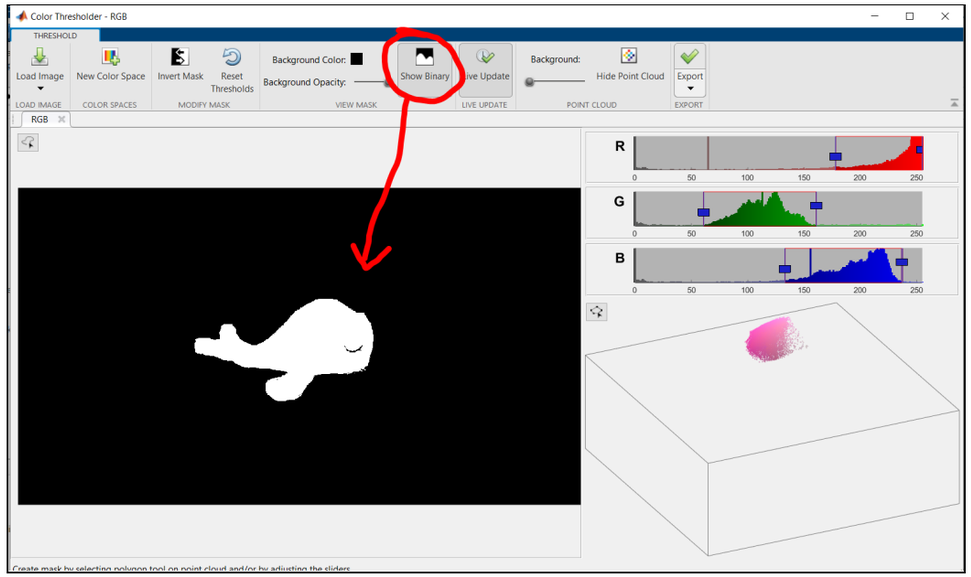
Select RGB color space (we will start out with RGB, but downstream you might find other color spaces let you better distinguish between similar colors) and then use the **lasso tool** in the top left of the image workspace to lasso the colored area of your target.:



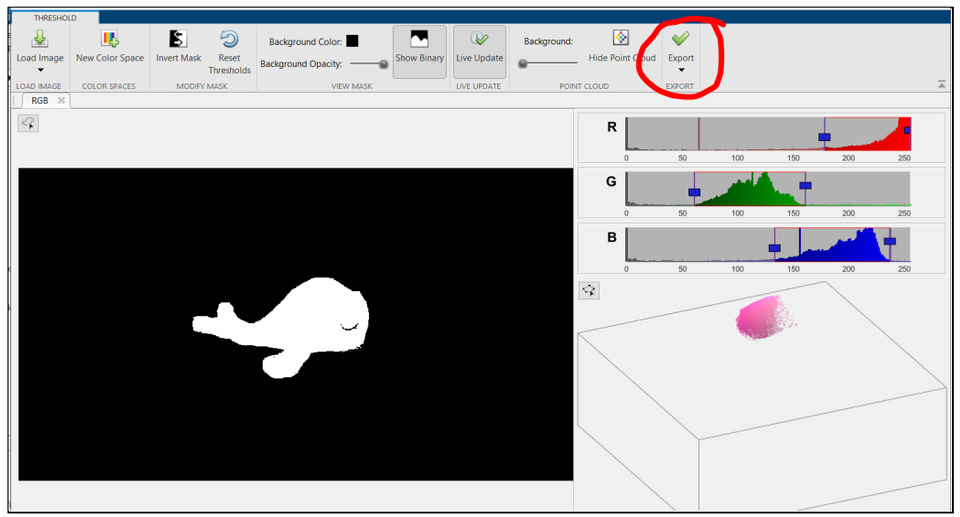
Use the two bar sliders on the R G B tracks to fine tune the filter to capture the best segmentation of the image that you can.



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



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



and the App will drop a new masking function, fully documented into your Editor window.

```
Editor - Untitled*
Untitled*
1  function [BW,maskedRGBImage] = createMask(RGB)
2  %createMask  Threshold RGB image using auto-generated code from colorThresholder app.
3  %  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
4  %  auto-generated code from the colorThresholder app. The colorspace and
5  %  range for each channel of the colorspace were set within the app. The
6  %  segmentation mask is returned in BW, and a composite of the mask and
7  %  original RGB images is returned in maskedRGBImage.
8
9  % Auto-generated by colorThresholder app on 15-Feb-2021
10 %------------------------------------------------------
11
12
13 % Convert RGB image to chosen color space
14 I = RGB;
15
16 % Define thresholds for channel 1 based on histogram settings
17 channel1Min = 178.000;
18 channel1Max = 255.000;
19
20 % Define thresholds for channel 2 based on histogram settings
21 channel2Min = 61.000;
22 channel2Max = 161.000;
23
24 % Define thresholds for channel 3 based on histogram settings
25 channel3Min = 133.000;
```

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

```
Editor - C:\Users\dbarrett\Dropbox\C_Olin Courses 2017\ENGR3390 Robotics\20XX steady state course material\course code\SenselabCode2021\purpleMask.m
purpleMask.m
1  function [BW,maskedRGBImage] = purpleMask(RGB)
2  % purpleMask  Threshold RGB image using auto-generated code from
3  % colorThresholder app.
4  %  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
5  %  auto-generated code from the colorThresholder app. The colorspace and
6  %  range for each channel of the colorspace were set within the app. The
7  %  segmentation mask is returned in BW, and a composite of the mask and
8  %  original RGB images is returned in maskedRGBImage.
9
10 % Auto-generated by colorThresholder app on 15-Feb-2021
11 %------------------------------------------------------
12
13
14 % Convert RGB image to chosen color space
```

Go back and repeat with a few of the other colored targets on the test stand.

Next, take your **SimplePiCameraTutorial** code and modify it as shown:

SimplePiCameraTutorial2021RevB.mlx | ColorShapeTrackerPiCameraTutorial2021RevB.mlx *

ColorShapeTrackerPiCameraTutorial.mlx

A color object tracking PI Camera Tutorial. This example shows how to use the snapshot function to aquire live images from Pi Camera, then find and track colored targets .

Sign *your names here* and give *the creation date* and *revision number*

```
1  clc                    % clear command window
2  clear                  % clear MATLAB workspace
3  imaqreset              % clear all image objects from memory
```

Set up robot control system (code that runs once)

```
4  % Create a global raspberry PI object so that it can be used in functions
5  % Create a *raspi* object. Note: use your pis IP address here
6     robotPi = raspi('192.168.10.141')
7  % To create a connection to the V2 Pi Camera
8  robotCam = cameraboard(robotPi,'Resolution','1280x720')
9  % Fix auto exposure problem , set eposure to work in lab lighting
10 % set whitebalance to manual too. Auto exposure and white balance drive
11 % computer vison algorythms crazy by constantly changing
12 SetUpPiCamera(robotCam);
13
```

Most of the code stays the same:

```
14 % Acquire a Frame
15 img = snapshotCustom(robotCam);
16
17 % Go off and use colorMask APP to make filter based on this image
18 % just load image img from Matlab workspace
19 gCamTest= input('todays filter done?, type G, then hit ENTER ','s');
20 clc;
21
22 % Display the frame in the imaqtool window
23 % imaqtool launches an interactive GUI to allow you to explore, configure,
24 % and acquire data from your installed and supported image acquisition devices.
25 imtool(img);
26
27 % explore image with tool
28 gCamTest= input('experiment with tool, type G, then hit ENTER ','s');
29 clc;
30
31 % livescript does some funky stuff when you try to update in line images quickly, so
32 % create a free floating figure for images to deal with livescript update problem
33 camWindow = figure('name','Robot Pi Camera','NumberTitle','off','Visible','on');
34 figure(camWindow)                          % go to camWindow for imshow
35 imshow(img,'Border','tight')
```

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

We will add a line to pause code after you collect an image of your target and let you jump out to run the color thresholder APP to build a new filter for it. In practice, you will want to build up a set of colorMask filters, probably one for each color target and then you would need to add some additional code to support looking for multiple differently colored targets, For clarity here, we will walk you through how to build just one of them.

The next section of code just sets up multiple tests:

```
39  nTests = input(['Enter small number of camera tests, ' ...
40      'followed by enter: ']);
41  clc                         % clear command window
42
43  r = rateControl(0.1);       % create a 0.1 hz loop rate
44  reset(r);                   % reset loop time to zero
```

The main control loop stays the same with the one modification being a full build out of the SENSE function to return the centroid and area of a colored target. You could use the centroid coordinates to follow the target (or avoid it) and you could use its area, along with a predetermined calibration table, to estimate your distance from it. Please modify your code as shown:

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

```
45  controlFlag = 1;
46  % create a loop control
47  while (controlFlag < nTests+1)   % loop till ntests data captured
48
49      % SENSE finds centroid and area of target
50      [centroids, targetArea] = SENSE(robotCam, camWindow);
51      THINK();                     % compute what robot should do next
52      ACT();                       % command robot actuators
53
54      % pause to allow you to move objects
55      camtest= input('move object to new position, type G, then hit ENTER ','s');
56      clc;
57
58      waitfor(r);                  % wait for loop cycle to complete
59      controlFlag = controlFlag+1; % increment loop
60  end
```

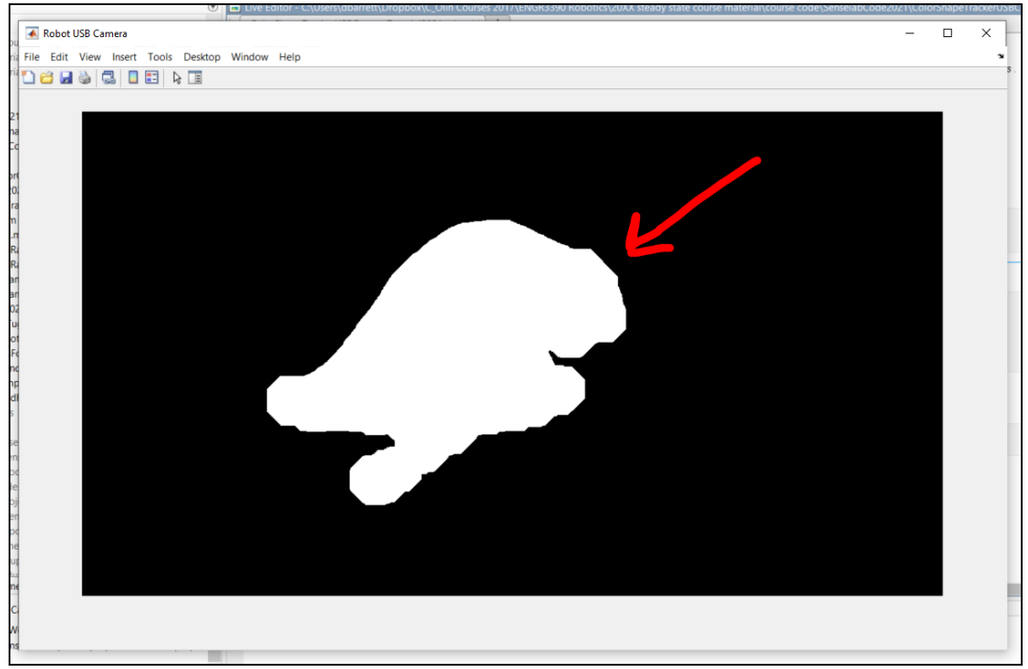
Diving right into the new SENSE function, please modify as we go:

```
104 function [centroids, targetArea] = SENSE(robotCam, camWindow)
105     % This function aquires a single image from the USBCamera testCam
106     % finds and returns centroid of purple area in image
107     % D. Barrett 2021 Rev A
108
109     % capture image
110     robotImage = snapshotCustom(robotCam);
111
112     figure(camWindow)                          % go to camWindow for imshow
113     imshow(robotImage)
114     camtest= input('check out target image, type G, then hit ENTER ','s');
115     clc;
116
117     %% Use Color Threshold App to create a colorMask function (purpleMask)
118     %  and place function in same directory as this script. Your functions
119     %  will have different names depending on color of target you want to
120     %  track, apply colorMask Function to captured image
121     %  this will create two new images
122     % [BW,maskedRGBImage] = purpleMask(RGB)
123     %  BW is binary mask of image
124     %  colorMaskeImg will be the color image inside the mask
125     [targetMask,purpleImg]=purpleMask6(robotImage);
126     figure(camWindow)                          % go to camWindow for imshow
127     imshow(targetMask)
128     camtest= input('check out masked image, type G, then hit ENTER ','s');
129     clc;
```

This function takes a snapshot form the pi cam and stores it in **robotImage.**

After applying the colorMask function to the current image, you get back two images. The targetMask one is a black and white mask of the filtered thresholded target that we will use for all the subsequent image processing.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d



You will take that BW image and remove much of the noise from it:

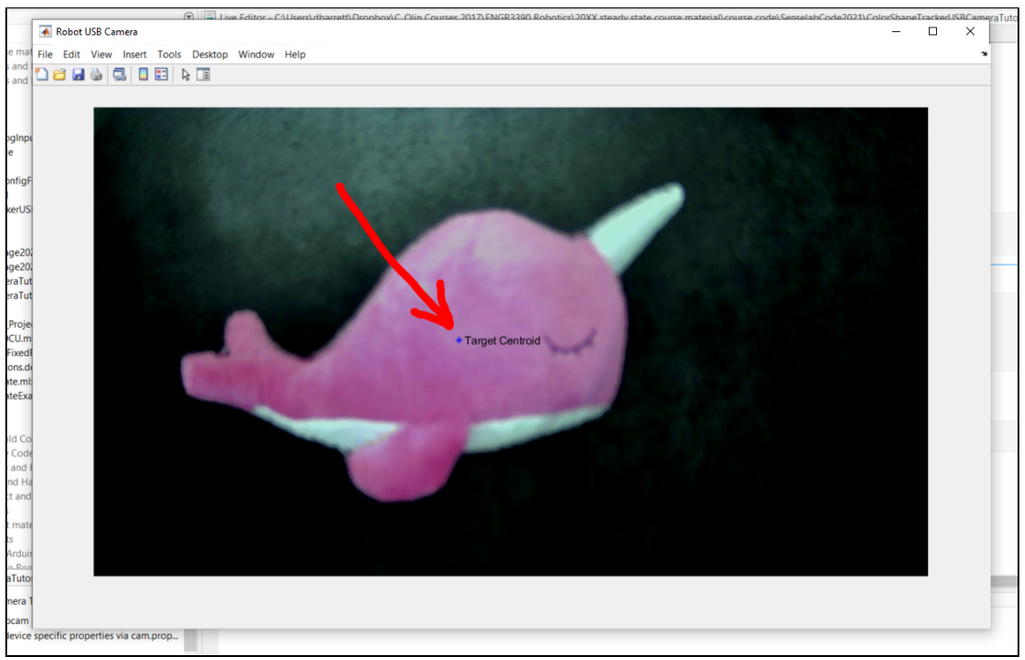
```
%% Preprocess image to remove noise
se = strel('disk',40);                  % structured element erosion function
cleanImage= imopen(targetMask, se);       % Morphologically open image
figure(camWindow)                           % go to camWindow for imshow
imshow(cleanImage)
camtest= input('check out cleaned image, type G, then hit ENTER ','s');
clc;
```

Then used that cleaned up image **cleanImage** to find the centroid of that colored target:

```
%% Calculate the centroid of the white part of masked area (target)
targetCenter = regionprops(cleanImage,'centroid');
% Store the x and y coorinates in a two column matrix
centroids = cat(1,targetCenter.Centroid);
if isempty(centroids)                          % if no target found
      centroids =[0 0];                        % place centroids at 0 0
end
```

Then display that centroid on original image

```
146
147     % Display the original image with the centroid locations superimposed.
148     figure(camWindow)                           % go to camWindow for imshow
149     imshow(robotImage)
150     hold on
151     plot(centroids(:,1),centroids(:,2),'b*')
152     text(centroids(:,1),centroids(:,2), '  Target Centroid');
153     hold off
154     camtest= input('check out target center, type G, then hit ENTER ','s');
155     clc;
156
```



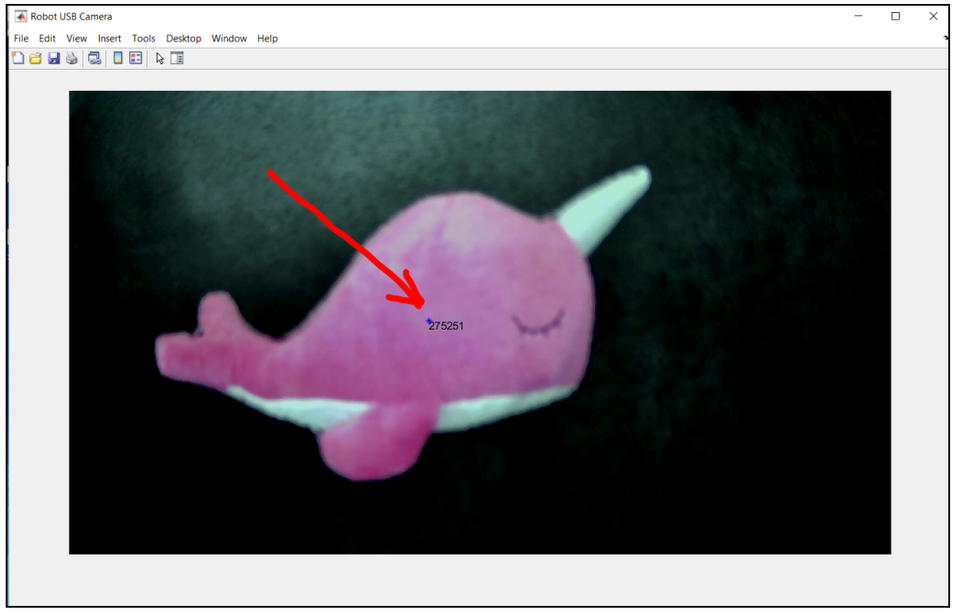
Having found the centroid, the code draws and labels its calculated location onto the original image. In image processing it is always good to do this, namely to draw your final results onto the original image as both a sanity check and as a way of understanding how the algorithms work under a variety of lighting conditions. As an experiment try what happens with lights turned low or turned off. How well does your target finder work?

The SENSE function returns the coordinates of the target back up to the main program. In a full robot control system (like the THINK lab), you could use the x-component of this centroid to steer the robot Tug toward the NarWhal.

In addition to the centroid, please add the following code to calculate the perceived area of the target:

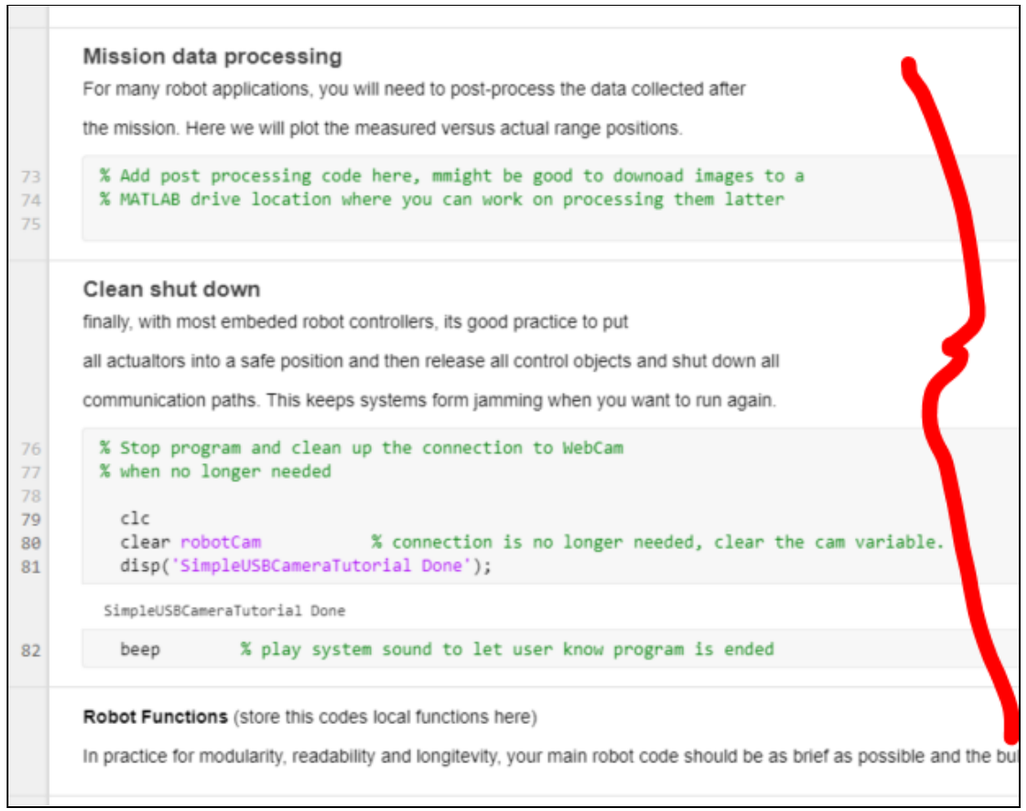
```
156
157     %%  Calculate the area of the target
158     targetArea = regionprops(cleanImage,'area');
159     if isempty(targetArea)                          % if no target found
160           area = 0;                   % place centroids at 0 0
161     else
162     area=targetArea.Area;
163     end
164     % Store the  x and y coorinates in a two column matrix
165     centroids = cat(1,targetCenter.Centroid);
166     if isempty(centroids)                          % if no target found
167           centroids =[0 0];                   % place centroids at 0 0
168     end
169     % Display the binary image with the centroid locations superimposed.
170     figure(camWindow)                                % go to camWindow for imshow
171     imshow(robotImage)
172     hold on
173     plot(centroids(:,1),centroids(:,2),'b*')
174     text(centroids(:,1),(centroids(:,2)+ 10),num2str(area));
175     hold off
176     camtest= input('check out target area, type G, then hit ENTER ','s');
177     clc;
178
179 end
```

Generating this:



As stated before, with a bit of experimental distance versus area measurements, you could get a rough sense of distance from the object by its perceived area. As you get closed it gets bigger. You could use this information to avoid running into it (driving into NarWhals is just bad and probably illegal) !

The remaining code stays unchanged and is shown here for completeness:



Mission data processing

For many robot applications, you will need to post-process the data collected after the mission. Here we will plot the measured versus actual range positions.

```
% Add post processing code here, mmight be good to downoad images to a
% MATLAB drive location where you can work on processing them latter
```

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 WebCam
% when no longer needed

  clc
  clear robotCam          % connection is no longer needed, clear the cam variable.
  disp('SimpleUSBCameraTutorial Done');
```

```
SimpleUSBCameraTutorial Done
```

```
  beep        % play system sound to let user know program is ended
```

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bu

Modify your code, run section by section and make sure you understand what each part does. This code will get you to the point of finding one target. Your next steps will be to extend the color filters to find other targets. As a challenge task, consider how you find a white target against a white background ?

Please see an instructor or Ninja for help as needed.

Finally, Turning Your Raspberry Pi Hardware Safely On and Off

Raspberry Pi runs a Linux operating system. Turning off the power while MATLAB is running can result in corrupting the operating system image on the SD card. To turn off your board safely, first shut down the Linux operating system by executing the following MATLAB commands from the MATLAB command line:

```
>>system(robotPi, 'sudo shutdown -h now');
>>clear robotPi
```

Final Demo: Using your newly acquired computer vision skills, please prepare two final demos:

Find Target:

1. Ninjas will place 1 fixed colored target on the test track rail. Please write a demo and application that correctly finds direction and distance to the colored target.
2. Ninjas will move 3 targets to a second set of locations. Please write and demo code that can correctly identify bearing and range to target as well as identify which target it is.
3. Stretch goal. Write demo code to track and give range and bearing to a moving target. Ninja will slowly rotate one target in the field of view of camera in front of white background

Find Hole:

1. Given a single fixed colored target, find the largest hole (unobstructed area) in the field of view and give a clean bearing to its center (Open water at 56 degrees).
2. Given a field of many fixed colored targets, find the largest hole to drive through and give range and bearing to its center (Open water at 76 degrees).
3. Stretch Goal. Given a field of slowly moving targets (via Ninja), find the largest hole to drive through and give range and bearing to its center (Open water at 45 degrees).

If you have time you can take on a stretch goal and try to build your code into an App with the App building skills you acquired in the MATLAB tutorial. As always, please see an instructor or Ninjas for help with any part of the above.

Figure: Pi-Cam test station

USB Web Camera

The Olin robotics lab uses the Microsoft Lifetouch Cinema USB camera as its high performance, low cost vision system camera of choice:

Figure: Lifetouch USB camera

After extensively searching through low cost USB camera options, this one was chosen for its high-quality 720p HD widescreen video together with crystal clear audio. The camera automatically sharpens your images and TrueColor adjusts exposure for bright, vibrant footage. And for even sharper video it has the high-precision glass lens to improve the picture even in low light conditions.

Specifications:
- 720p HD video chat For a true HD-quality experience.
-Auto focus Images stay sharp and detailed, even close-ups.
-High-precision glass element lens Provides sharp image quality.
-TrueColor technology with face tracking
-Automatically controls exposure for bright and colorful video.
-360-degree rotation rotates halfway in both directions for an all-around view.
-Wideband microphone for premium sound recordings:

In this lab you will find a Lifetouch USB camera mounted in a robot boat perception suite, surrounded by 6 SharpIR range sensors. You can use this system to both find targets and holes by color and also find range to image depth information by careful use of the known area of the target and projected geometry..

Figure: LifeTouch Camera Station

MATLAB Image Processing Background
Images are represented as grids, or matrices, of picture elements (called pixels). In MATLAB an image typically is represented as a matrix in which each element corresponds to a pixel in the image. Each element that represents a particular pixel stores the color for that pixel. There are two basic ways that the color can be represented:
■ True color, or RGB, in which the three color components are stored (red, green, and blue, in that order).
■ Index into a colormap: the value stored is an integer that refers to a row

in a matrix called a colormap. The colormap stores the red, green, and blue components in three separate columns.

For an image that has m n pixels, the true color matrix would be a three dimensional matrix with the size m n 3. The first two dimensions represent the coordinates of the pixel. The third index is the color component; (:,:,1) is the red, (:,:,2) is the green, and (:,:,3) is the blue component.

The indexed representation instead would be an m n matrix of integers, each of which is an index into a colormap matrix that is the size p 3 (where p is the number of colors available in that particular colormap). Each row in the colormap has three numbers representing one color: first the red, then green, then blue components, as we have seen before. For example,

[1 0 0] is red
[0 1 0] is green
[0 0 1] is blue

The format of calling the image function is: **image(mat)** where the matrix mat is a matrix that represents the colors in an m n image (m n pixels in the image). If the matrix has the size m n, then each element is an index into the current colormap.

The function **imread** can read in an image file, for example a JPEG (.jpg) file. The function reads color images into a three-dimensional matrix.

For Example:

```
>> myimage1 = imread('Fishing_1.JPG');
>> size(myimage1)
ans =
1536 2048 3
```

There is a lot to learn to begin using this powerful set of image processing tools well. For now we will have you focus on some simple ones and leave plenty of room for expansion if you get interested in the Computer Vision space.

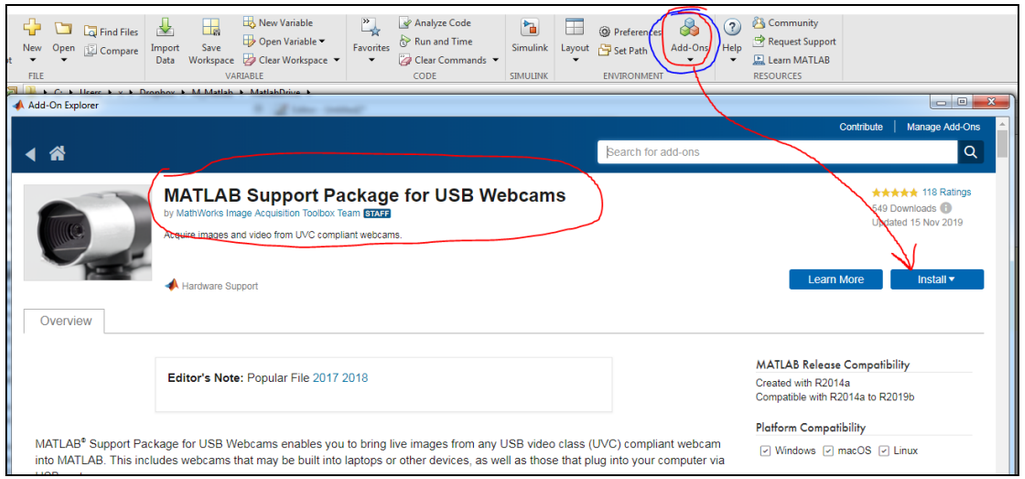
MATLAB has a very deep and quite wide collection of image processing functions and can pretty much do any robot sensing application you might be interested in.

Before moving into image processing, you are going to need to get your lab's USB camera hooked up as well as get access to the encyclopedia of image processing functions MATLAB has to offer.

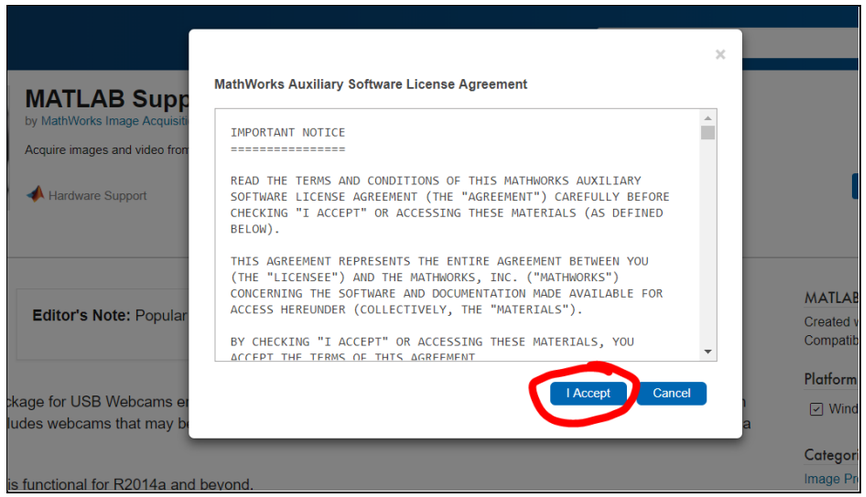
First, Plug the Microsoft webcam in and then wait for it to load the USB driver (may take a few minutes) and follow the on screen instructions to get it hooked up and added to your WIndows operating system

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

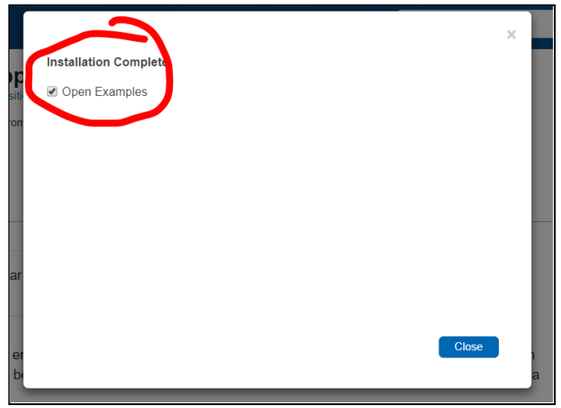
Next go up to the top MATLAB **Home** menu, click on **Add-Ons, Get Add-ons,** search for and then find the **MATLAB Support Package for USB Webcams:**



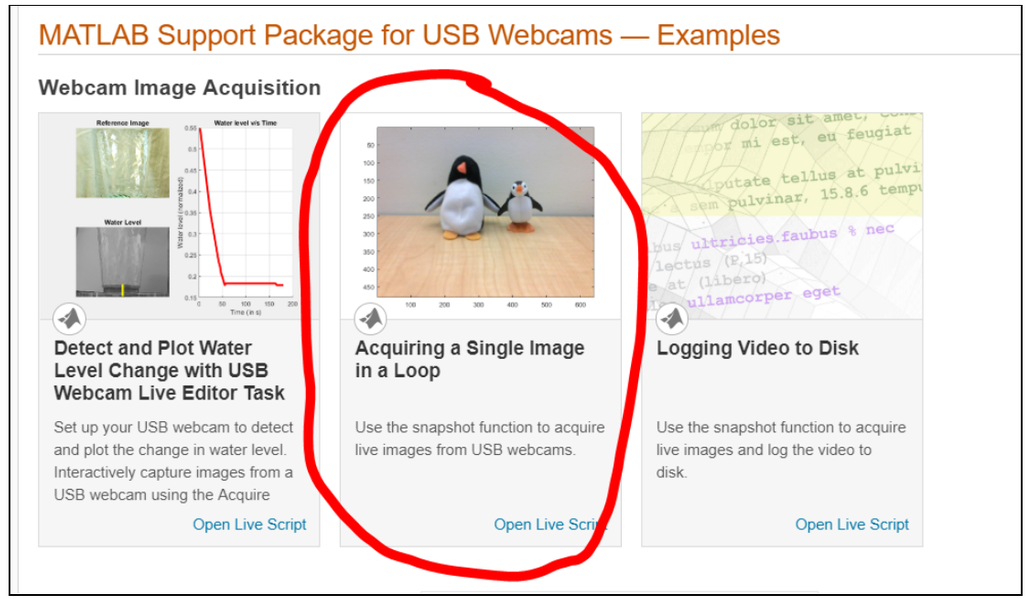
Click on the Blue Install button. Accept Auxiliary Software License agreement.



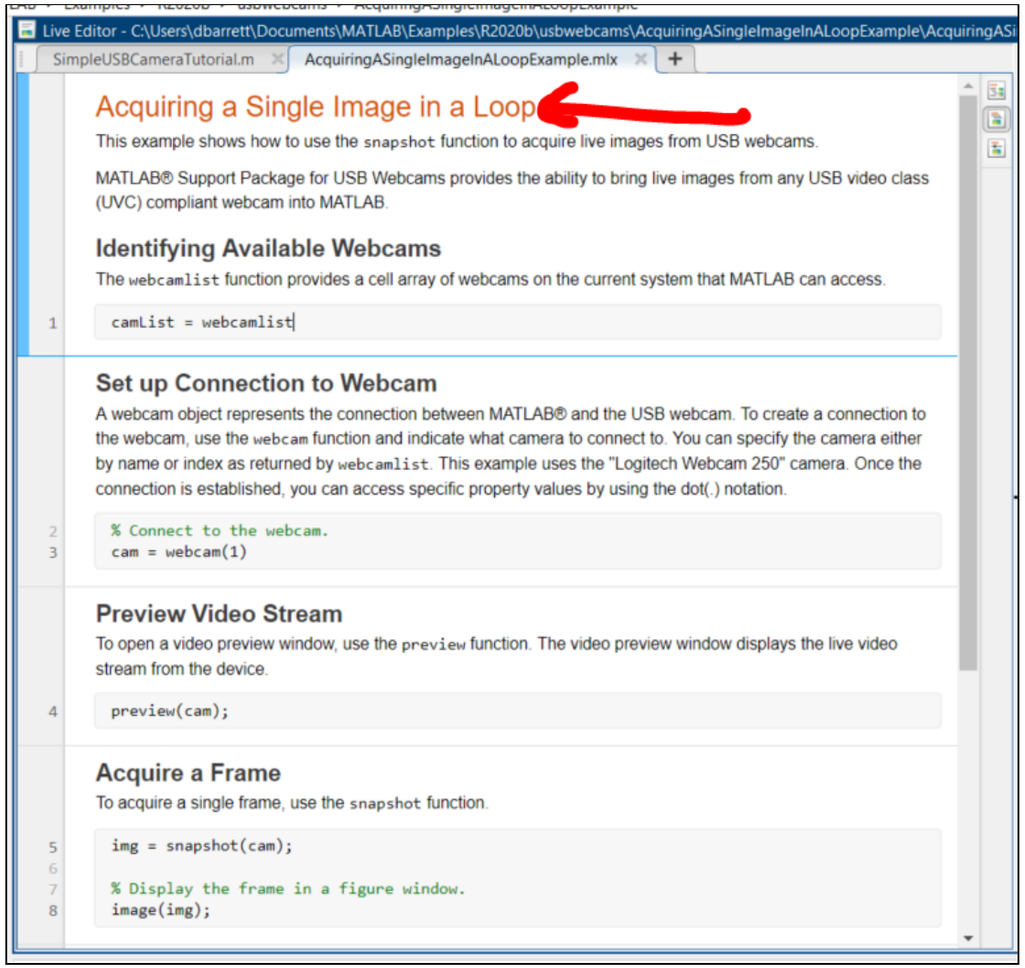
Wait a bit for it to download and install the USB camera package. Open up the Examples:



And get to three useful short Tutorial Examples, walk through center one:

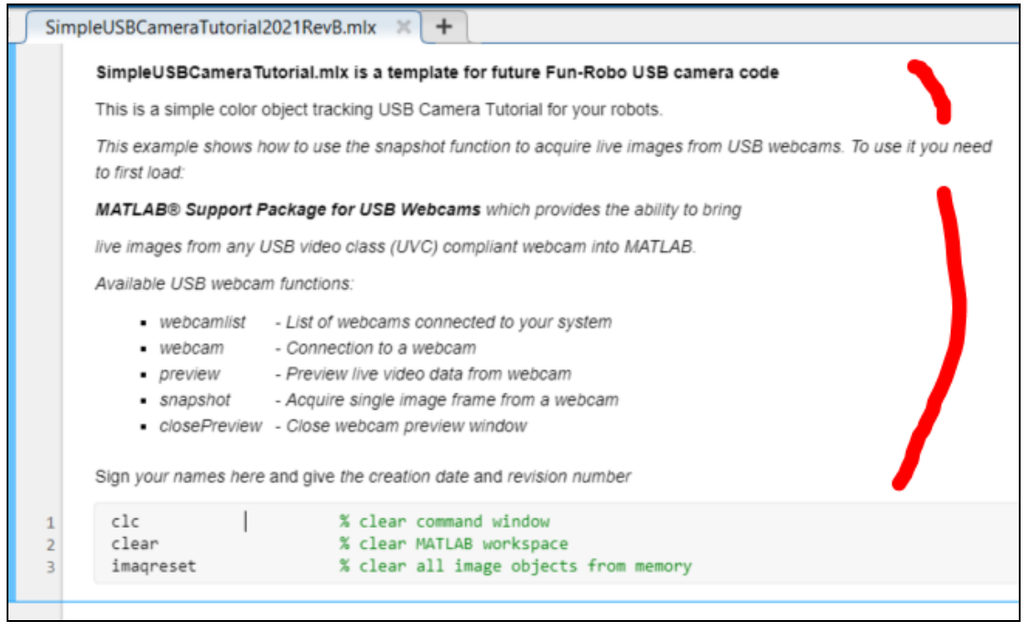


ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

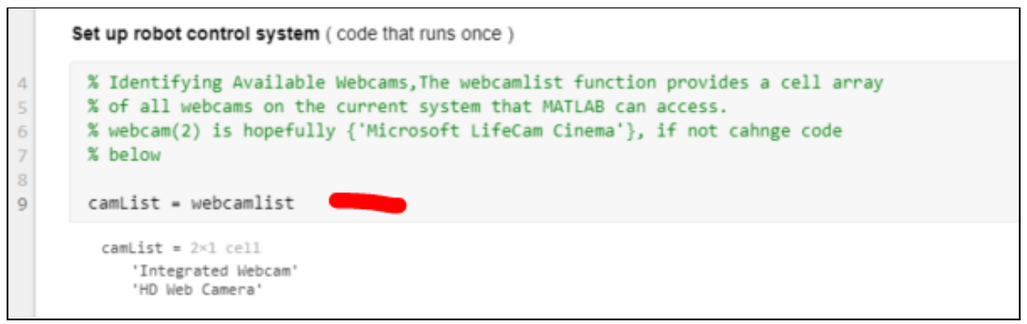


Recommend running in debugger mode, line by line, to get a good grasp of what it is doing and how all of the functions work. This short script will connect you to your (or the labcart's) webcamera, take a few pictures and store them.

Next, let's write your first image processing program. Please start with your Robot Control Template, save a copy as **SimpleUSBCameraTutorial** and then enter the code below, section by section and we will walk you through collecting and inspecting your first webcam images.Please modify as shown:



The next lines of code, interrogate your laptop to see what cameras are connected (you should have 2, your built in one [1] and the LifeCam [2]).



The following will establish a connection to the LifeCam. Please make sure you chose the lifecam or you will be taking images of your own face !

Set up robot control system (code that runs once)

```
% Identifying Available Webcams,The webcamlist function provides a cell array
% of all webcams on the current system that MATLAB can access.
% webcam(2) is hopefully {'Microsoft LifeCam Cinema'}, if not cahnge code
% below

camList = webcamlist
```

```
camList = 2×1 cell
    'Integrated Webcam'
    'HD Web Camera'
```

```
% Set up Connection to the USB Webcam, print its properties
robotCam = webcam(2)
```

```
robotCam =
  webcam with properties:

                    Name: 'HD Web Camera'
    AvailableResolutions: {'1920x1080'  '1280x960'  '1280x720'  '800x600'  '640x480'  '640x360'}
              Resolution: '1920x1080'
              Saturation: 128
                Contrast: 128
           WhiteBalance: 417
              Brightness: 100
      WhiteBalanceMode: 'auto'
  BacklightCompensation: 0
                     Hue: 128
               Sharpness: 128
```

Each different brand of USB webcam has a different set of properties that either define it or can be set by an external program. The above properties come from a different camera than the test station. The test station's LifeCam properties are shown below:

```
%  Name: 'Microsoft LifeCam Cinema'
%    AvailableResolutions: {1×12 cell}
%              Resolution: '640x480'
%                   Focus: 16
%                Contrast: 5
%            ExposureMode: 'manual'
%              Saturation: 83
%                Exposure: -6
%             FocusMode: 'auto'
%                  Zoom: 0
%             Sharpness: 25
%                   Pan: 0
%            Brightness: 133
%                  Tilt: 0
%           WhiteBalance: 4500
%       WhiteBalanceMode: 'manual'
%       BacklightCompensation: 1
```

This next section lets you set up your LifeCam for computer vision work. In order for it to work well as a webchat device, it will automatically do things like white balance, and change exposure levels to adapt to changing light conditions in your dorm room. But these are two things you really don't want changing while you try to do image segmentation, etc. in computer vision code. Having these properties change on the fly will break all of your algorithms. Having created a camera object called **cam** you can reach in an set its properties via the object notation **cam.property = something** In this case we want the **WhiteBalance** and **ExposureMode** to stay fixed.

You will set up your USB camera via a function called **SETUPUSBCAMERA**:

```
% Fix auto exposure problem set it to manual, set eposure to work in lab lighting
% set whitebalance to manual too. Auto exposure and white balance drive
% computer vison algorythms crazy by constantly changing
SETUPUSBCAMERA(robotCam);

% Preview Video Stream
preview(robotCam)
% wave hand in front of lens to make sure camera is working
gCamTest= input('move hand in front of camera, type G, then hit ENTER ','s');
clc;
```

In the following example. The Function will set up the camera used for this tutorial, you will need to edit that a bit by removing and placing comet symbols to get it setup for the LifeCam on the SENSE teststand. Please enter code as shown:

Sense Functions (store all Sense related local functions here)

```
81  function []= SETUPUSBCAMERA(robotCam)
82      % SETUPUSBCAMERA creates and configures an Webcam to be a simple robot
83      % vision system. It requires a standard Webcam attached to
84      % your computer and takes webcam object name as sole input
85      % You need to set your cameras unique parameters to optimize picture
86      % D. Barrett 2021 Rev A
87
88      % Fix auto exposure set it to manual, set whitebalance to manual too
89      % robotCam.ExposureMode = 'manual';      % for LifeCam
90      robotCam.WhiteBalanceMode = "manual";
91      % experimentally determing best exposure setting for lab, enter here
92      %robotCam.Exposure = -8 ;                % for LifeCam
93      %robotCam.WhiteBalanceMode = 'manual' ;  % for LifeCam
94      robotCam.Brightness = 100;
95  end
```

Having reconfigured the USB webcam to not auto adjust either white balance or exposure level, the next section of code creates a viewing window and show you a live video feed from your LifeCam:

```
15  SETUPUSBCAMERA(robotCam);
16
17  % Preview Video Stream
18  preview(robotCam)
19  % wave hand in front of lens to make sure camera is working
20  gCamTest= input('move hand in front of camera, type G, then hit ENTER ','s');
21  clc;
22
23  % close Preview window
24  closePreview(robotCam);
```

It will open a preview window, let you wave your hand in front of a live video feed from the camera to let you know the camera is connected and is working.

When all is okay, you can type a **G (for go)** into the command window and teh program will close the window and proceed.

If your live camera feed is too light or too dark you will need to go change the value in the **SETUPUSBCAMERA**: robotCam.exposure = n (more negative makes it darker, more positive lighter) to get a good saturated color image:

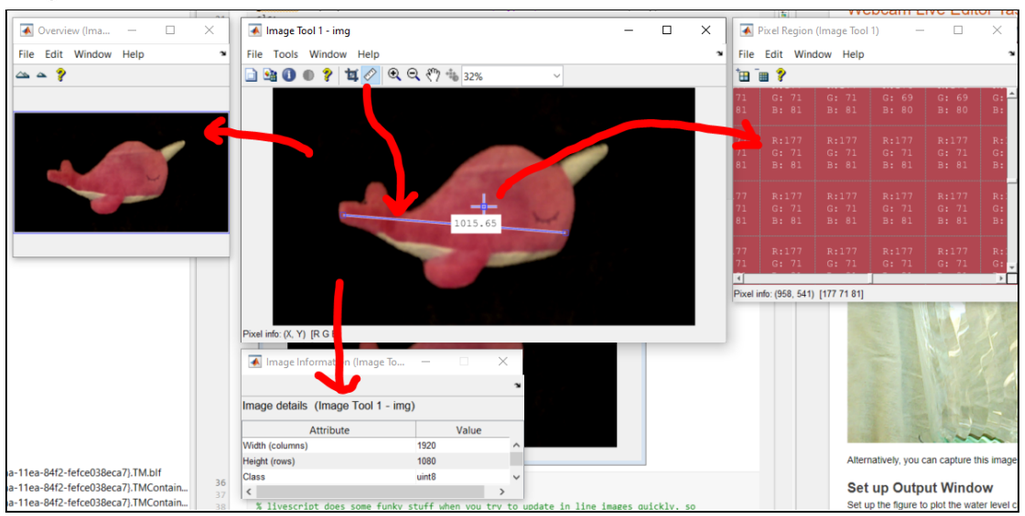
Sense Functions (store all Sense related local functions here)

```
81  function []= SETUPUSBCAMERA(robotCam)
82      % SETUPUSBCAMERA creates and configures an Webcam to be a simple robot
83      % vision system. It requires a standard Webcam attached to
84      % your computer and takes webcam object name as sole input
85      % You need to set your cameras unique parameters to optimize picture
86      % D. Barrett 2021 Rev A
87
88      % Fix auto exposure set it to manual, set whitebalance to manual too
89      % robotCam.ExposureMode = 'manual';      % for LifeCam
90      robotCam.WhiteBalanceMode = "manual";
91      % experimentally determing best exposure setting for lab, enter here
92      %robotCam.Exposure = -8 ;                % for LifeCam
93      %robotCam.WhiteBalanceMode = 'manual' ;  % for LifeCam
94      robotCam.Brightness = 100;
95  end
```

Once you get the live frame looking right, the next section acquires a single frame and stores it in the MATLAB workspace and loads it into the **imtool utility** so that you can do an in depth exploration of it :

```
25
26    % Acquire a Frame
27    img = snapshot(robotCam);
28
29    % Display the frame in the imaqtool window
30    % imaqtool launches an interactive GUI to allow you to explore, configure,
31    % and acquire data from your installed and supported image acquisition devices.
32    imtool(img);
33
34    % explore image with tool
35    gCamTest= input('experiment with tool, type G, then hit ENTER ','s');
```

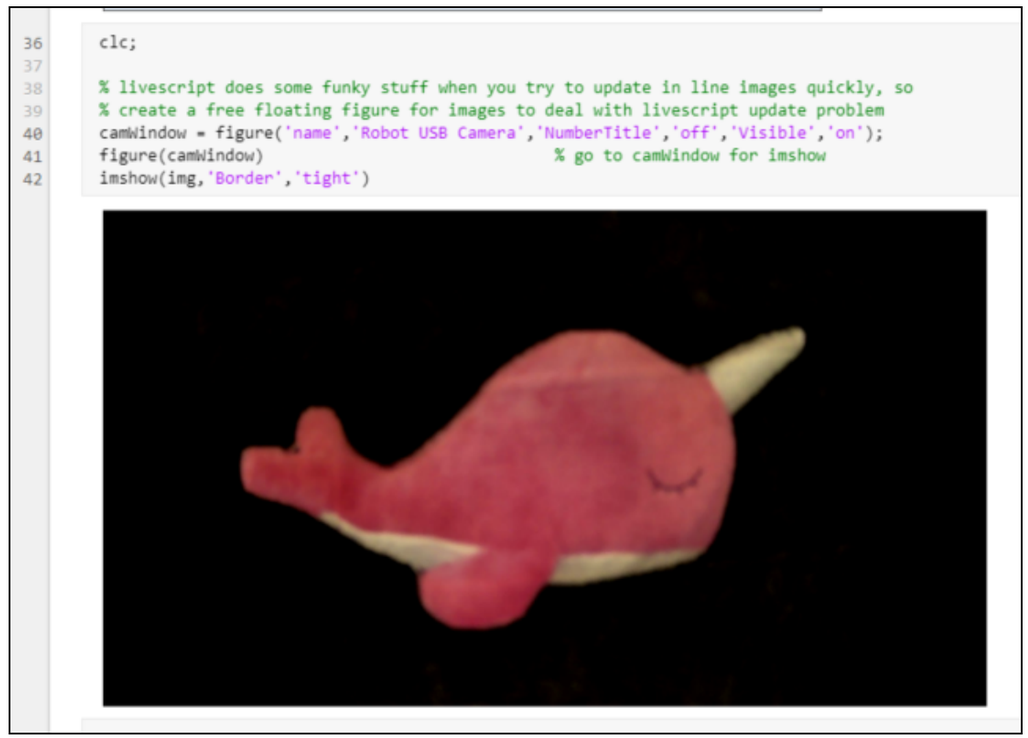
The **imtool** lets you probe the actual RGB pixel values in a region, measure objects on the screen to tell how many pixels they span and the size and content of the image.



Play around with the tools until you are familiar with them then type in **G** (for go) followed by the enter key.

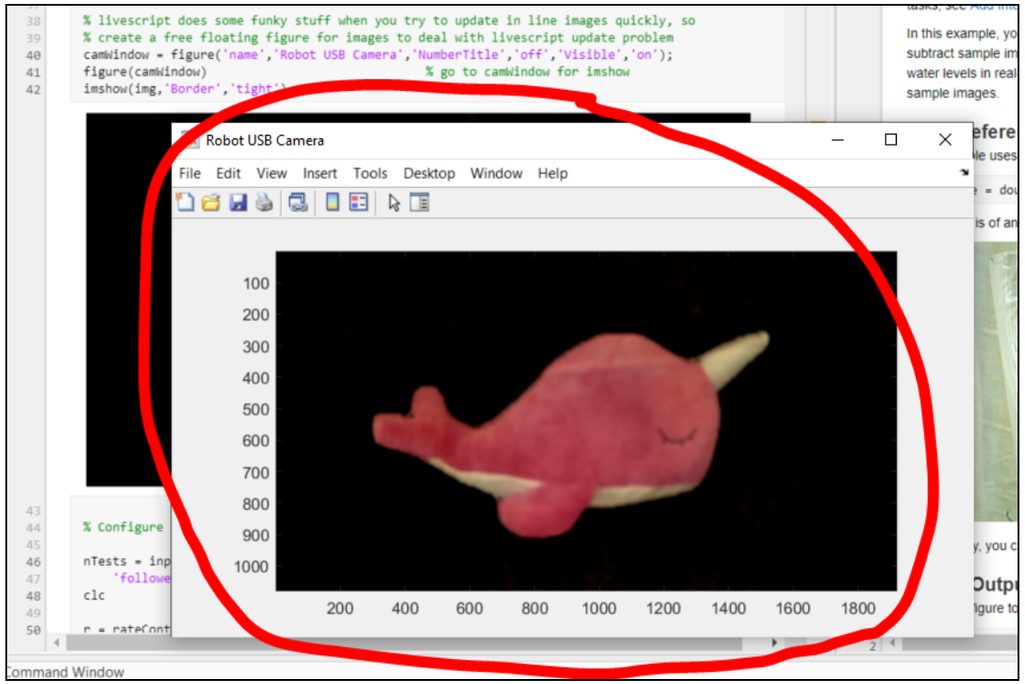
The next set of code creates an external figure to drop images in:

```
36    clc;
37
38    % livescript does some funky stuff when you try to update in line images quickly, so
39    % create a free floating figure for images to deal with livescript update problem
40    camWindow = figure('name','Robot USB Camera','NumberTitle','off','Visible','on');
41    figure(camWindow)                          % go to camWindow for imshow
42    imshow(img,'Border','tight')
```



Because of some program oddness in the Java engine behind Live Scripts, it sometimes does weird things with in-line figures in the Matlab code window. To overcome this and to also give you a semi-permanent record of the image after the script completes, this code will create an external stand alone figure for use later in the code.

The figure looks like this:



Next there is a little piece of code that sets up the number of experimental test to do:

```
% Configure test loop to collect n data points.

nTests = input(['Enter small number of camera tests, ' ...
    'followed by enter: ']);
clc                         % clear command window

r = rateControl(0.1);       % create a 0.1 hz loop rate
reset(r);                   % reset loop time to zero
```

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

Followed by the robot control loop:

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

```
controlFlag = 1;
% create a loop control
while (controlFlag < nTests+1)   % loop till ntests data captured

    robotIMG = SENSE(robotCam);    % capture a USB camera image
    figure(camWindow)              % go to camWindow to show new image with imshow
    image(robotIMG)                % show image in camWindow figure

    THINK();                       % compute what robot should do next
    ACT();                         % command robot actuators

    % pause to allow you to move objects
    camtest= input('move object to new position, type G, then hit ENTER ','s');
    clc;

    waitfor(r);                  % wait for loop cycle to complete
    controlFlag = controlFlag+1; % increment loop
end
```

In this example code, the loop is pretty simple.the THINK and ACT functions are empty placeholders. The SENSE function pulls an image off the USB webcam and there's a little logic to pause the loop to let you move targets between images. Taking a look at the SENSE function:

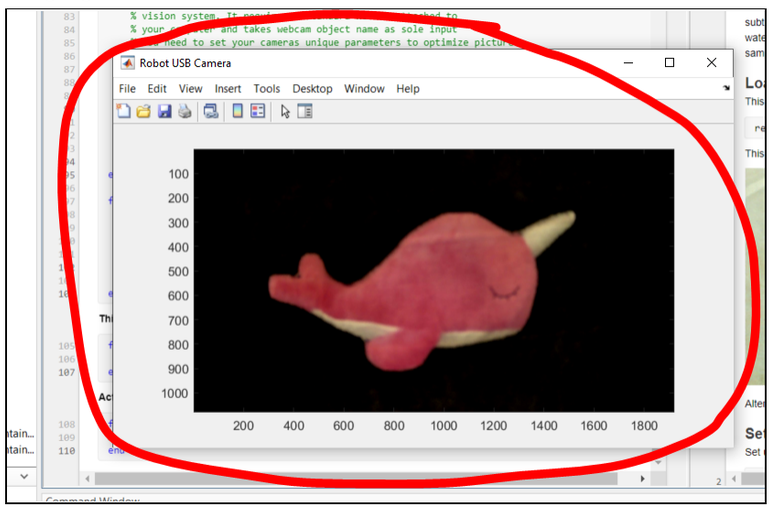
```
    robotCam.Brightness = 100;
end

function [robotImage] = SENSE(robotCam)
    % This function aquires a single image from the USBCamera testCam
    % and displays it in a stand alone figure
    % D. Barrett 2021 Rev A

   robotImage = snapshot(robotCam);

end
```

Think Functions (store all Think related local functions here)

It just takes a snapshot of what is in front of the USB camera and returns that as a **robotImage**. The THINK and ACT functions are empty placeholders waiting for future work. It's a good practice for both modularity and clear code structure to populate your code with placeholder functions to be filled in later.

```
104   end

      Think Functions (store all Think related local functions here)

105   function THINK()
106      % null function, not much thinking to do here.
107   end

      Act Functions (store all Act related local functions here)

108   function ACT()
109   % null function, not much acting to do here.
110   end
```

Running the code will take a snapshot and display it both within the livescript and in a free standing figure called **Robot USB Camera**



Finally to wrap up your first image procession script add a placeholder section in which to do further processing and a clean shutdown section, that releases the webcam as shown below:

Mission data processing

For many robot applications, you will need to post-process the data collected after the mission. Here we will plot the measured versus actual range positions.

```
70  % Add post processing code here, mmight be good to downoad images to a
71  % MATLAB drive location where you can work on processing them latter
72
```

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.

```
73  % Stop program and clean up the connection to WebCam
74  % when no longer needed
75
76    clc
77    clear robotCam          % connection is no longer needed, clear the cam variable.
78    disp('SimpleUSBCameraTutorial Done');

SimpleUSBCameraTutorial Done

79    beep        % play system sound to let user know program is ended
```

It's very important to release your webcam, so that other applications, like Zoom, can use it.

Run the program a number of times. Modify it a bit to let you save images as files (image.jpg) in your team matlab drive directory. Try a few different targets and against both the black and white backgrounds. When you are working in computer vision, you want to have a lot of trial data image files to try out new algorithms on, without needing to go to the lab and fire up the whole test bench.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Image processing in MATLAB is a pretty big ocean with lots and lots of powerful functions. In order to get a better fundamental grasp of it, you will next need to work your way through an on-line tutorial covering the basic functions and principles. After a lot of searching, I found a really well written and executed one that covers most of what you will need to know to effectively write code for this robot lab. Please run through this extremely well written on-line (40min) tutorial:

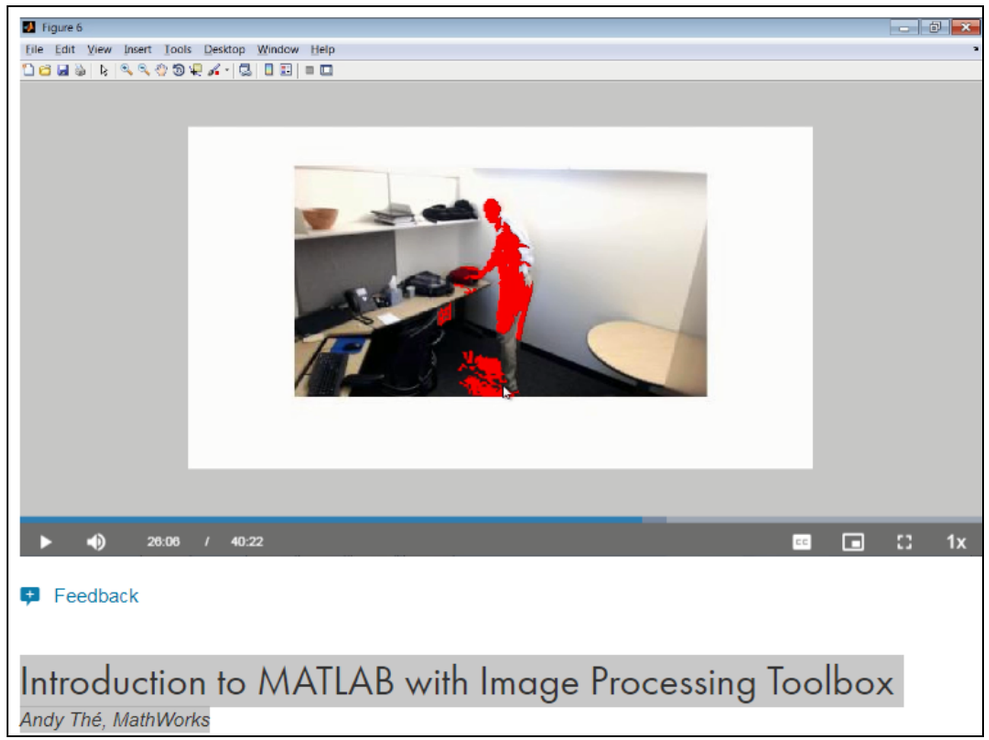
Introduction to MATLAB with Image Processing Toolbox *Andy Thé, MathWorks*
https://www.mathworks.com/videos/introduction-to-matlab-with-image-processing-toolbox-90409.html

It will give you the basic image processing skills and some pretty cool advanced ones needed to make your final USB camera computer vision demo code. In short summary, it will teach you how to pull in an image into MATLAB, segment it, find statistics on it, use thresholding and differencing techniques to pull an object of interest out of it and finally how to roll all of that up into reusable functions and a nice professional application.

If you crank the playback speed up to 1.5X or 2X (right bottom corner controls) in the boring parts or in parts you already know, then slow down for interesting bits, you could probably scan through it in 20 min, but it would be a better learning experience for you to watch a little, write that code live on your laptop, watch a little more, write a little more code, and so on, until you get all the way through to the final **APP developer part.** This video is from a few MATLAB versions back, so you will need to extrapolate the old APP developer work to the new APP developer built into 2020b.

This one example lets you build a pretty cool computer vision intruder detector that uses background subtraction to pick out a new object in front of a fixed background.

In the place of his office intruder images, you can take and use two images of the lab test track setup, one with an empty black background everywhere and one with a colored target in front of that background.



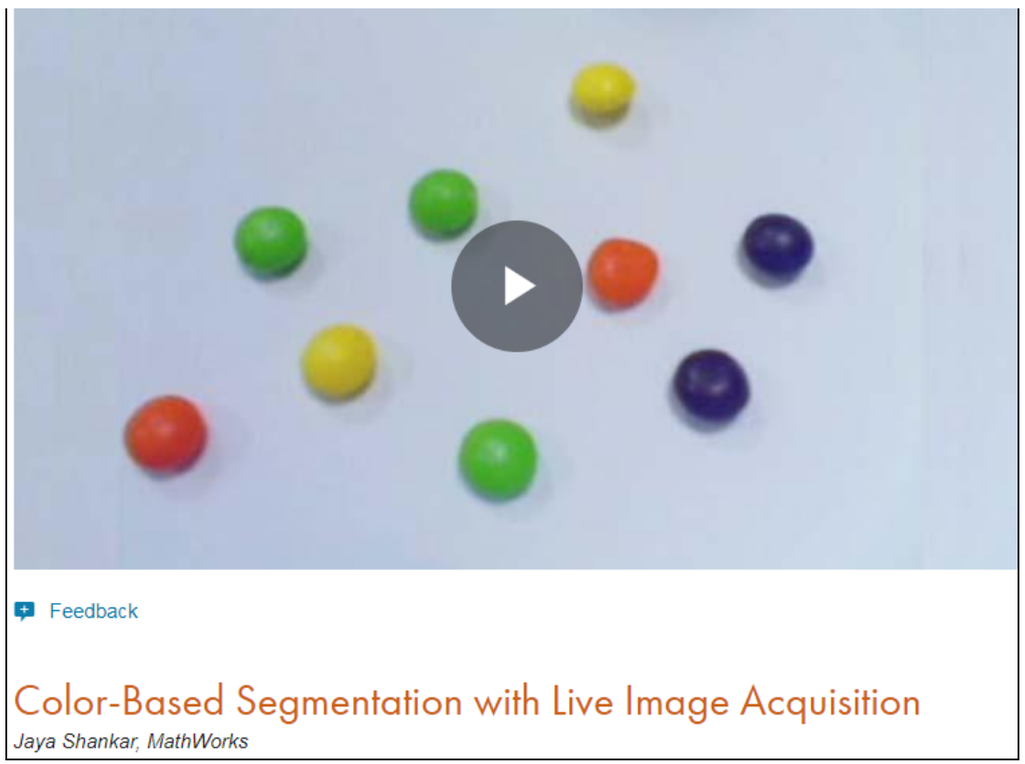
This tutorial is good at finding what has changed in a video workspace and that is pretty much exactly what you will need to do for the first part of the USB lab. Please:

1. Remove all targets from the camera field of view and take a single image of test space.
2. Place a single target in the space and take a second picture of it (without moving USB camera)/
3. Use the code you generated from the above Tutorial video to write a short **Find-The-Target** MATLAB livescript script.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

After mastering the computer vision technique of differencing to find a target, it would be good to be able to find and identify multiple different colored targets. Please view this short video tutorial on how to do so:

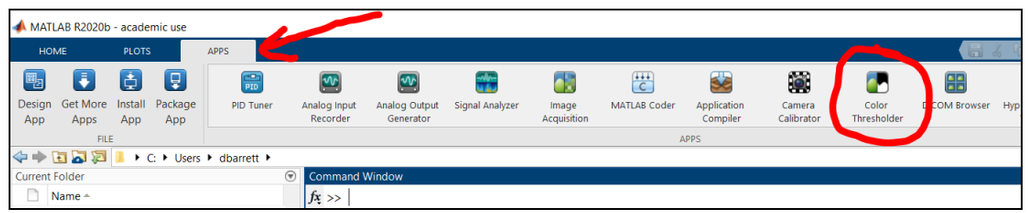
https://www.mathworks.com/videos/color-based-segmentation-with-live-image-acquisition-68771.html



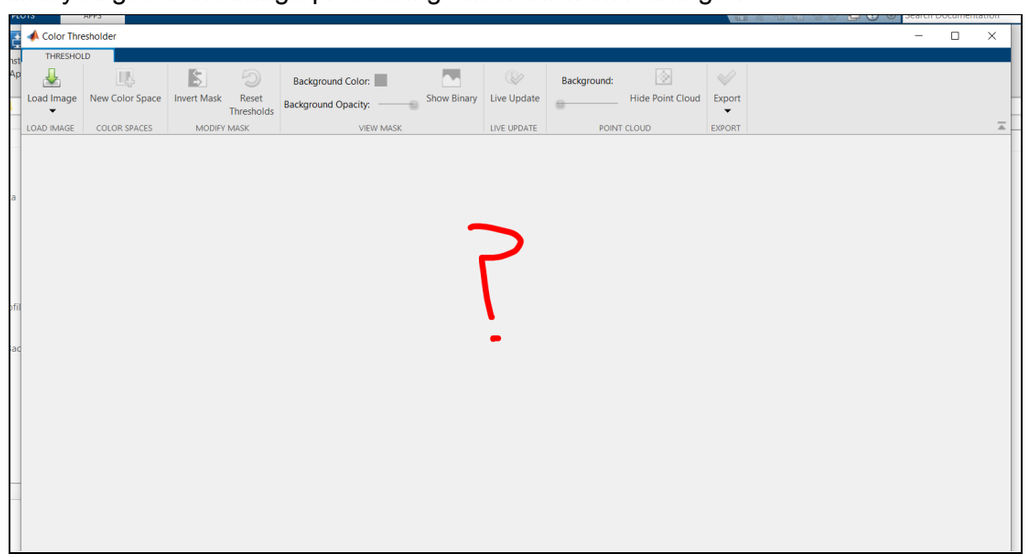
And then, drawing heavily from the tools shown in the above tutorial and with the guide below, let's write a MATLAB script that lets you find and identify the multiple colored targets on the test station cart.

Using your existing simple USB camera Tutorial as a code starter, we are going to use one of MATLABS built in computer vision Apps to create a color filter that will segment out and find just objects of the particular color that you choose. It will generate a really cool function that will take your input image and reliably find just those parts of it that have the color you specified.

You might want to use this tool multiple times to build several color filters, one for each potential target. Go to **APPS tab** then **ColorThresholder**

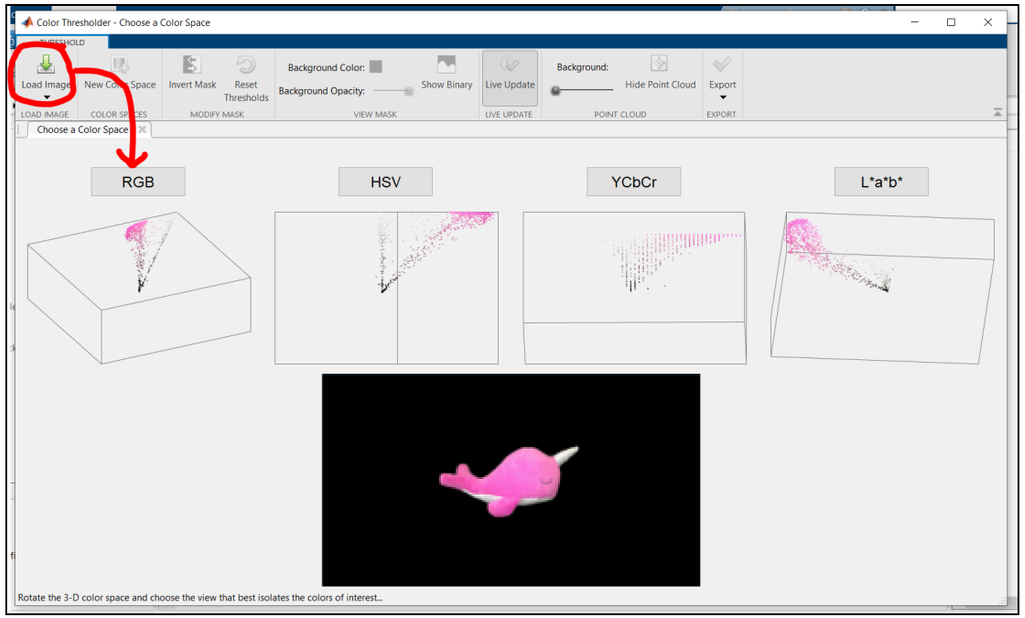


There are a lot of APPS in 2020b, you may need to scan down the full page of APPS until you get to the image processing ones. Click on it and get:

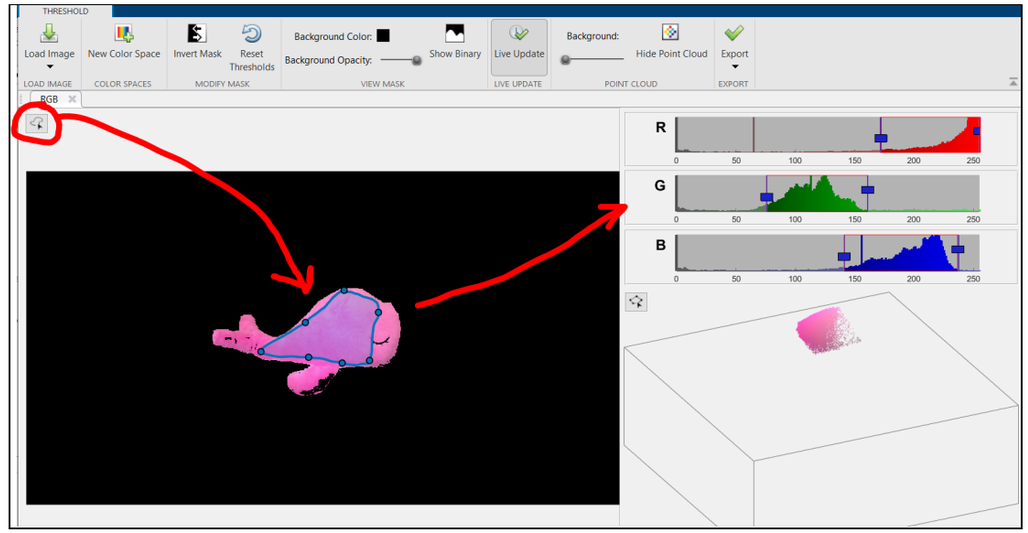


ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

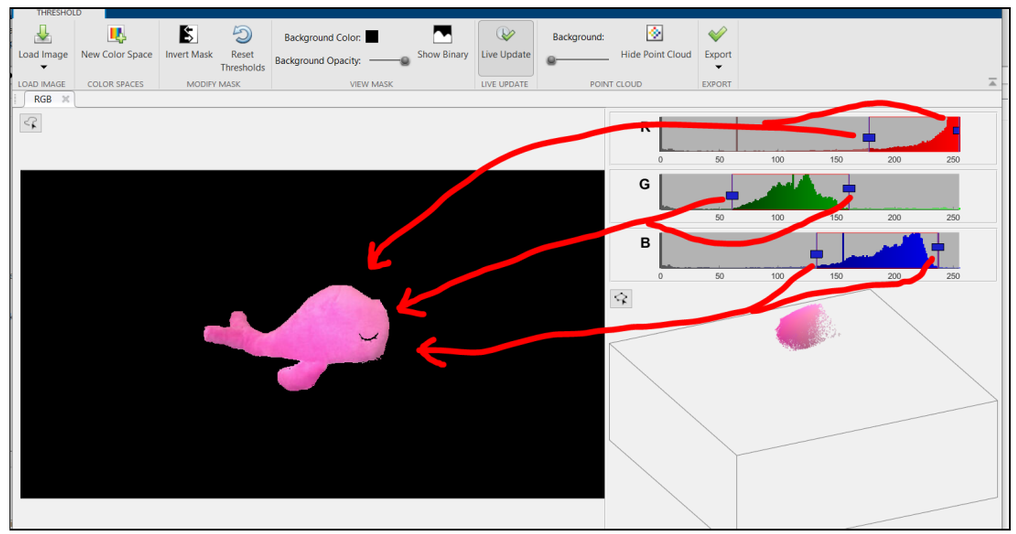
Click on **Load Image** and load one of your saved images from the previous section of this tutorial, it will appear in the center work space:



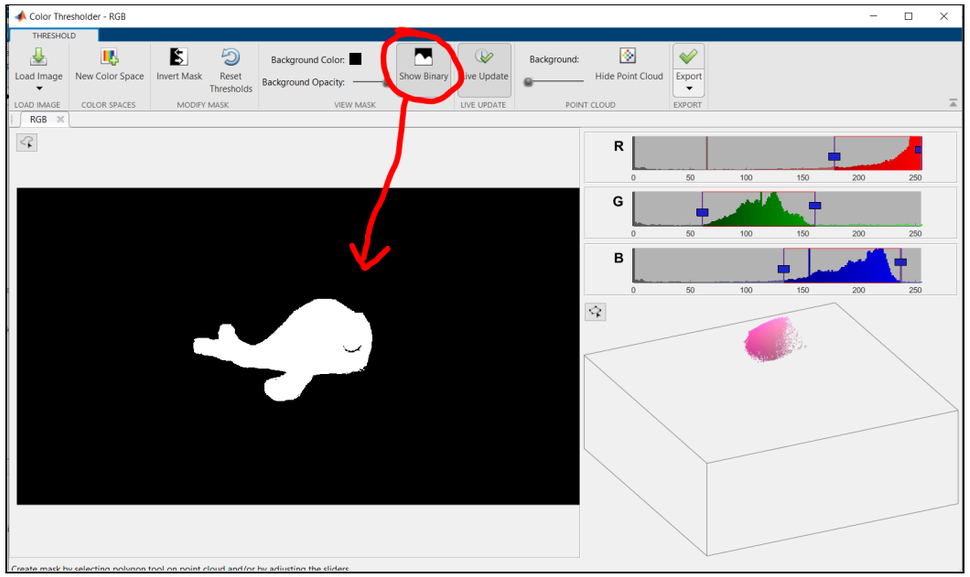
Select RGB color space (we will start out with RGB, but downstream you might find other color spaces let you better distinguish between similar colors) and then use the **lasso tool** in the top left of the image workspace to lasso the colored area of your target.:



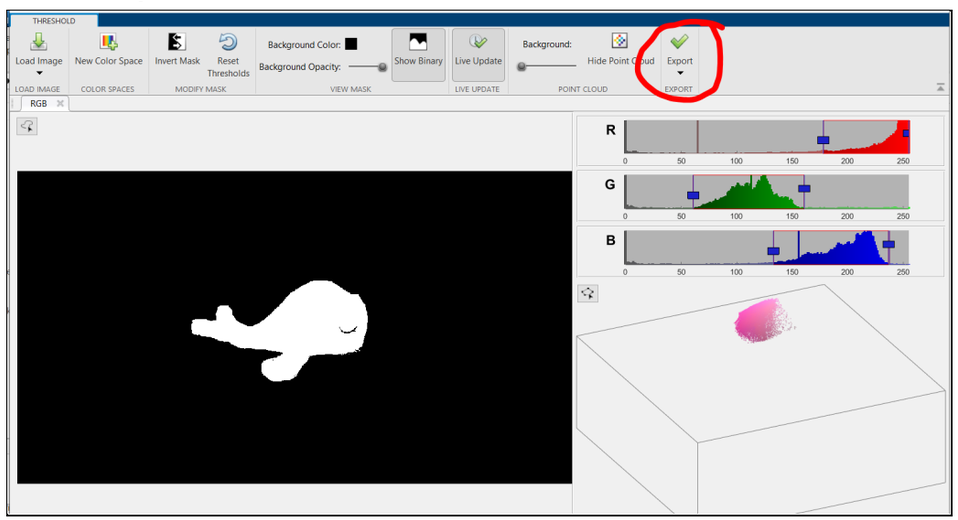
Use the two bar sliders on the R G B tracks to fine tune the filter to capture the best segmentation of the image that you can.



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



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



and the App will drop a new masking function, fully documented into your Editor window.

Editor - Untitled*

Untitled*

```
1  function [BW,maskedRGBImage] = createMask(RGB)
2  %createMask  Threshold RGB image using auto-generated code from colorThresholder app.
3  %  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
4  %  auto-generated code from the colorThresholder app. The colorspace and
5  %  range for each channel of the colorspace were set within the app. The
6  %  segmentation mask is returned in BW, and a composite of the mask and
7  %  original RGB images is returned in maskedRGBImage.
8
9  % Auto-generated by colorThresholder app on 15-Feb-2021
10 %------------------------------------------------------
11
12
13 % Convert RGB image to chosen color space
14 I = RGB;
15
16 % Define thresholds for channel 1 based on histogram settings
17 channel1Min = 178.000;
18 channel1Max = 255.000;
19
20 % Define thresholds for channel 2 based on histogram settings
21 channel2Min = 61.000;
22 channel2Max = 161.000;
23
24 % Define thresholds for channel 3 based on histogram settings
25 channel3Min = 133.000;
```

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

Editor - C:\Users\dbarrett\Dropbox\C_Olin Courses 2017\ENGR3390 Robotics\20XX steady state course material\course code\SenselabCode2021\purpleMask.m

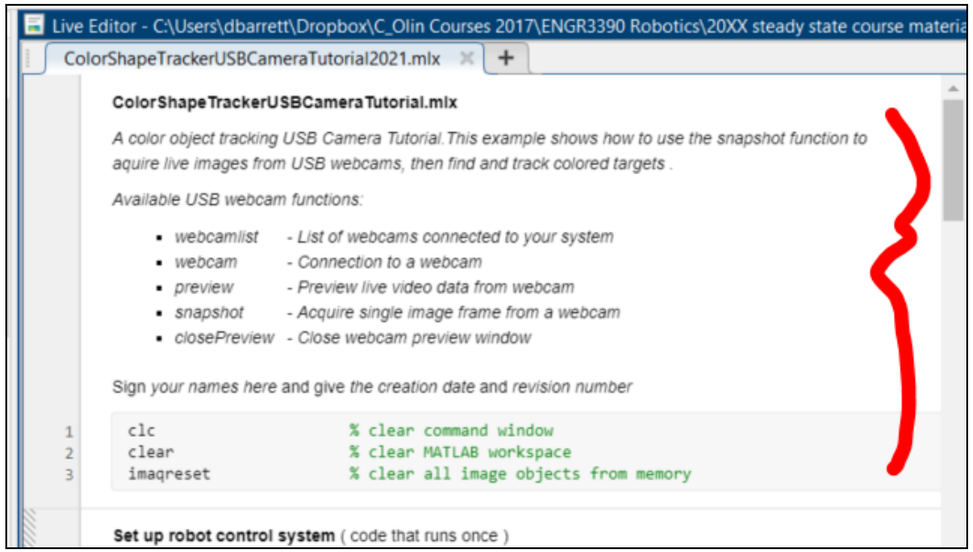
purpleMask.m

```
1  function [BW,maskedRGBImage] = purpleMask(RGB)
2  % purpleMask  Threshold RGB image using auto-generated code from
3  % colorThresholder app.
4  %  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
5  %  auto-generated code from the colorThresholder app. The colorspace and
6  %  range for each channel of the colorspace were set within the app. The
7  %  segmentation mask is returned in BW, and a composite of the mask and
8  %  original RGB images is returned in maskedRGBImage.
9
10 % Auto-generated by colorThresholder app on 15-Feb-2021
11 %------------------------------------------------------
12
13
14 % Convert RGB image to chosen color space
```

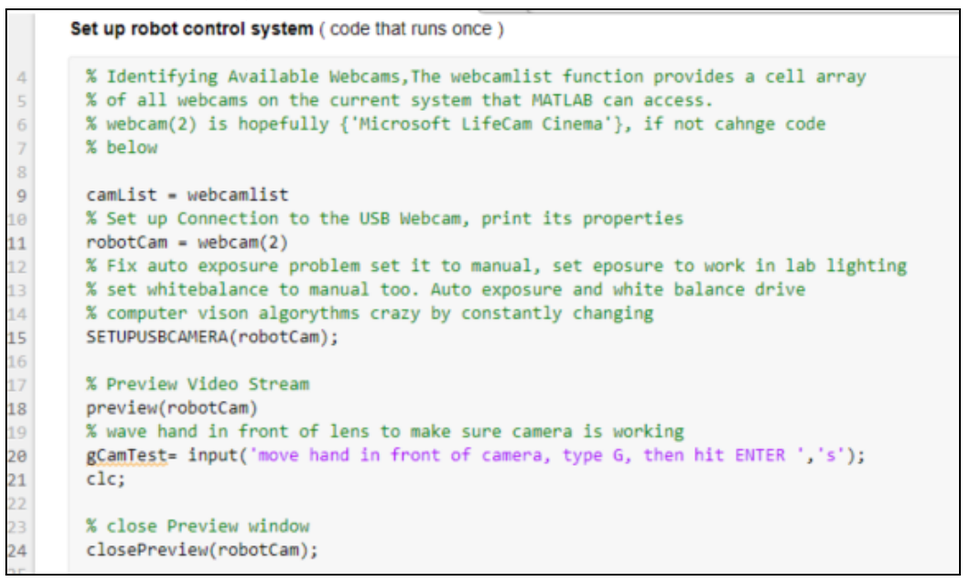
Go back and repeat with a few of the other colored targets on the test stand.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

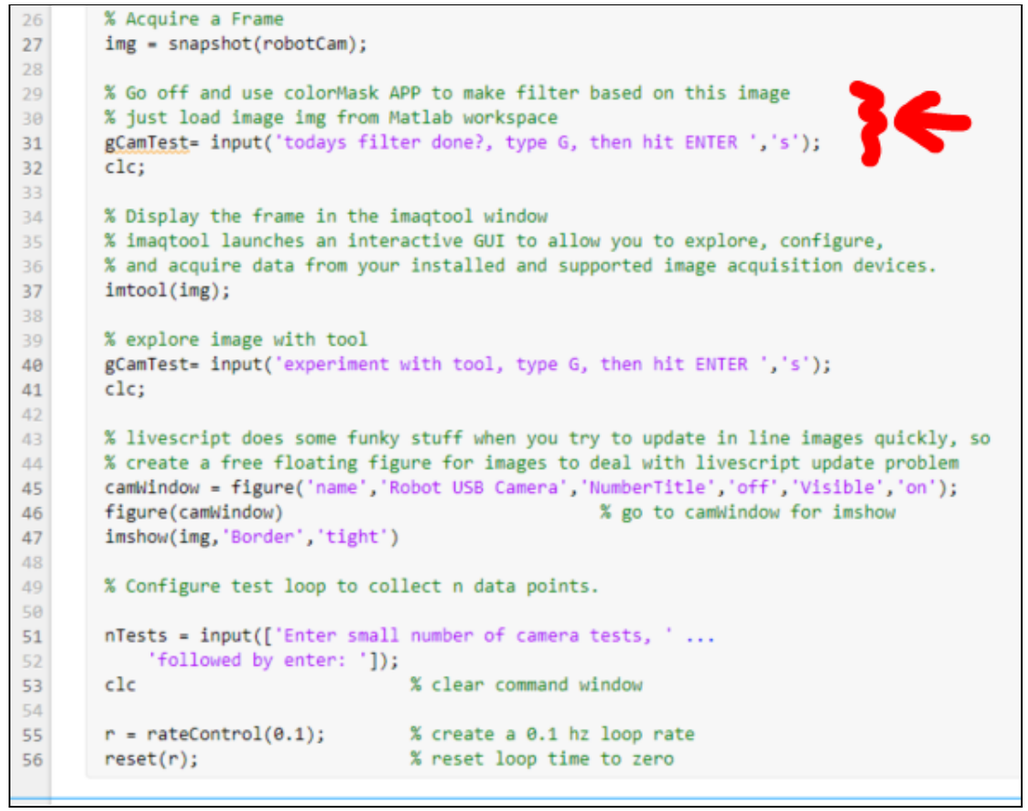
Next, take your **SimpleUSBCameraTutorial** code and modify it as shown:



Most of the code stays the same:



We will add a line to pause code after you collect an image of your target and let you jump out to run the color thresholder APP to build a new filter for it:



In practice, you will want to build up a set of colorMask filters, probably one for each color target and then you would need to add some additional code to support looking for multiple differently colored targets, For clarity here, we will walk you through how to build just one of them.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

The main control loop stays the same with the one modification being a full build out of the SENSE function to return the centroid and area of a colored target. You could use the centroid coordinates to follow the target (or avoid it) and you could use its area, along with a predetermined calibration table, to estimate your distance from it. Please modify your code as shown:

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

```
controlFlag = 1;
% create a loop control
while (controlFlag < nTests+1)   % loop till ntests data captured

    % SENSE finds centroid and area of target
    [centroids, targetArea] = SENSE(robotCam, camWindow)
    THINK();                     % compute what robot should do next
    ACT();                       % command robot actuators

    % pause to allow you to move objects
    camtest= input('move object to new position, type G, then hit ENTER ','s');
    clc;

    waitfor(r);                  % wait for loop cycle to complete
    controlFlag = controlFlag+1; % increment loop
end
```

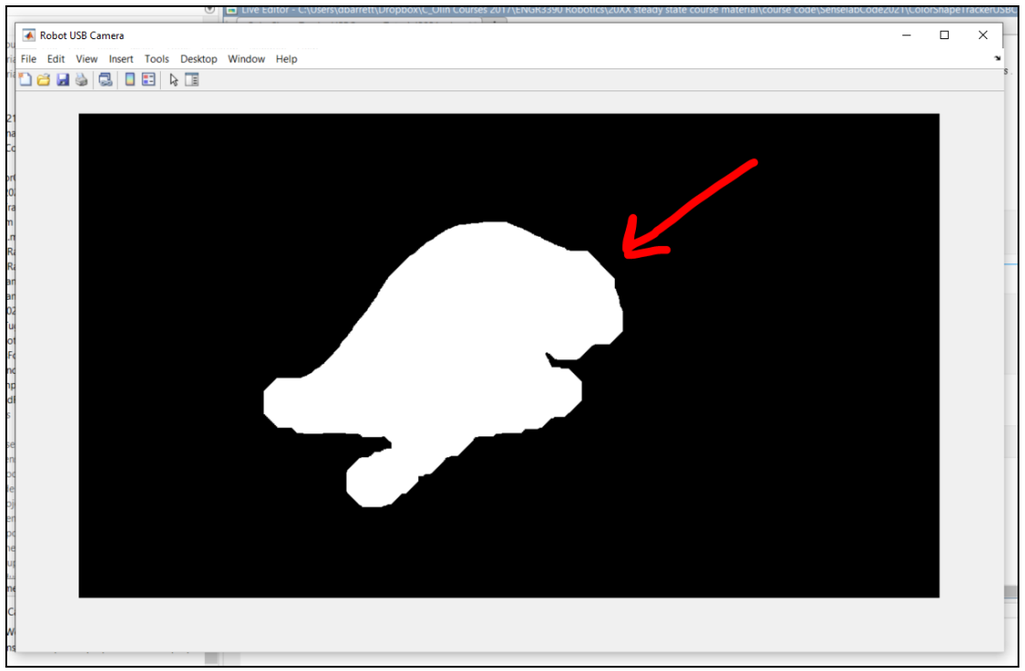
Diving right into the new SENSE function, please modify as we go:

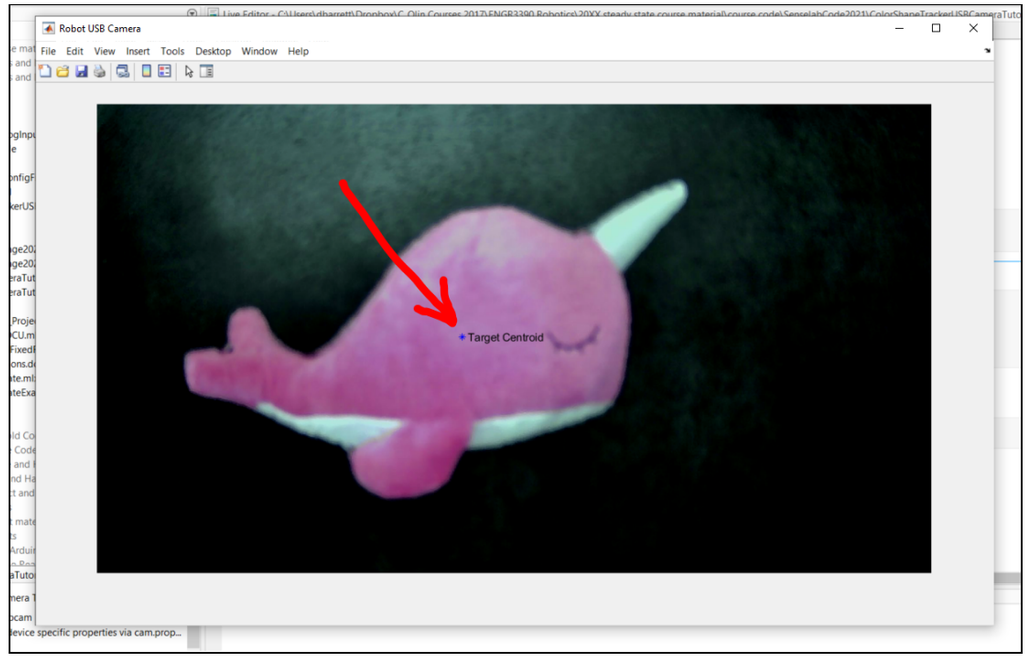
```
function [centroids, targetArea] = SENSE(robotCam, camWindow)
   % This function aquires a single image from the USBCamera testCam
   % finds and returns centroid of purple area in image
   % D. Barrett 2021 Rev A

   % capture image
   robotImage = snapshot(robotCam);

   %% Use Color Threshold App to create a colorMask function (purpleMask)
   %  and place function in same directory as this script. Your functions
   %  will have different names depending on color of target you want to
   %  track, apply colorMask Function to captured image
   %  this will create two new images
   % [BW,maskedRGBImage] = purpleMask(RGB)
   %  BW is binary mask of image
   %  colorMaskeImg will be the color image inside the mask
   [targetMask,purpleImg]=purpleMask2(robotImage);
   figure(camWindow)                          % go to camWindow for imshow
   imshow(targetMask)
   camtest= input('check out masked image, type G, then hit ENTER ','s');
   clc;
```

After applying the colorMask function to the current image, you get back two images. The targetMask one is a black and white mask of the filtered thresholded target that we will use for all the subsequent image processing.

You will take that BW image and remove much of the noise from it:

```
clc;

%% Preprocess image to remove noise
se = strel('disk',50);               % structured element erosion function
cleanImage= imopen(targetMask, se);  % Morphologically open image
figure(camWindow)                                % go to camWindow for imshow
imshow(cleanImage)
camtest= input('check out cleaned image, type G, then hit ENTER ','s');
clc;
```





Then used that cleaned up image **cleanImage** to find the centroid of that colored target:

```
clc;

%% Calculate the centoid of the white part of masked area (target)
targetCenter = regionprops(cleanImage,'centroid');
% Store the  x and y coorinates in a two column matrix
centroids = cat(1,targetCenter.Centroid);
% Display the original image with the centroid locations superimposed.
figure(camWindow)                              % go to camWindow for imshow
imshow(robotImage)
hold on
plot(centroids(:,1),centroids(:,2),'b*')
text(centroids(:,1),centroids(:,2), '  Target Centroid');
hold off
camtest= input('check out target center, type G, then hit ENTER ','s');
clc;
```

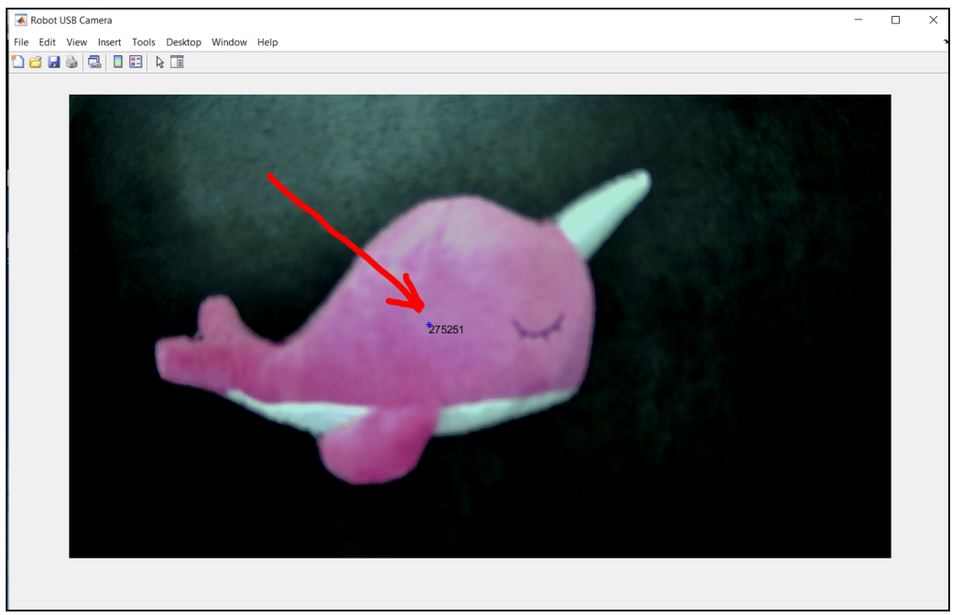
Having found the centroid, the code draws and labels its calculated location onto the original image. In image processing it is always good to do this, namely to draw your final results onto the original image as both a sanity check and as a way of understanding how the algorithms work under a variety of lighting conditions. As an experiment try what happens with lights turned low or turned off. How well does yoru target finder work?

The SENSE function returns the coordinates of the target back up to the main program. In a full robot control system (like the THINK lab), you could use the x-component of this centroid to steer the robot Tug toward the NarWhal.

In addition to the centroid, please add the following code to calculate the perceived area of the target:

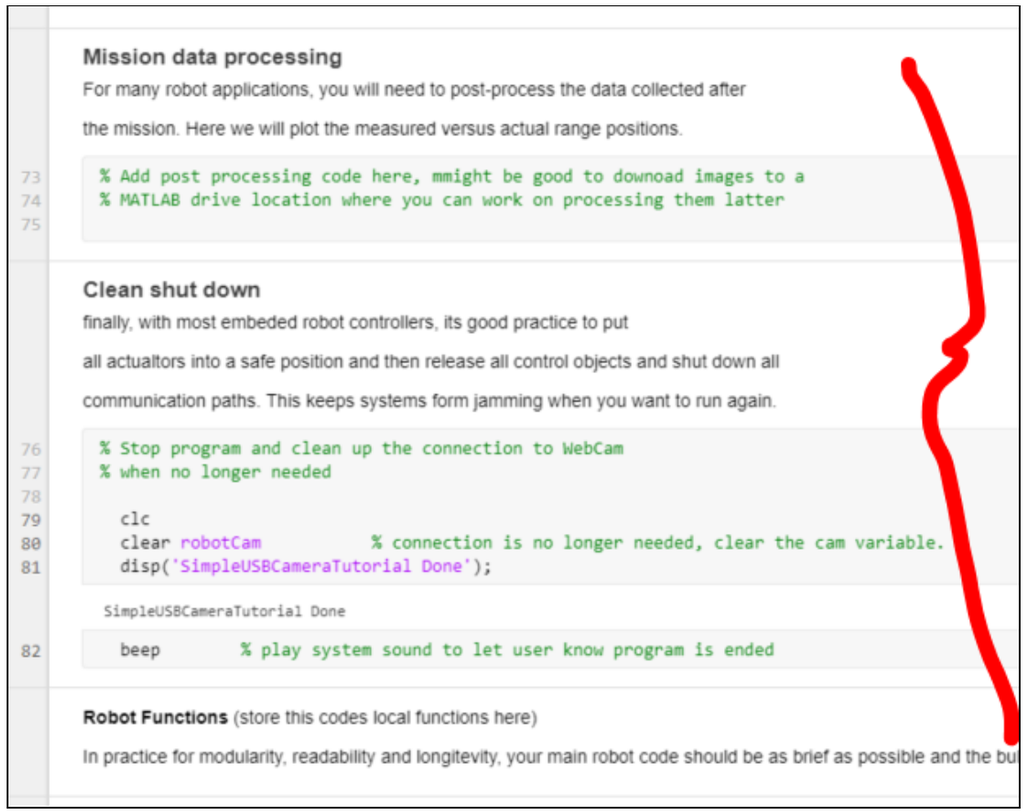
```
142     clc;
143
144     %%  Calculate the area of the target
145     targetArea = regionprops(cleanImage,'area');
146     area=targetArea.Area;
147     % Store the  x and y coorinates in a two column matrix
148     centroids = cat(1,targetCenter.Centroid);
149     % Display the binary image with the centroid locations superimposed.
150     figure(camWindow)                          % go to camWindow for imshow
151     imshow(robotImage)
152     hold on
153     plot(centroids(:,1),centroids(:,2),'b*')
154     text(centroids(:,1),(centroids(:,2)+ 10),num2str(area));
155     hold off
156     camtest= input('check out target area, type G, then hit ENTER ','s');
157     clc;
158
159 end
```

Generating this:



As stated before, with a bit of experimental distance versus area measurements, you could get a rough sense of distance from the object by its perceived area. As you get closed it gets bigger. You could use this information to avoid running into it (driving into NarWhals is just bad and probably illegal) !

The remaining code stays unchanged and is shown here for completeness:



Mission data processing

For many robot applications, you will need to post-process the data collected after

the mission. Here we will plot the measured versus actual range positions.

```
% Add post processing code here, mmight be good to downoad images to a
% MATLAB drive location where you can work on processing them latter
```

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 WebCam
% when no longer needed

  clc
  clear robotCam          % connection is no longer needed, clear the cam variable.
  disp('SimpleUSBCameraTutorial Done');
```

```
SimpleUSBCameraTutorial Done
```

```
  beep        % play system sound to let user know program is ended
```

Robot Functions (store this codes local functions here)

In practice for modularity, readability and longitevity, your main robot code should be as brief as possible and the bu

Modify your code, run section by section and make sure you understand what each part does. This code will get you to the point of finding one target. Your next steps

will be to extend the color filters to find other targets. As a challenge, consider how you find a white target against a white background ?

Please see an instructor or Ninja for help as needed.

Final Demo: Using your newly acquired computer vision skills, please prepare two final demos:

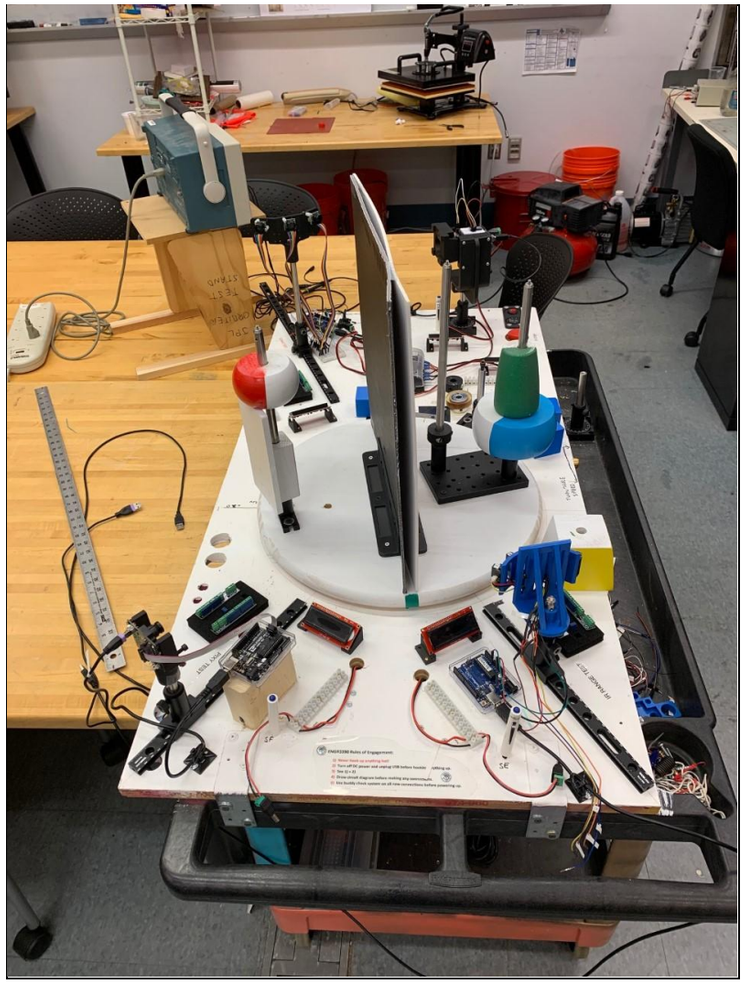
Find Target:

4. Ninjas will place 1 fixed colored target on the test track rail. Please write a demo and application that correctly finds direction and distance to the colored target.
5. Ninjas will move 3 targets to a second set of locations. Please write and demo code that can correctly identify bearing and range to target as well as identify which target it is.
6. Stretch goal. Write demo code to track and give range and bearing to a moving target. Ninja will slowly rotate one target in the field of view of camera in front of white background

Find Hole:

4. Given a single fixed colored target, find the largest hole (unobstructed area) in the field of view and give a clean bearing to its center (Open water at 56 degrees).
5. Given a field of many fixed colored targets, find the largest hole to drive through and give range and bearing to its center (Open water at 76 degrees).
6. Stretch Goal. Given a field of slowly moving targets (via Ninja), find the largest hole to drive through and give range and bearing to its center (Open water at 45 degrees).

If you have time you can take on a stretch goal and try to build your code into an App with the App building skills you acquired in the MATLAB tutorial. As always, please see an instructor or Ninjas for help with any part of the above.



Hokuyo LIDAR Range Sensors

The widely available Hokuyo URG-04LX LIDAR sensors are the robotics world low-cost planar range sensors of choice. Rather than giving the robot a single range measurement over a short range, the Hokuyo LIDAR will allow 240 polar range measurements, within a single plane, over a 10m range.

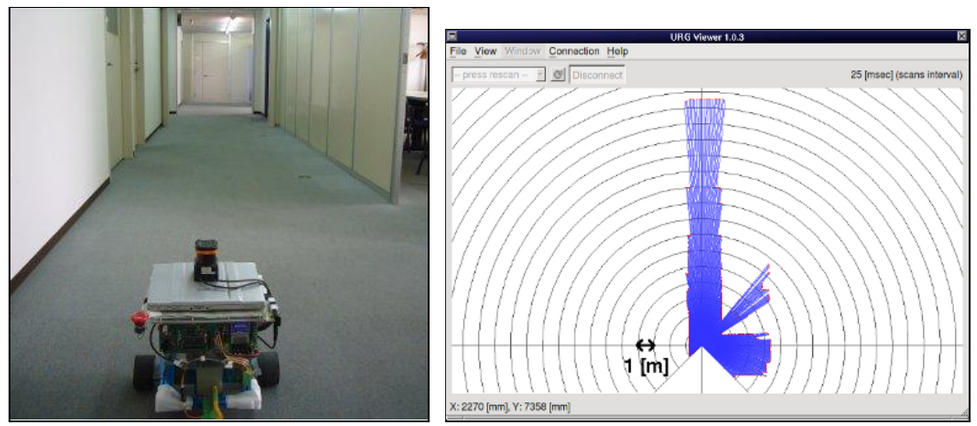
:Hokuyo URG-04LX on PanTilt test fixture

URG-04LX is a laser sensor for area scanning. The light source of the sensor is an infrared laser of wavelength 785nm with laser class 1 safety. Scan area is 240º semicircle with a maximum radius of 4000mm. The scan pitch angle is 0.36º and the sensor outputs the distance measured at every point (683 steps).

Laser beam diameter is less than 20mm at 2000mm with maximum divergence 40mm at 4000mm.

Principle of distance measurement is based on calculation of the phase difference between projected and reflected Laser light, due to which it is possible to obtain stable measurement with minimum influence from the object's color and reflectance.

The URG sensor is a distance laser sensor made by Hokuyo Automatic. This sensor can acquire highly accurate range data over a wide scan range.



Robot using Lidar to Map Hallway

The first figure shows an URG sensor mounted on the robot top and the second its surrounding environment. Obtained raw data from the sensor are pairs of ranges and directions which can be plotted graphically as is shown in the second figure. In the this figure, the sensor is placed at the origin of the robot coordinate system, and obtained ranges and directions are drawn by blue lines. Objects that appear to be a

continuous line such as walls are shown by red points plotted, and this is done not by the sensor itself, but the viewer written application.

This sensor has a communication channel such as a USB port or serial RS232 port, and can be controlled by command interface. The communication protocol is common in all sensors.

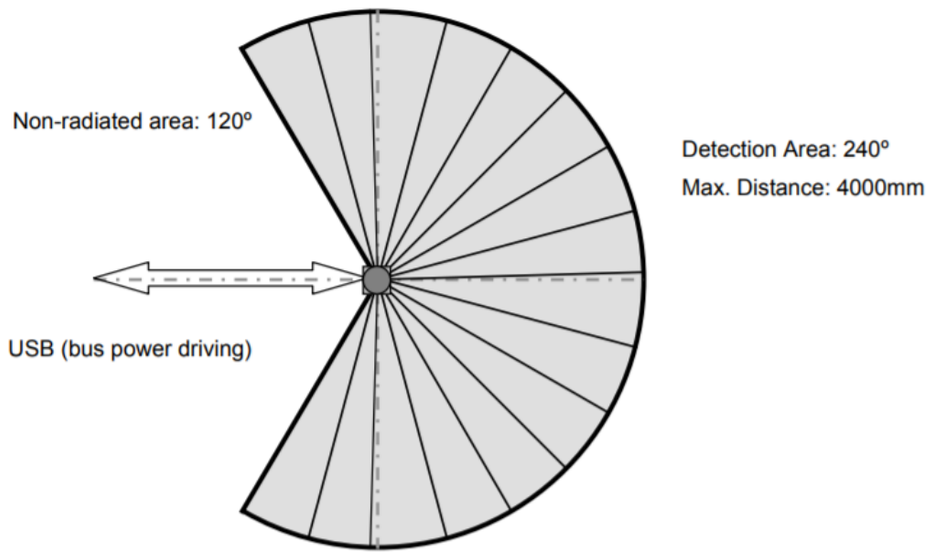


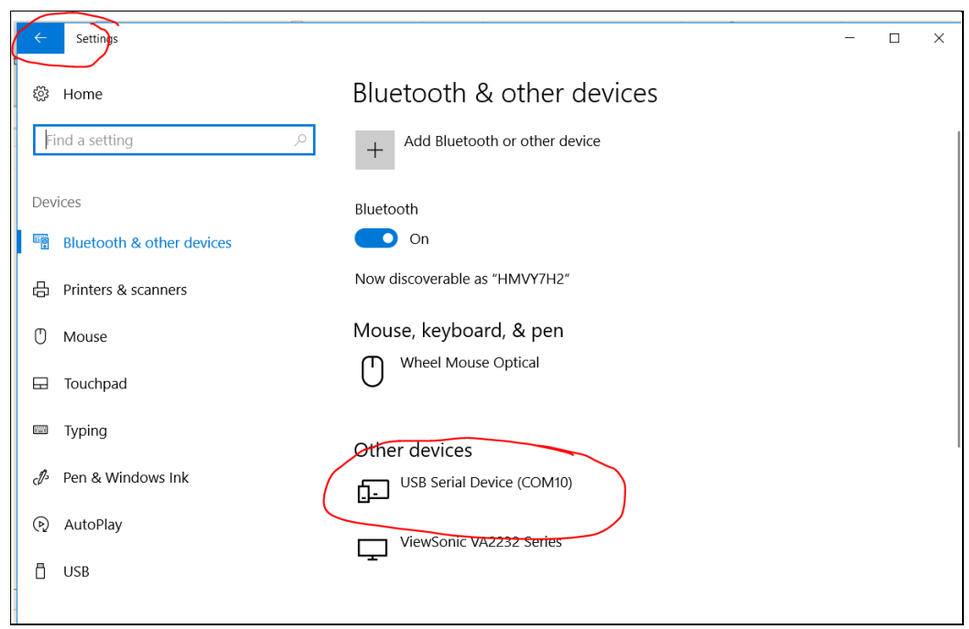
Figure 5: Hokuyo Scanning Zone

URG-04LX-UG01 is a 2-dimensional laser sensor for measuring the distance to the objects.
●Light weight (160g). Best for robot!
●Low-power consumption (2.5W) for longer working hours.
●Wide-range (5600mm×240°).
●Accuracy (±30mm).*
● Distance and angle data output with high angular resolution (0.352°).

For more detail on LIDAR see Hokuyo website:
https://www.hokuyo-aut.jp/search/single.php?serial=165

Hokuyo Lab Set Up Work

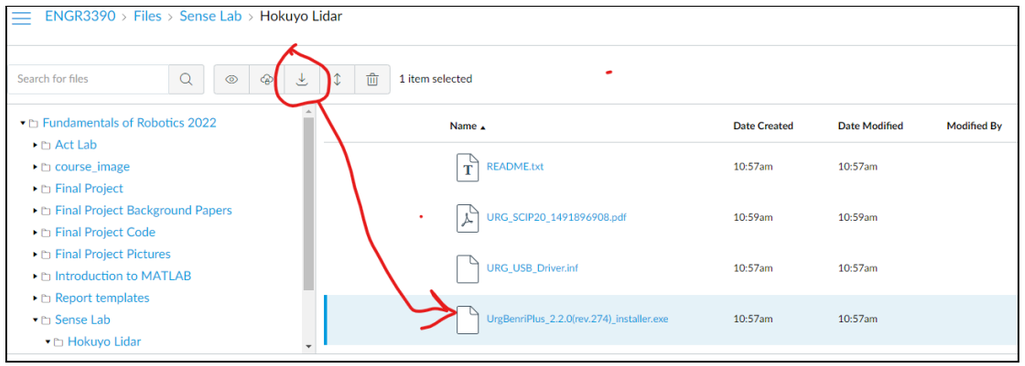
First, Plug in the Hokuyo LIDAR on the test bench. Then go to your **Windows 10 Settings>Devices(Bluetooth, printers,mouse)** and look at the **Other devices** category.. It should show up as something like a USB **Serial Device (COM10)**. Please write down your COM port number. You will need to edit the code we will give you for basic LIDAR operations to match it.



ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

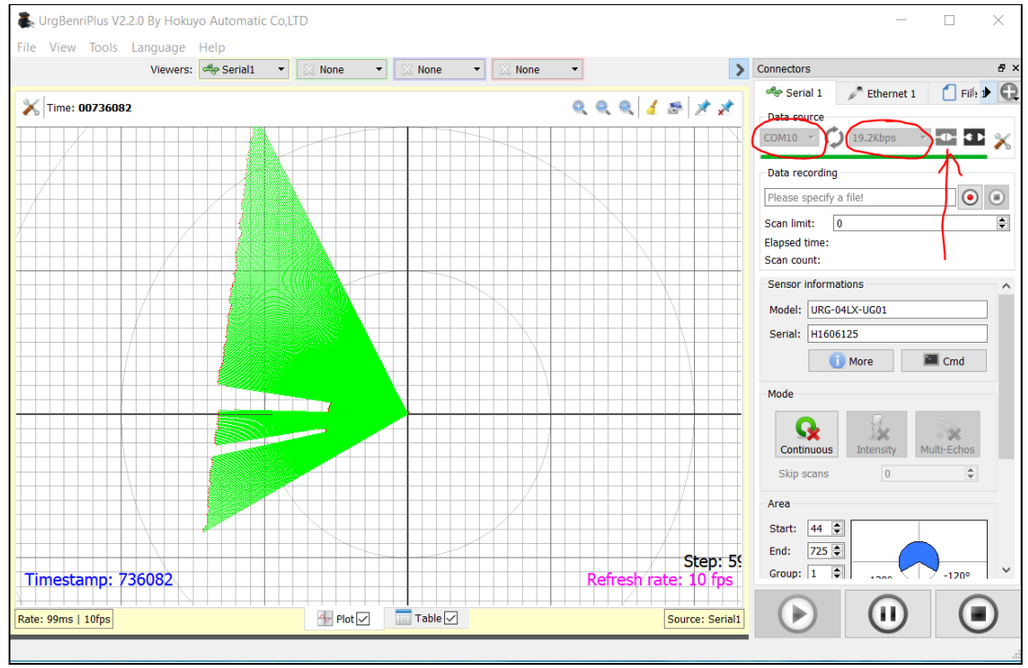
Next go into the Canvas folder and download a copy of the zipped Hokuyo Lidar viewer program **UrgBenriPlusV2.2.0_Installer.exe**.

You will need to download the files and then click on the installer to load the application on your Laptop:



Please see a course Ninja or Instructor for help in downloading and installing application if you need it. This process will probably trigger your student laptop virus protection programs and Windows will ask you if you really really want to install the Hokuyo app. This file comes directly from the Hokuyo website. We had to register with them as an end user to get access to it. Olin students went through and thoroughly scanned all of these files with a virus checker (passed cleanly) and then went online and searched for any instances of tracking and malware associated with it and could find none. So we are confident enough with it and believe it is safe for your laptop..

Plug in the Lidar and run the application, set the Com port to match your comport number, set the communication rate to 19.2kbps and then click on the little connect icon next to the comport. You should get something like this:



And get your first live lidar scans! You should be able to clearly see the flat wall behind the sensor targets as well as see multiple targets mounted on rail in front of it.

Ful MATLAB support for this Hokuyo lidar is a bit underdeveloped at the moment and seems to all derive from a single piece of graduate student hacking back in 2007 (?). At a practical level this means that the Lidar example code we are going to give you is a little bit fragile and you can jam the Lidar in an unresponsive state.

When writing new code, if Lidar stops working, it is really hard to tell if it's your new code, or if you just jammed the Lidar's com port or its buffer and the Lidar is hung up. This can get quite frustrating very quickly. The **UrgBenniPlus** tool will be very useful to double check that LIDAR is not hung, and you should have it ready and waiting for if-when you do manage to freeze the Lidar. Matlab and Olin are interested in writing a new more robust Hokuyo communication driver. If this is something you would be interested in doing as either a stretch lab goal or in place of this lab, please contact Dave.

Developing a device driver for a complex sensor, like the Hokuyo Lidar, is a complex software heavy task requiring much highly skilled understanding of the devices serial communication protocols. If you are really interested in this part of robotics, in addition to the Hokuyo communication specification there is a great resource for Hokuyo Lidars here:

https://sourceforge.net/p/urgnetwork/wiki/top_en/

The code we will use in this lab has original parentage here:
https://www.mathworks.com/matlabcentral/fileexchange/57425-matlab-driver-for-hokuyu-urg-family?s_tid=FX_rc1_behav

https://www.mathworks.com/matlabcentral/fileexchange/36700-hokuyo-urg-04lx-lidar-driver-for-matlab

When done playing with the app and moving various targets in front of it to see what they look like to the Lidar, please click on the **disconnect** button (next to connect button and free up the USB driver to talk with the MATLAB code you will generate next.

First LIDAR program

Most robotics Engineers usually work a level or two above the Lidar device drivers, up at the algorithm and system integration level. To save you the 4 weeks (and some pain) of developing your own Matlab-Hokuyo communication code, we will give you a starting program that has most of the parts you will need for this lab built in. Code is a little wonky and somewhat fragile, but it works.

The first section of this code contains a short bit of documentation that asks you to run the **UrgBenriPlus** Hokuyo viewer first, just to make sure that the Lidar isn't wedged from the last user, your comport number is correct and that your device drivers are all intact.

Because communicating with the Hokuyo Lidar is a bit complex, it's complexity is modularly packaged away inside a **software object.** In simple terms, as you get more and more skilled in programming, you will want to pursue writing application independant reusable modular code where functionality is cleanly separated from the application it resides in. One way to do this is to bind up all the variables, data-structures and functions that perform some useful operation into a software object. This leads to a part of the programming universe called **object oriented programming** (OOPs). Most higher languages support objects and object oriented programming, and Matlab is no exception. We won't touch on it much in Fun-Robo, but occasionally someone else might have written a useful piece of code that used objects and we will reuse that to save your time recreating it in a non-object format.We will do that here with this first Lidar code, which will create and use a Lidar object to facilitate communications with your lidar.

During programming, it's often the case where you jump around a lot in the code and incidentally leave variables and objects in the Matlab workspace that can cause conflicts later on. The first code you write will clean up that workspace, to save you from those potential conflicts. Please open a new livescript and write:

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

SimpleHokuyoLidar2022.mlx

Connect and sample data from a Hokuyo URG-40LX Laser scanner

The widely available Hokuyo URG-04LX LIDAR sensors are the robotics world low-cost planar range sensors of choice. Rather than giving the robot a single range measurement over a short range, the Hokuyo LIDAR will allow 240 polar range measurements, within a single plane, over a 10m range.

Before running this code please check what USB comport your Lidar is plugged into and then run the **UrgBenriPlusV2.2.0** Hokuyo viewer to make sure the Lidars fully operational and wasn't left in a jammed state.

You can run code sequentailly, section by section, but beware backing up or running sections out of sequence! Serial ports are tricky beasts and don't like it if you don't open and close them carefully.

Origial code Copyright The MathWorks, Inc 2019. This code has been heavily mofified 2-4-2022 to support Olin Fun-Robo Lidar Lab by D.Barrett

```
disp("program running");    % Indicate lidar code starts ok
```

```
program running
```

```
% Due to the complexity of talking with the Hokuyo over a Windows 10 serial link,
% parts of this code are bit wonky and sometimes throws some non-fatal
% errors when running.
% We will turn off error warnings for now till course staff or students can
% find and fix this.
warning('off','all');       % Temporarily turn Matlab warnings off

% Lidar is contained in a Matlab serial object, we need to clean up any old lidar
% objects in the work space to prevent conflicts when creatign a new one.
% Search for and delete old ones. For Fun-Robo, don't worry about what an
% object is for now, just treat it like a big fancy function.
serialObjs = instrfind;     % Read instrument objects from memory to MATLAB workspace
if ~isempty(serialObjs)     % If there are old objects close them
    fclose(serialObjs);     % close serial port attached to old Lidar
    delete(serialObjs);     % delete old lidar objects
end

% The following code clears up Matlab Workspace of any old work that might
% conflict with your active code
clear
clc
```

Type in the comments as shown, both to get used to doing this and so you can reuse this code later in your final project.

Please add this warning turn off code now, and you can comment tit out later when we have a software fix for the older serial code that's going to be unsupported in some future version of Matlab (this older serial code triggers the warning)

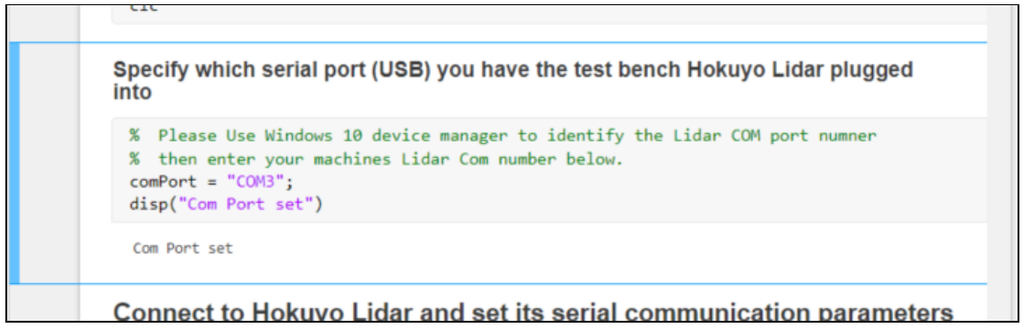
This code clears up any old objects hanging around to give you code conflicts as you create new ones.

Always, Always clear your Matlab workspace before running any program!

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Run the first section. Should have no errors.

Next write a sort new section to specify what USB serial port the Lidar is plugged into:



Specify which serial port (USB) you have the test bench Hokuyo Lidar plugged into

```
%  Please Use Windows 10 device manager to identify the Lidar COM port numner
%  then enter your machines Lidar Com number below.
comPort = "COM3";
disp("Com Port set")
```

```
Com Port set
```

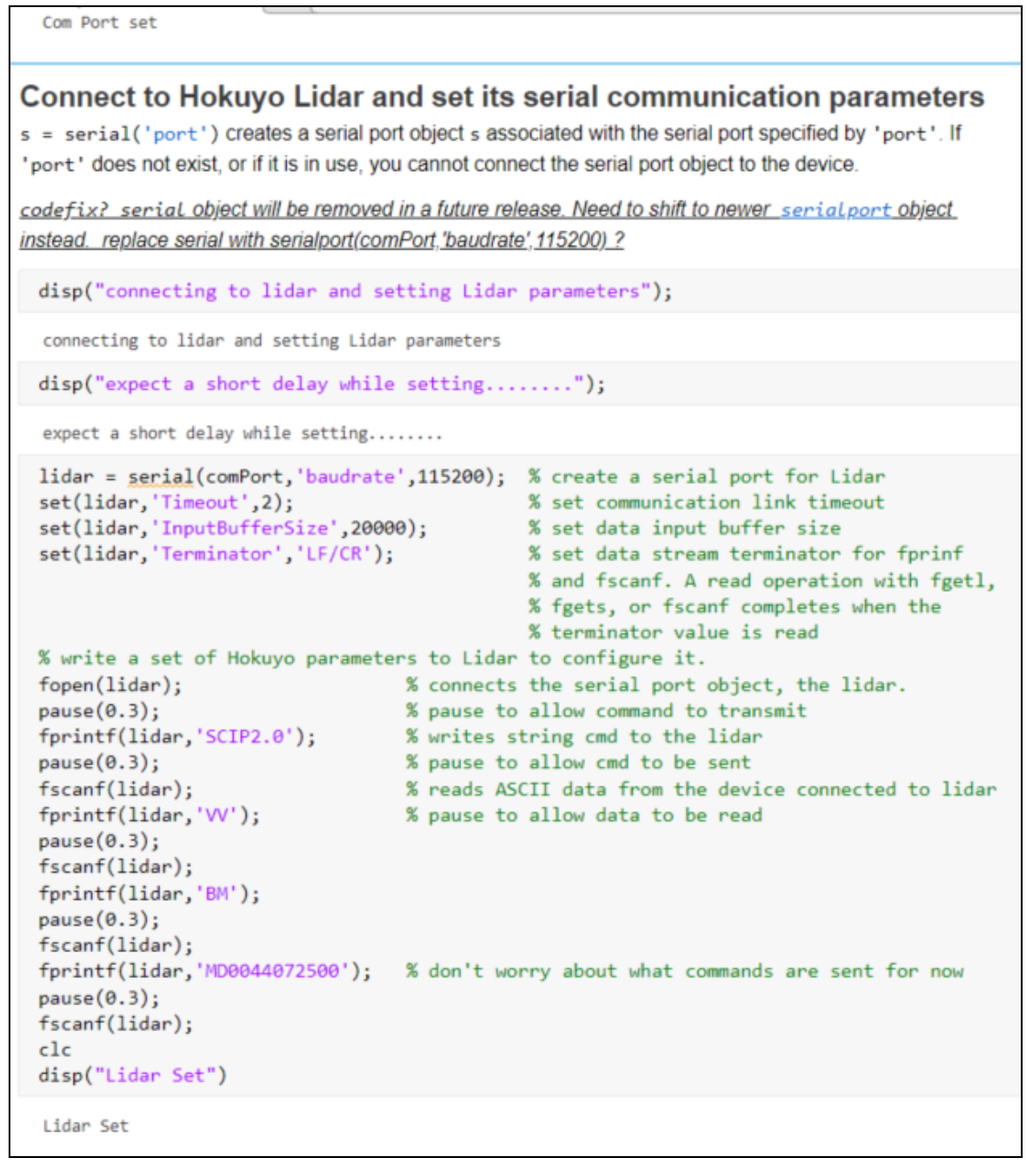
Connect to Hokuyo Lidar and set its serial communication parameters

This should all just run cleanly too.

Next we will write a new section that creates a lidar object for the Hokuyo, configures the serial communication link between your laptop and the lidar and sets this link up to send back range data.

The whole serial communications world is a bit arcane with text being sent like ‘SCIP2.0’ and ‘MD0044072500’. This usually falls into the sub-world of serial communication device drivers and is heavily driven by whatever complexity the device manufacturer wants to build into it. These serial data messages are usually ASCII code based, with the laptop sending some ASCII text to the device and then device sending some back. It’s much like sending text messages to a friend (LOL) and then needing to decode what they sent back to you (1174). In our case, the ASCII codes are fully described in the Hokuyo Lidar Communications specification document.

Please create a new section in your lidar script and add the following code:



```
Com Port set
```

Connect to Hokuyo Lidar and set its serial communication parameters

`s = serial('port')` creates a serial port object `s` associated with the serial port specified by `'port'`. If `'port'` does not exist, or if it is in use, you cannot connect the serial port object to the device.

codefix? serial object will be removed in a future release. Need to shift to newer serialport object instead. replace serial with serialport(comPort,'baudrate',115200) ?

```
disp("connecting to lidar and setting Lidar parameters");
```

```
connecting to lidar and setting Lidar parameters
```

```
disp("expect a short delay while setting.....");
```

```
expect a short delay while setting.....
```

```
lidar = serial(comPort,'baudrate',115200);  % create a serial port for Lidar
set(lidar,'Timeout',2);                     % set communication link timeout
set(lidar,'InputBufferSize',20000);         % set data input buffer size
set(lidar,'Terminator','LF/CR');            % set data stream terminator for fprinf
                                            % and fscanf. A read operation with fgetl,
                                            % fgets, or fscanf completes when the
                                            % terminator value is read
% write a set of Hokuyo parameters to Lidar to configure it.
fopen(lidar);                   % connects the serial port object, the lidar.
pause(0.3);                     % pause to allow command to transmit
fprintf(lidar,'SCIP2.0');       % writes string cmd to the lidar
pause(0.3);                     % pause to allow cmd to be sent
fscanf(lidar);                  % reads ASCII data from the device connected to lidar
fprintf(lidar,'VV');            % pause to allow data to be read
pause(0.3);
fscanf(lidar);
fprintf(lidar,'BM');
pause(0.3);
fscanf(lidar);
fprintf(lidar,'MD0044072500');  % don't worry about what commands are sent for now
pause(0.3);
fscanf(lidar);
clc
disp("Lidar Set")
```

```
Lidar Set
```

In short order, the Matlab warning we are suppressing is tied back to the fact that it is phasing out the **serial** object in its code base and replacing it with a newer (maybe better?) **serialport** object and this old code uses the older **serial** one. The two may not be one-to-one compatible, so for now, for course needs, we will stick with the older **serial** and perhaps one of this year's students can loop back and see if they can upgrade the whole class to serialport.

Next we create **lidar** as a serial object and configure it to allow serial communication with Hokuyo over its USB port.

Finally we open that link and send a bunch of coded text messages to it that configure the lidar to let it send us back a set of range data. Please don't get hung up on the ASCII configuration messages at this point. You can loop back and dive in at the configuration level later if you want to optimize these settings for final project use.

Run section, if all goes well you will get **"Lidar Set".**

Please note: At this point you have created a serial object and opened a communication path to it. You can't back up and rerun this code without really jamming things up! You need to carefully close the serial port and flush buffers before you can reopen it, which we will do in the final section of the lidar code. For now, just always run code from the first section down to where you are currently working while you are developing.

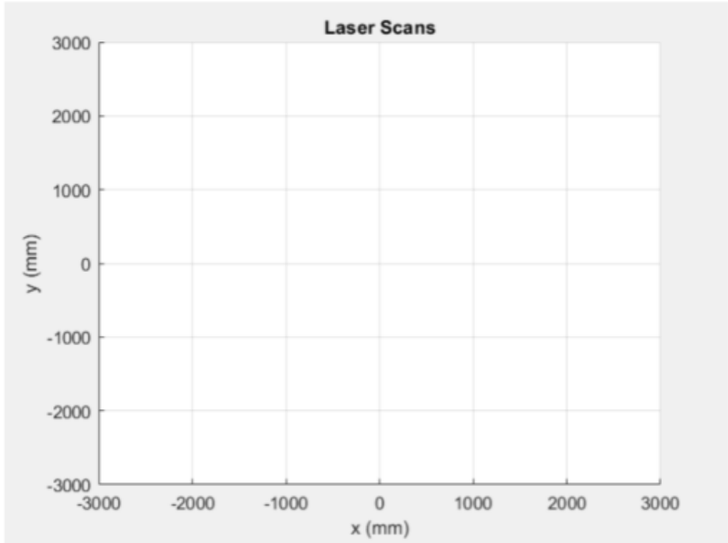
Next, create a new section and add the following code to create a stand alone figure to plot collected Lidar data in (see opposite).

The first set of code creates and labels the lidar plotting window, the second bit creates a graphic line primitive that will be used to draw the lidar data in the section to follow this one.

Please add code as shown:

Initialize the Lidar display figure window

```
% figure creates a stand-alone figure window, The resulting figure is the current figure
% for all plots until you change it.
LaserPlot1.figure = figure('Name','Hokuyu URG-04LX data','NumberTitle','off',...
    'MenuBar','figure','units','normalized','Visible','on');
LaserPlot1.axis1=axes('parent',LaserPlot1.figure ,'units','normalized','NextPlot','replace');
grid(LaserPlot1.axis1,'on')
LaserPlot1.axis1.Title.String='Laser Scans';
LaserPlot1.axis1.XLabel.String = 'X Axis';
LaserPlot1.axis1.YLabel.String = 'Y Axis';
% create a primative line object to use plotting lidar data
% XData and YData are empty vectors waiting to be filled with Lidar data
laserRange = line('Parent',LaserPlot1.axis1,'XData',[],'YData',[],'LineStyle','none',...
    'marker','.','color','b','LineWidth',2);
grid on
axis([-3000 3000 -3000 3000])
xlabel ('x (mm)')
ylabel ('y (mm)')
```




```
disp("Laser Scans figure set");
```

```
Laser Scans figure set
```

This code will also create a free-floating **Laser Scans** plot window.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Almost done, create a new section and add the code needed to read lidar range data over the serial link and then to plot it on the figure. Please add code shown below:

Sample and Visualize Data

read Lidar over USB port, parse data, create a plot of data in a 60 sec continuous loop, use CTRL-c to stop early

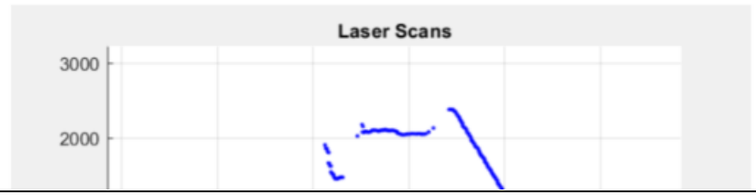
Note: Please don't pause and leave this section of the code, as it needs to complete and then go through a clean shutdown in next section to avoid jamming the communication link to the Lidar

Note: Please download the two support Matlab functions FunRoboLidarScan.m and decodeSCIP.m from Canvas and then place them in the working folder with this code before proceeding

```
disp("Read and Plot Lidar Data, type and hold cntrl-c to stop")
```

```
Read and Plot Lidar Data, type and hold cntrl-c to stop
```

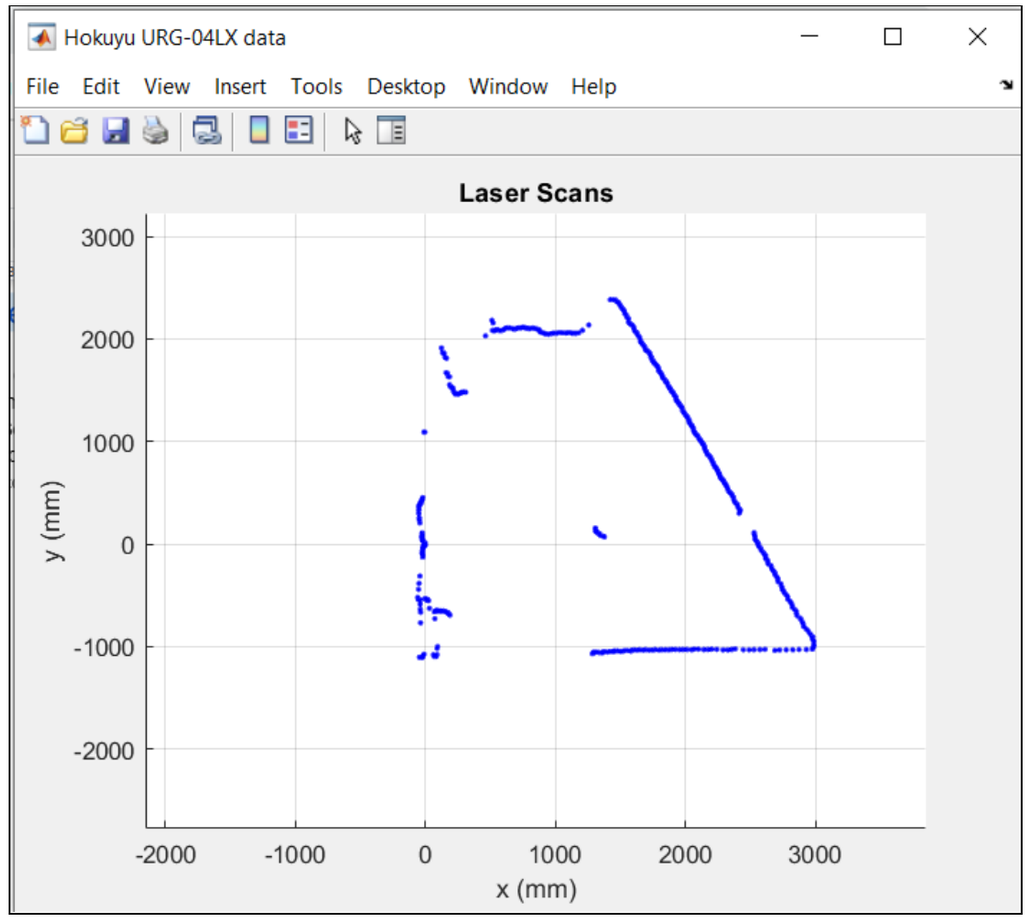
```
angles=(-120:240/682:120-240/682)*pi/180; % Convert Sensor steps to angles for plotting
tStart = tic;                             % start experiment timer
iscan=1;
while (iscan == 1)                        % continous loop, type and hold cntl-c to break
    [A]=FunRoboLidarScan(lidar);          % actual lidar scan range data stored in [A]
    laserRange.XData=A.*cos(angles);      % Use trig to find x-coord of range
    laserRange.YData=A.*sin(angles);      % Use trig to find y-coord of range
    % drawnow -updates figures and processes any pending callbacks.
    % Use this command if you modify graphics objects and want to see the updates
    % on the screen immediately. Use here to update laser range draws
    drawnow                               % will draw laserRange in open Laser Scan window
    pause(0.2)                            % pause to allow serial communications to keep up
    tElapsed = toc(tStart);               % measure time (in sec) since start of experiment
    if (tElapsed > 60)                    % is experiment goes too long stop sample loop
        iscan =0;
    end
end
```



.

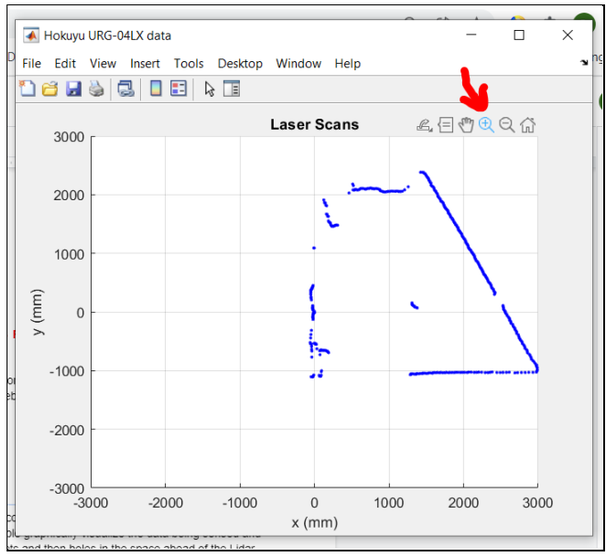
This section sets up a continuous lidar sampling loop that will run for 60 seconds (between tic and toc). You can extend time to anything you'd like and can stop the loop at any time with a Ctrl-C.

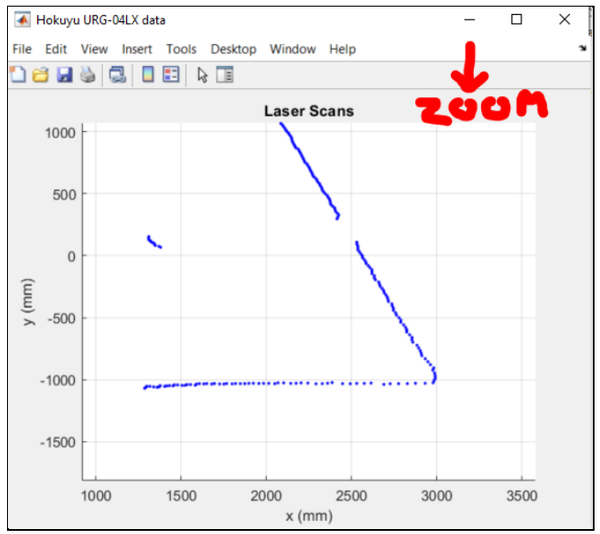
The first part of the code converts the data vector cell positions to angles. The next part **tic** starts the experimental clock. The **while** loop asks for a laser scan via the function **FunRoboLidarScan** (Please download the two support Matlab functions FunRoboLidarScan.m and decodeSCIP.m from Canvas and then place them in the working folder with this code before proceeding) does some trigonometry and then draws the new set of lidar data in the floating Lidar Scan window:



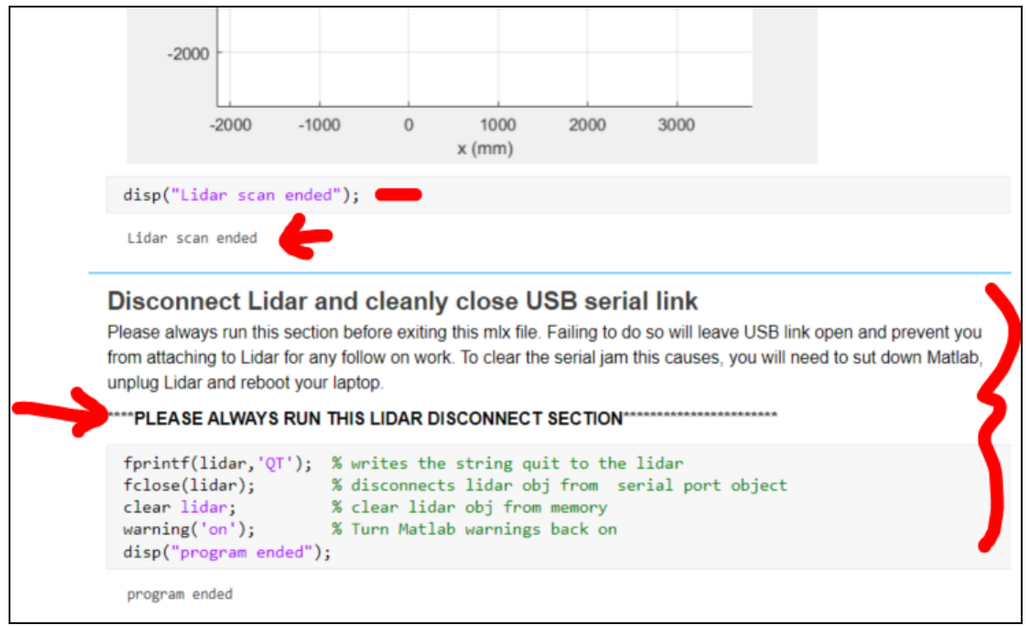
ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2022d

Please note: if you place cursor over figure plotting area, you can use tools like zoom to move and zoom figure:





Finally you just need to add a last section that closes lidar serial communications, closes the lidar serial port object and shuts down all cleanly. Please create a new section and add the code shown below:



Now, run your completed code and place a variety of target objects in front of lidar. What do you see in your lidar scan window? Does lidar work the same with different colors? How about with reflective objects? Is the calibration correct? Get a ruler and check.

Note: ******** **PLEASE ALWAYS RUN LAST SECTION TO CLOSE LIDAR LINK CLEANLY** **********

If you don't, you will jam the Com port lidar link and need to shut down your computer, unplug Lidar and then reboot and replug all to get running again.

Visualization

Once the prototype LIDAR scan code is up and running, your goal is to develop a MATLAB script that will both be able graphically visualize the data being sensed and then use that data to find first targets and then holes in the space ahead of the Lidar

Each Lidar scan is basically a line vector of target ranges. MATLAB has a rich collection of functions that let you find minimums, maxes, step-like changes in a signal, etc. Take a look through the library of signal processing options and find a set of functions or write your own algorithm to add perception to the sense data your Lidar coleded.

One possibility is a map showing the total sensor data, overlaid with the highlighted sections that the robot has identified as holes and a marker (squares, circles etc) at the location where the robot has identified an object. You can also scan the space before Lidar with no targets, scan with targets and use some form of vector difference techniques to find where targets are.

A big stretch goal here is to write a bit of MATLAB arduino code that will let you tilt the existing 2d Lidar to one angle, scan, tilt to the next angle, scan and repeat this to collect a 3d point cloud of data. With a 3d set of data you can easily tell the target objects apart. Please see Instructor or Ninjas for help if you would like to take this on.

Final Demos:
Using your collection of new MATLAB Lidar code tools, and liberal use of the **disp("message")** statement, please perform the following two tasks for your final Sens Lab 2 demo:

Find The Target:
Ninja will place a set of fixed targets on the turntable in front of the Lidar. Your sub team's mission is to find them, tell them apart and give a clean bearing measurement to them. (i.e: target at 45 deg, target at 90 deg, etc,)

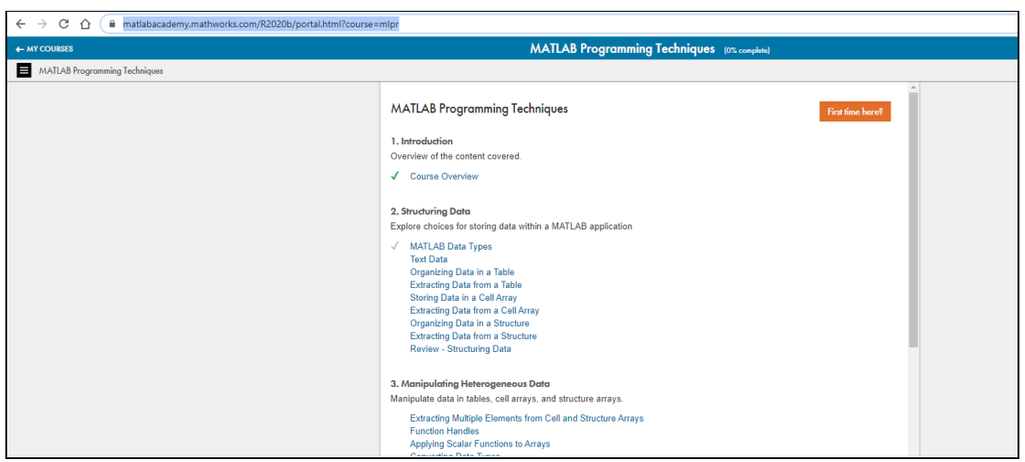
Find The Hole:
Ninja will place a collection of targets on the turntable in front of the Lidar. Your mission is to find the bearing to the largest hole between them for your robot to drive through. (i.e. open ground between 60 deg and at 45 degrees).

Bonus points for a scanning Lidar demo. Extra bonus points if you can use the App building tools to turn this into an App.

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.

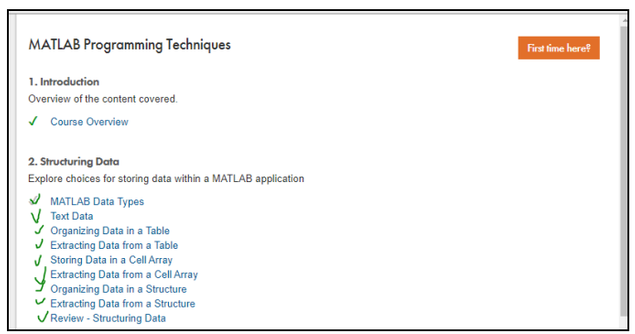
Please see a course instructor or Ninjas for MATLAB help as needed.

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