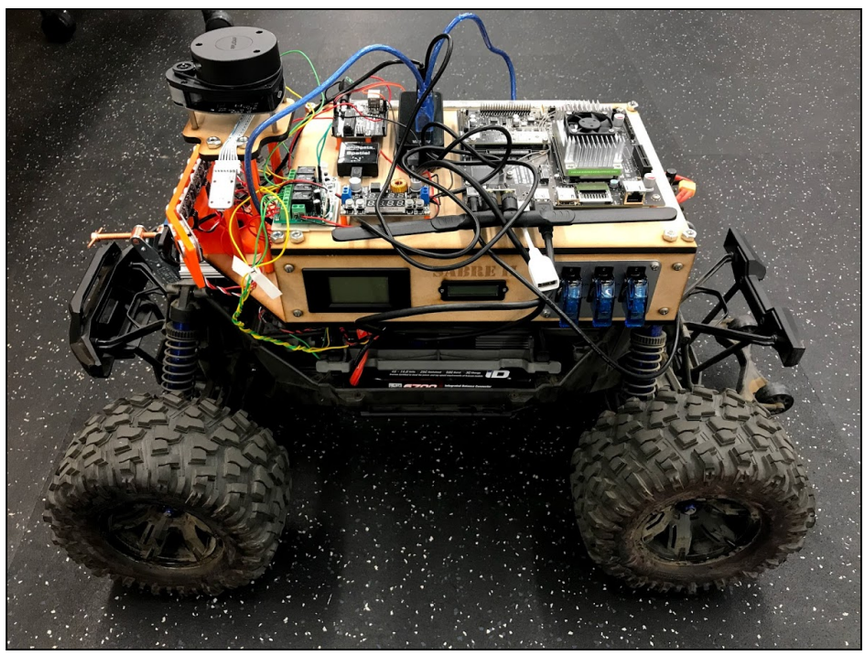
ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

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.

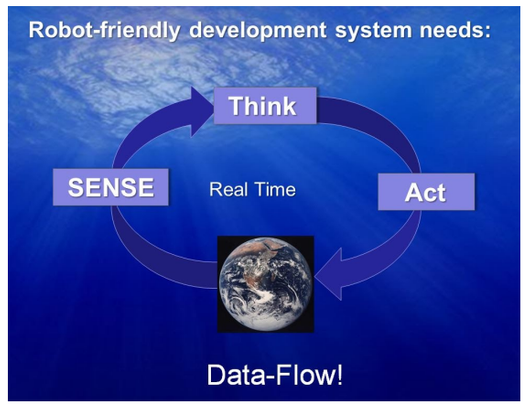


Figure 1 : Sense-Think-Act Control Loop

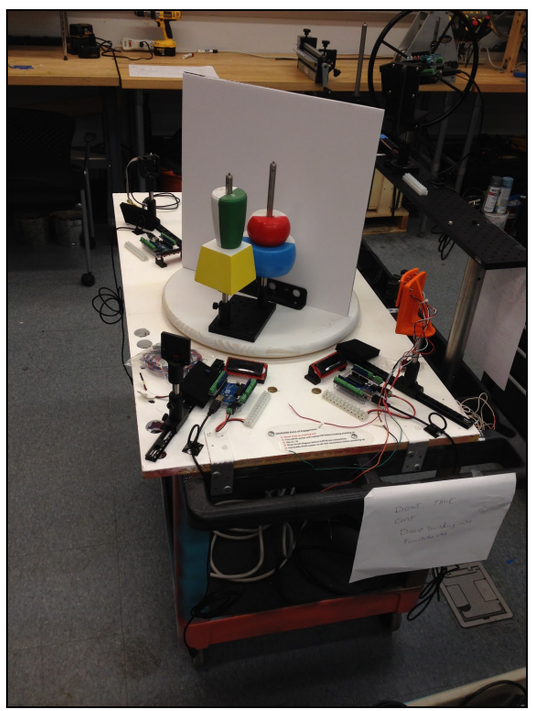
SENSE: involves adding sensors to a robot to answer three primary questions; “Where am I?”, “What is around me?” and “How am I?”. The art of robot perception is to develop your skills in choosing the appropriate sensor, learning to electrically hook it up into your robots existing circuitry, collecting 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.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

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.



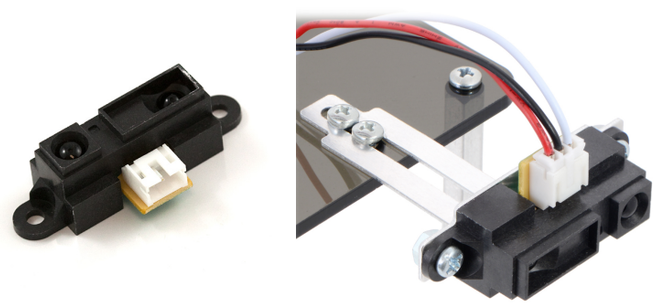
Sense Lab 1 Hardware

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.

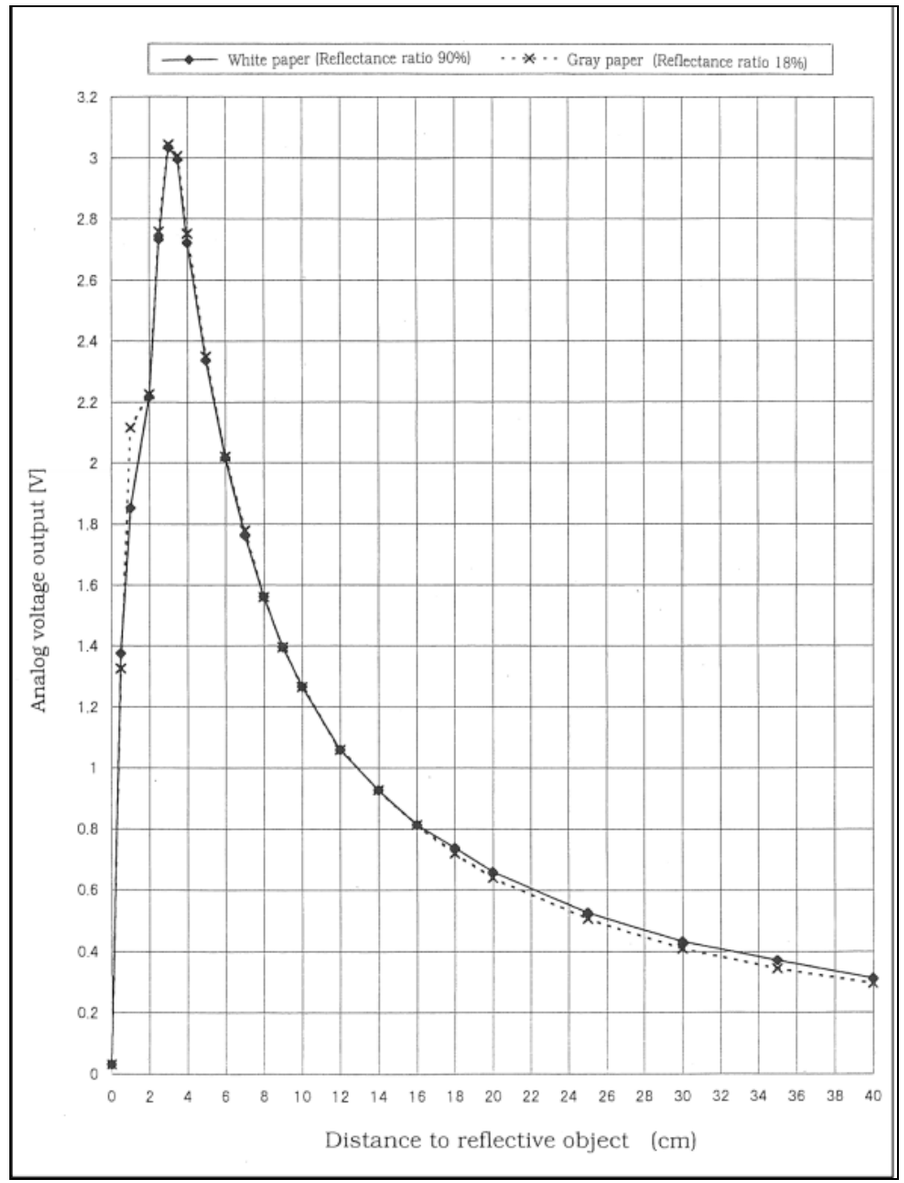
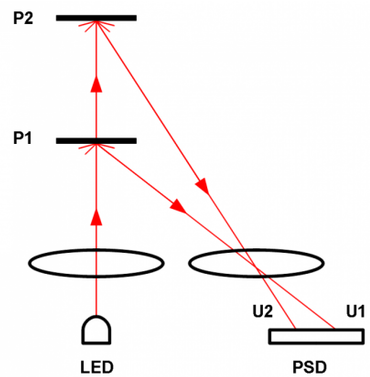


Figure 3: Analog Voltage vs. Distance

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.

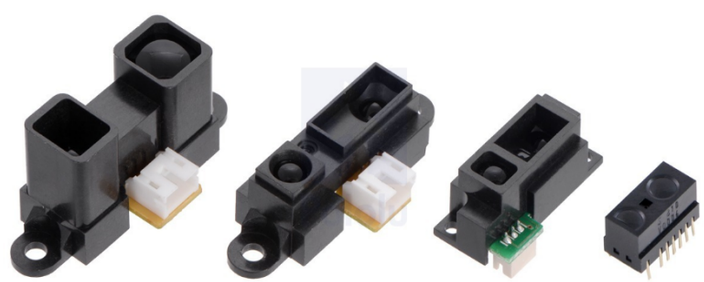
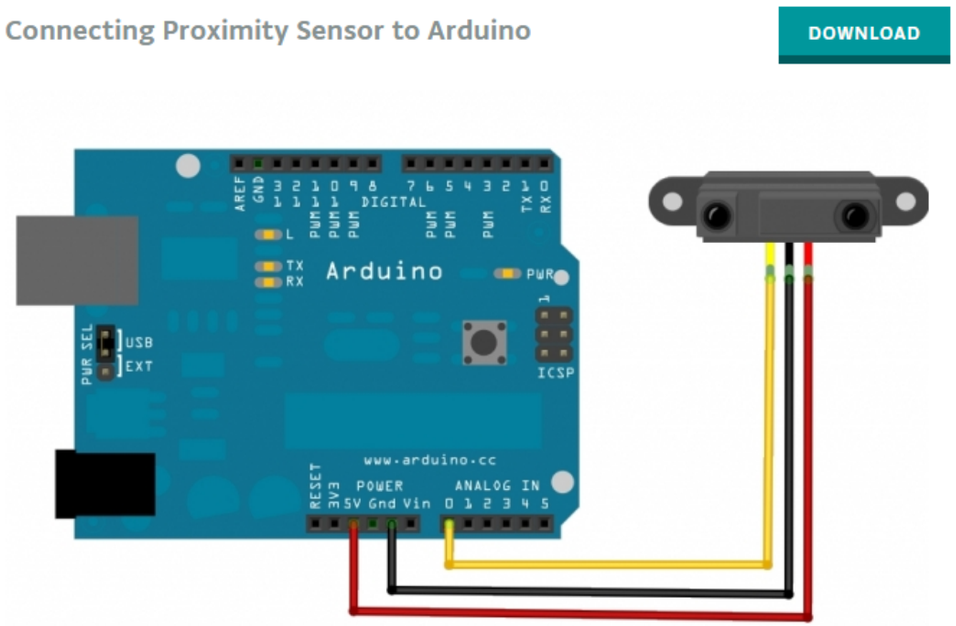


Figure 5: Family of Sharp IR Range Sensors

Connecting Sharp IR Sensors to and 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:

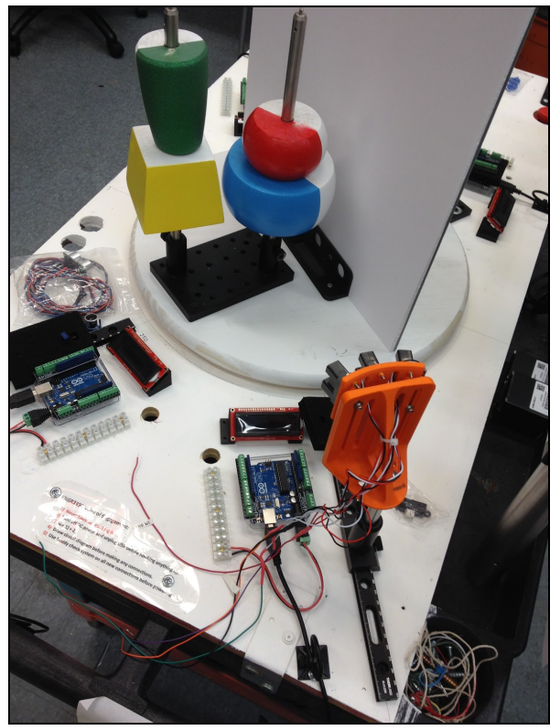
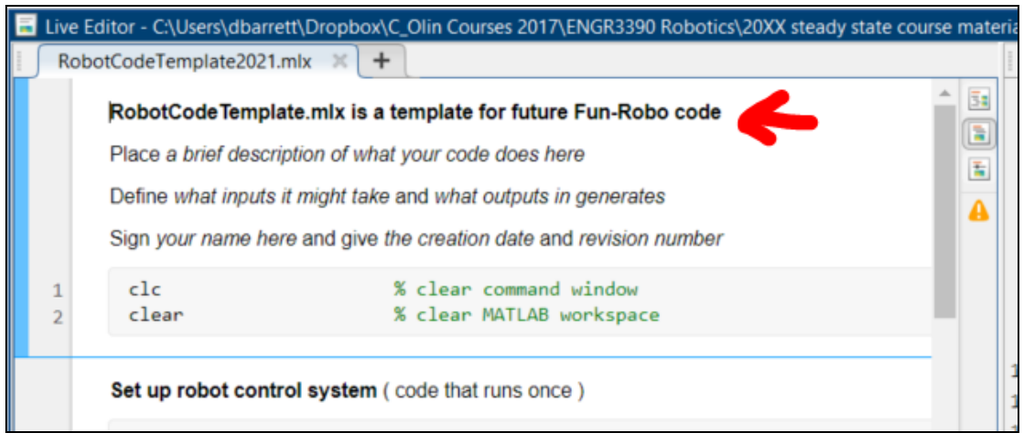


Figure: Sharp IR test station

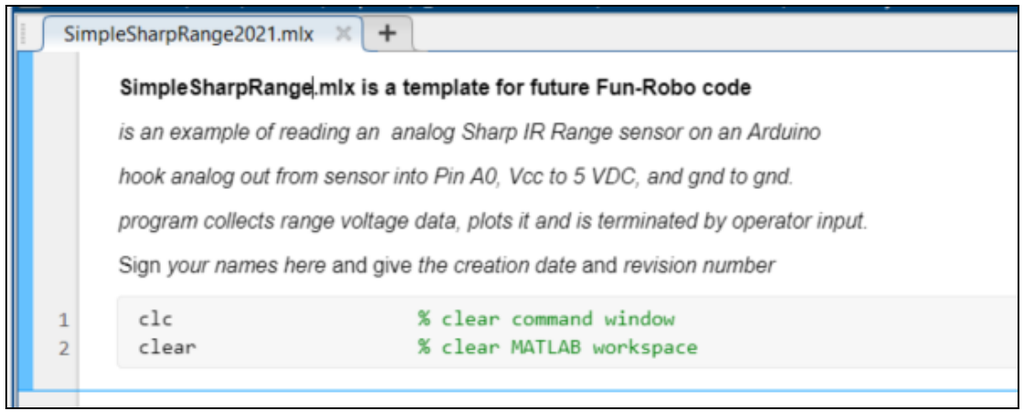
Getting the raw data into the Arduino is relatively easy, but you will quickly find that you are going to have to carefully calibrate each type of Sharp Range Sensor, you might need a different calibration curve for different color targets and you are going to need to find a way to weave all of that 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.

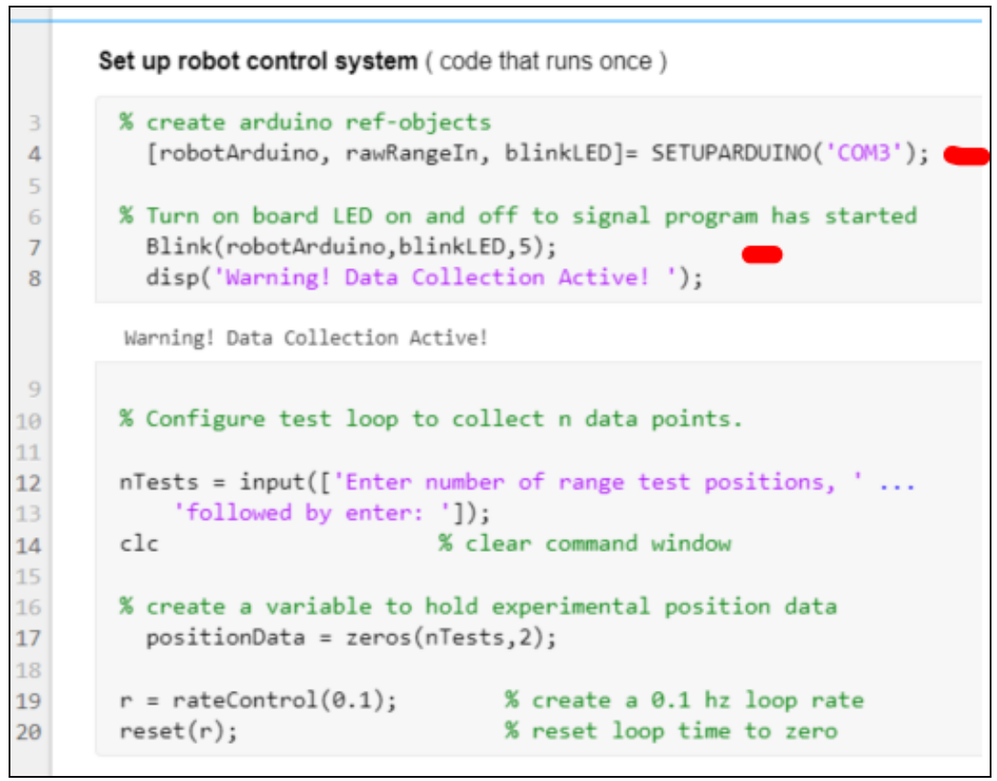
Let's take on each of these tasks sequentially. First open up a copy of your Matlab Robot Template:



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) 2021

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) 2021

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 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) 2021

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

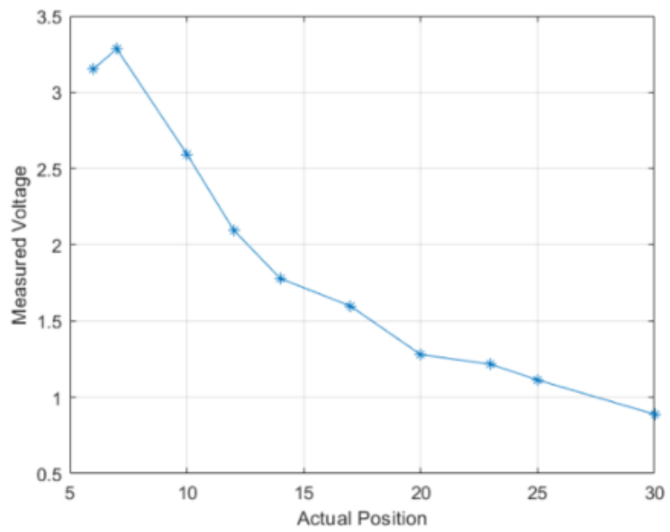
Mission data processing

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
```

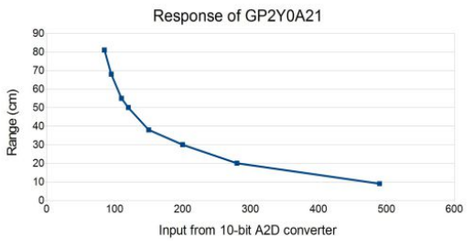
```
48    clear robotArduino
49    beep              % play sytem sound to let user know program is ended
```

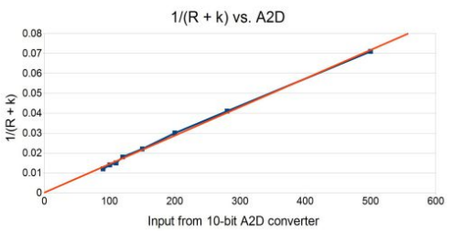
Robot Functions (store this codes local functions here)

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.

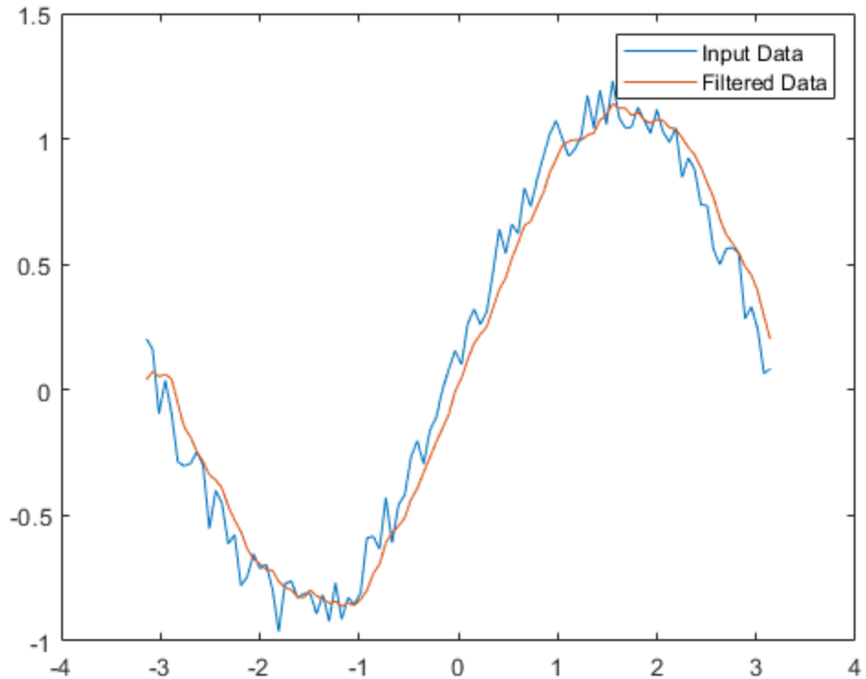




3). **Filtering?** Next, write a MATLAB script to collect 10 range samples at a single distance from white backplane target about one second apart. Record them in a rangeDataOvertime variable matrix along with the sample time of each measurement and plot them. Does the data vary with time? If so, you 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 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.

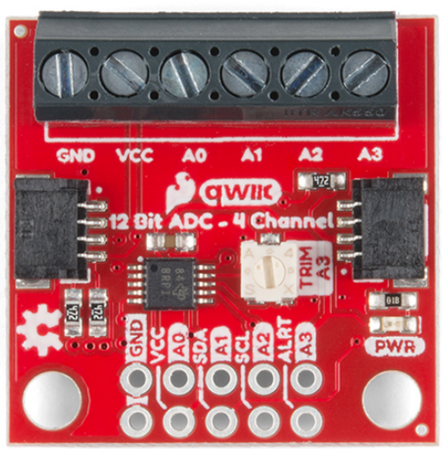
As always, please see an instructor or Ninjas for help with any part of the above work.

Sharp IR Range Sensors with a Raspberry-Pi

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 to your Arduino code to port it to your Raspberry Pi.

To add additional capability to your Pi we will be using the SparkFun Qwiic fast prototyping system.SparkFun's Qwiic Connect System uses 4-pin JST connectors to quickly interface development boards with sensors, LCDs, relays and more.You can see https://www.sparkfun.com/qwiic for more details.

In this lab we will be using the **Qwiic 12-Bit ADC** to add 4 new Analog Ins to your Pi:



The SparkFun Qwiic 12 Bit ADC can provide four channels of I2C controlled ADC input to your Qwiic enabled project. These channels can be used as single-ended inputs, or in pairs for differential inputs. What makes this even more powerful is that it has a programmable gain amplifier that lets you "zoom in" on a very small change in analog voltage (but will still affect your input range and resolution). Utilizing the handy Qwiic system, no soldering is required to connect it to the rest of your system. However, we still have broken out 0.1"-spaced pins in case you prefer to use a breadboard.

The ADS1015 uses its own internal voltage reference for measurements, but a ground and 3.3V reference are also available on the pinouts for users. This ADC board includes screw pin terminals on the four channels of input, allowing for solderless connection to voltage sources in your setup. It also has an address jumper that lets you choose one of four unique addresses (0x48, 0x49, 0x4A, 0x4B). With this, you can connect up to four of these on the same I2C bus and have sixteen channels of ADC. The maximum resolution of the converter is 12-bits in differential mode and 11-bits for single-ended inputs. Step sizes range from 125µV per count to 3mV per count depending on the full-scale range (FSR) setting.

They have included an onboard 10K trimpot connected to channel A3. This is handy for initial setup testing and can be used as a simple variable input to your project. But don't worry, we cut the isolation jumper so you can use channel A3 however you'd like.

An analog to digital converter (ADC) is a very useful tool for converting an analog voltage to a digital signal that can be read by a microcontroller. The ability to convert from analog to digital interfaces allows users to use electronics to interface to interact with the physical world.

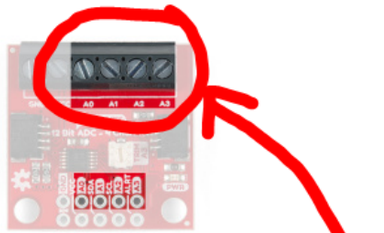
There is a power status LED to help make sure that your Qwiic ADC is getting power. You can power the board either through the polarized Qwiic connector system or the breakout pins (3.3V and GND) provided. This Qwiic system is meant to use 3.3V, be sure that you are NOT using another voltage when using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

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

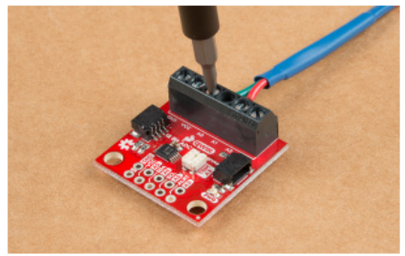
Caution: The analog input voltage range is (GND - .3V) to (VDD + .3V); anything slightly higher or lower and you will damage the ADC chip. If you are using the Qwiic system, this is approximately -.3V to 3.6V in reference to the GND pin. If the voltages on the input pins can potentially violate these conditions, use external Schottky diodes and series resistors to limit the input current to safe values (as mentioned in the datasheet).

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.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021



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



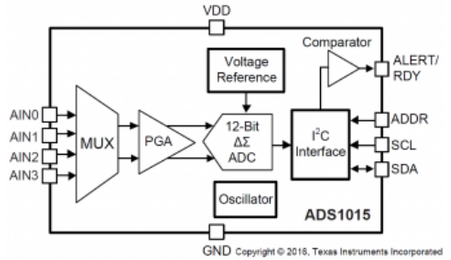
Example of attaching inputs through screw terminals. Click to enlarge.

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-capactior 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.

Note: The ADS1015 has an integrated voltage reference; **an external reference voltage cannot be used**.

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 below.

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
I^2C Address	**0x48** (**Default**), 0x49, 0x4A, or 0x4B

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.

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.

Duty Cycling

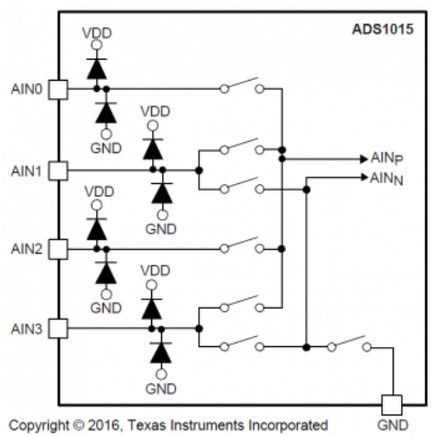
In single-shot mode, the ADS1015 can be duty cycled to periodically request high data rate readings. This emulates an intermediary configuration between the low power consumption of the single-shot mode and the high data rates of the continuous-conversion mode.

Continuous-Conversion

In this mode, the ADS1015 continuously performs conversions on analog voltage measurements and places the data in the conversion register. If the configuration settings are changed in the middle of the conversion process, the settings take effect once the current process is completed.

Input Multiplexer (MUX)

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.

There are 8 MUX configurations to designate the analog voltage inputs to the ADC of the ADS1015, shown in the table below. The default configuration of the ADS1015 uses inputs AIN_0 and AIN_1 as highlighted in bold in the table below.

Inputs	MUX Configurations							
	Differential				Single-Ended			
AIN_P	**AIN_0**	AIN_0	AIN_1	AIN_2	AIN_0	AIN_1	AIN_2	AIN_3
AIN_N	**AIN_1**	AIN_3	AIN_3	AIN_3	GND	GND	GND	GND

Programmable Gain Amplifier

A programmable gain amplifier (PGA) is implemented before the ΔΣ ADC. The ADS1015 has 6 programmable gain settings, which are expressed in the full-scale range (FSR) of the ADC scaling. The maximum analog measurement is then defined by the smaller of the FSR or VDD. By default, the ADS1015 has a resolution of 1mV by using an FSR of ±2.048V as highlighted in bold in the table below.

Gain:	16	8	4	**2**	1	2/3
Resolution (LSB):	0.125 mV	0.25 mV	0.5 mV	**1 mV**	2 mV	3 mV
FSR (12-bit):	±256 mV	±512 mV	±1.024 V	**±2.048 V**	±4.048 V	±6.144 V

Analog-to-Digital Conversion

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.

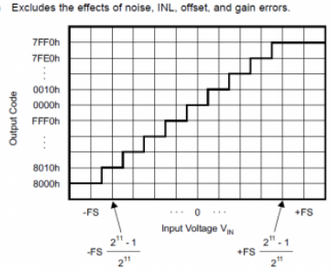


Figure 18. Code Transition Diagram

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.

Explanation of how data output and sign bit work from datasheet.

The ADS1015 offers 7 selectable output data rates of 128 SPS, 250 SPS, 490 SPS, 920 SPS, 1600 SPS, 2400 SPS, or 3300 SPS. Conversions for the ADS1015 settle within a single cycle; thus, the conversion time is equal to 1/DR.

Limitation of ADS1015

Since the ADS1015 only uses an internal reference voltage, the FSR is to be defined by the design of the 12-bit ADC:

FSR = LSB x 212
where, the LSB = 0.125, 0.25, 0.5, 1, 2, or 3 mV.

Due to the configuration options, it is difficult to make full use of the full-scale range of the ADS1015 with common (useful) voltages. See the examples below for a more detailed explanation:

If the FSR = 2.048V and VDD = 3.3V:

Resolution of the digital data: 1mV (defined by FSR)
Input voltage range: 0-3.3V (defined by VDD)
Data Range: 0000h-7FF0h (HEX) or 0-2.048V
The range of the input voltage that can be read by the ADC is limited by FSR. Any voltage higher than the FSR (but less than VDD) reads the same maximum value in the digital output because the FSR is maxed out. In this case, you are maximizing the resolution (use of the data output), but not the full, allowable range of analog input (0-3.3V).
If the FSR = 4.096V and VDD = 3.3V:

Resolution of the digital data: 2mV (defined by FSR)
Input voltage range: 0-3.3V (defined by VDD)
Data Range: 0000h-0672h (HEX) or 0-3.3V
The data range is limited by VDD, any higher input voltage will continue to have data up to the electrical specifications of the ADS1015 ~VDD + 0.3V, where the IC gets damaged. In this case, the input voltage is being maximized to the electrical

specification (0-3.3V). However, this means you are effectively only using 80% of the full resolution of the ADC (i.e. a 10.8-bit ADC at 3.3V).
Data Rate & Conversion Time

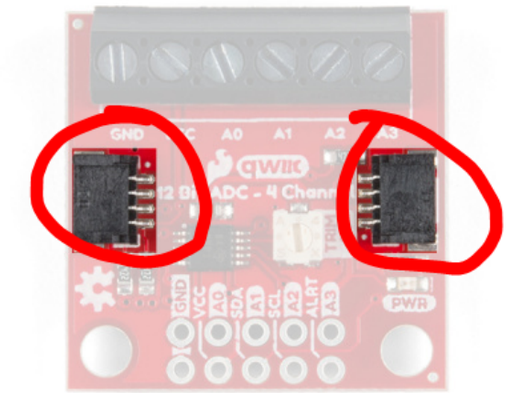
Qwiic or I2C I2C Address
The ADS1015 has 4 available I2C addresses, which are set by the address pin, ADDR. On the Qwiic ADC, the default slave address of the ADS1015 is 0x48 (HEX) of 7-bit addressing, following I2C protocol. The ADS1015 does have an additional general call address that can be used to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

Default I2C Slave Address: 0x48

I2C Registers
The ADS1015 has four 16-bit registers, which are accessible through the I2C bus using the Address Pointer register. The Address Pointer register is an 8-bit byte that is written immediately after the slave address byte, low R/W bit.

Address	Description
N/A	Address Pointer Register (8-bit): Used to grant R/W access to the four available registers on the ADS1015.
0x00	Conversion Register (16-bit): Contains result of last conversion (i.e. measurement).
0x01	Config Register (16-bit): Used the for configuration setting of the ADS1015.
0x02	Low Threshold Register (16-bit): Low threshold value for digital comparator.
0x03	High Threshold Register (16-bit): High threshold value for digital comparator.

Connections
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.)

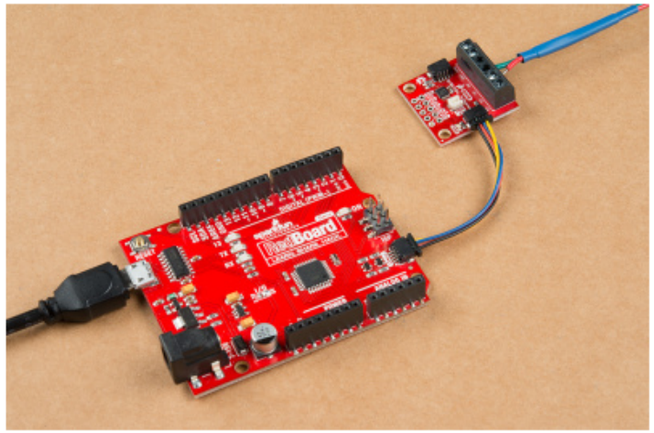


Annotated image of the Qwiic connectors.

Pin Label	Pin Function	Input/Output	Notes
3.3V	Power Supply	Input	**3.3V** on Qwiic system (*should be stable*)
GND	Ground	Input	Ground and Single-Ended Reference Voltage for ADC.
SDA	I^2C Data Signal	Bi-directional	Bi-directional data line. Voltage should not exceed power supply (e.g. 3.3V).
SCL	I^2C Clock Signal	Input	Master-controlled clock signal. Voltage should not exceed power supply (e.g. 3.3V).
ALERT/RDY	Alert/Interrupt	Output	Comparator

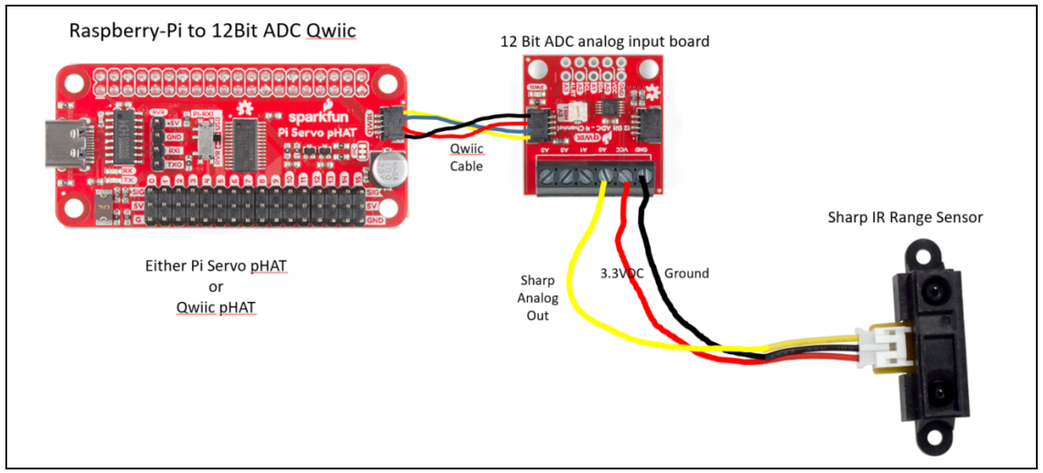
Hardware Assembly
With the Qwiic connector system, assembling the hardware is fairly simple. For the examples below, all you need to do is connect your Qwiic ADC to Qwiic enabled microcontroller with a Qwiic cable. Otherwise, you can use the I2C pins, if you don't

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.



Example setup with RedBoard Qwiic.

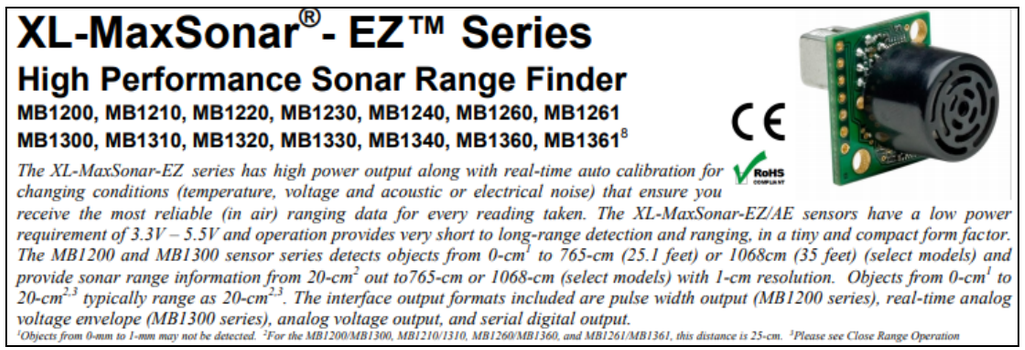
Additionally, you can connect your input voltages to the available inputs on the screw terminals and or use the breakout pins on the board. Make sure the connections are fully inserted and that you are ground looping your inputs. Ground looping can be done by connecting the ground of your input to the ground of the ADC. Please wire up your first Sharp IR as shown below:



Add software here

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).

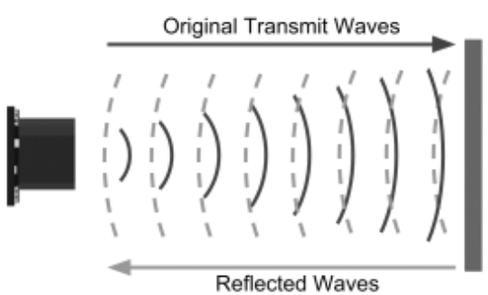
Pin 6 -+5V- Vcc – Operates on 3.3V - 5V. The average (and peak) current draw for 3.3V operation is 2.1mA (50mApeak) and at 5V operation is 3.4mA (100mA peak) respectively. Peak current is used during sonar pulse transmit. Please reference page 4 for minimum operating voltage versus temperature information.

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.

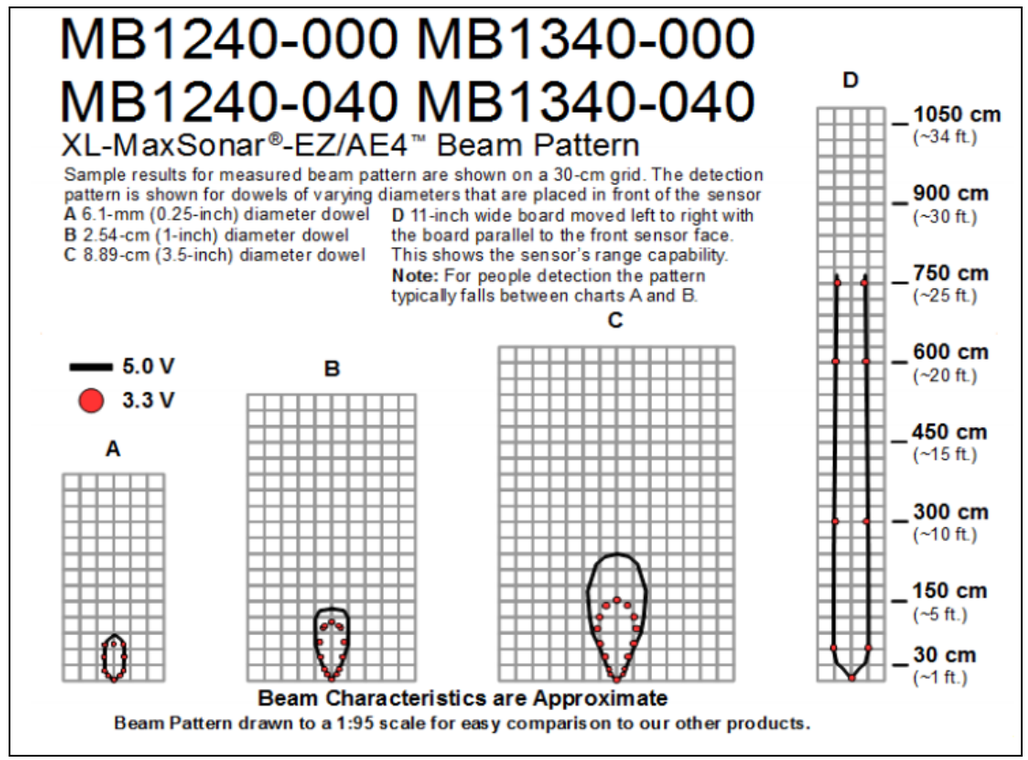
Its data sheet is here:

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.

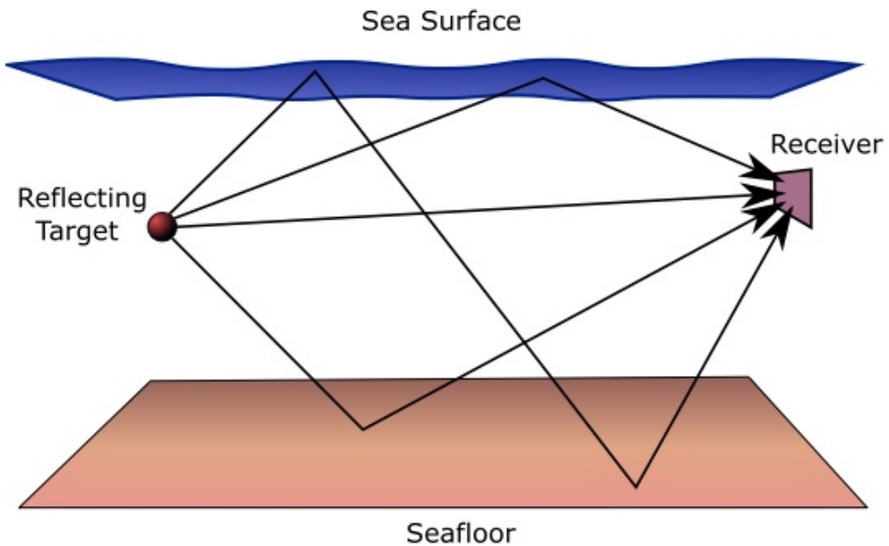


Figure: Example of multipath sonar environment

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:

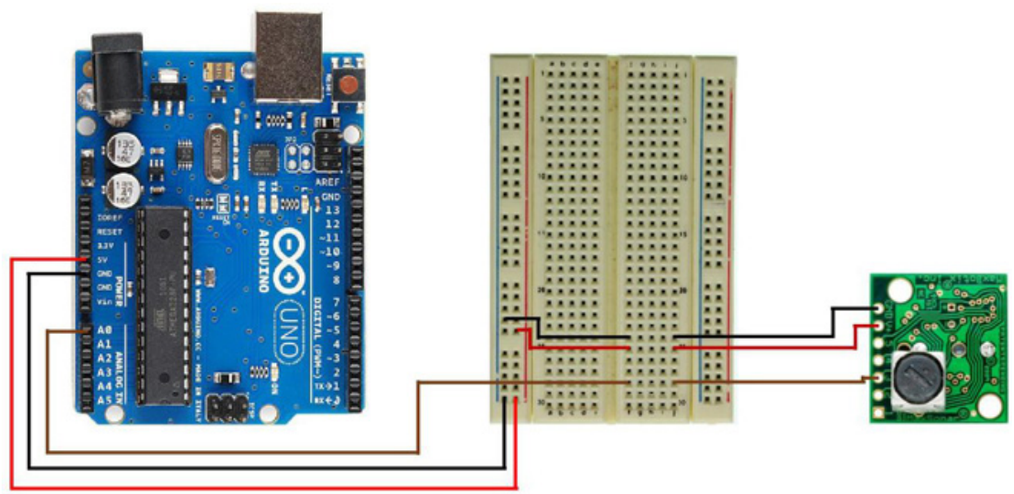
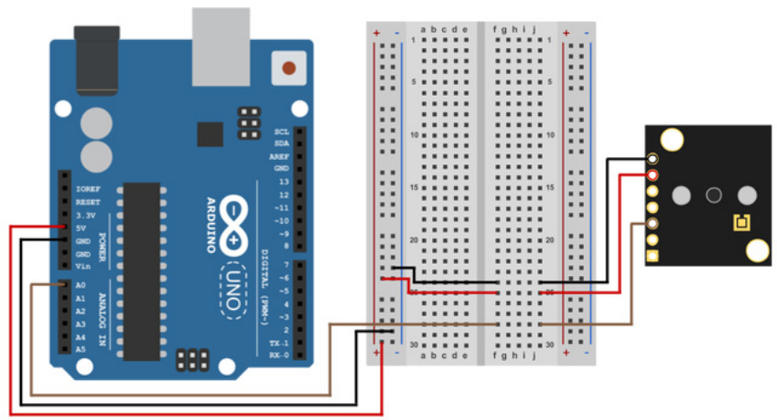


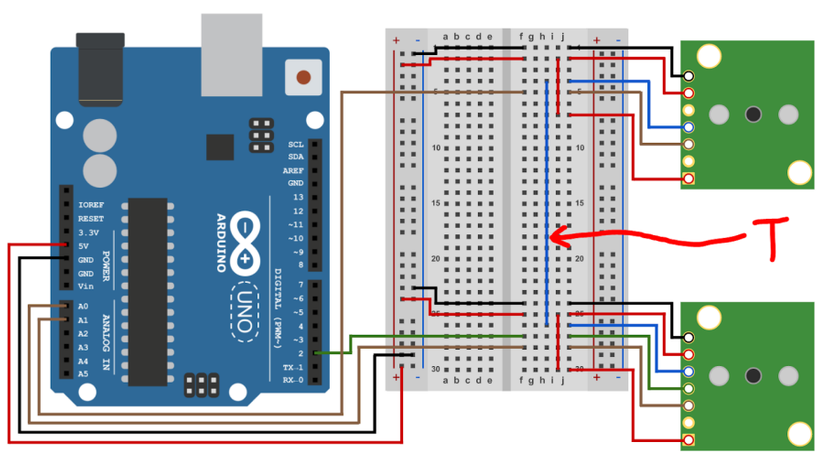
Figure : Arduino MaxSonar wiring diagram (analog input)

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:

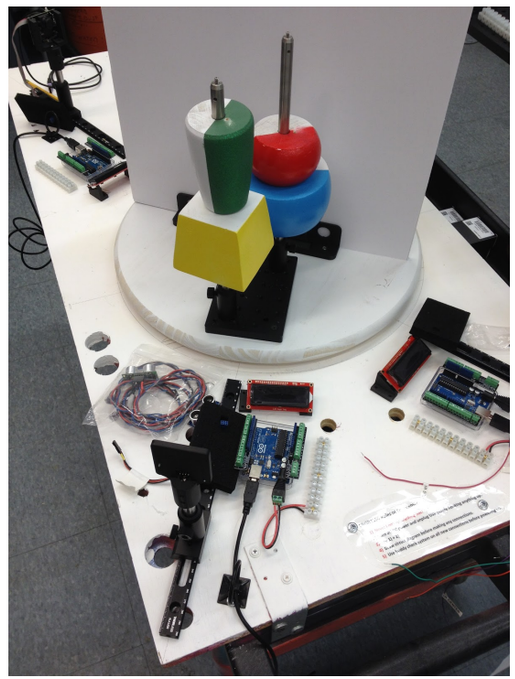


Figure: Sonar sensor test station

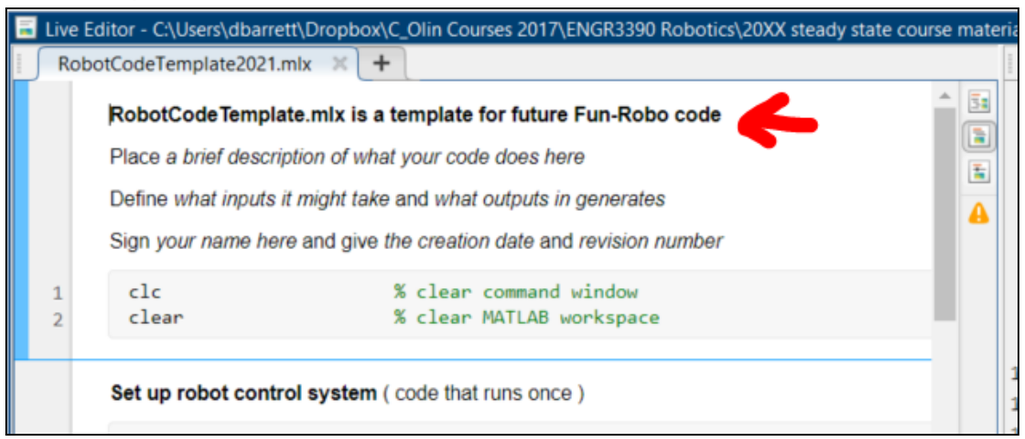
Getting the raw sonar data into the Arduino is relatively easy, but you will quickly find that you are going to have to carefully calibrate each type of sonar sensor, you might need a different calibration curve for different shape targets and you are going to need to find a way to weave all of that 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) 2021

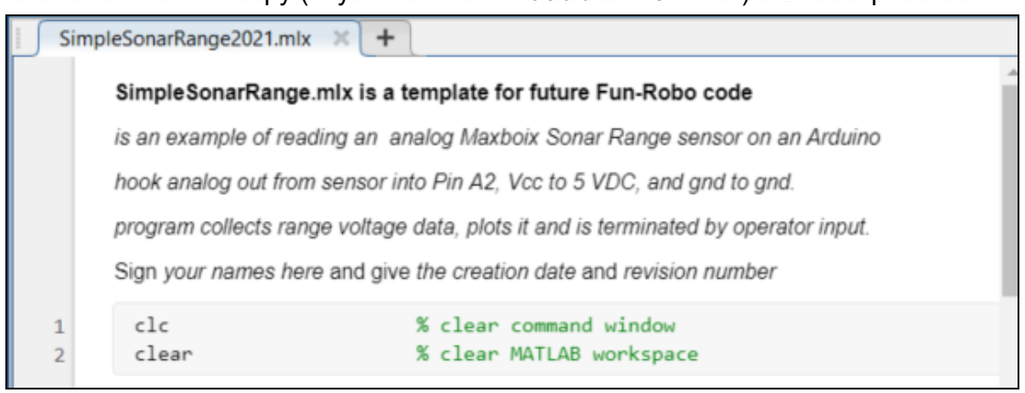
OCU user output section of your MATLAB script program that will print out “Target” or “Hole”.

Let's take on each task sequentially.

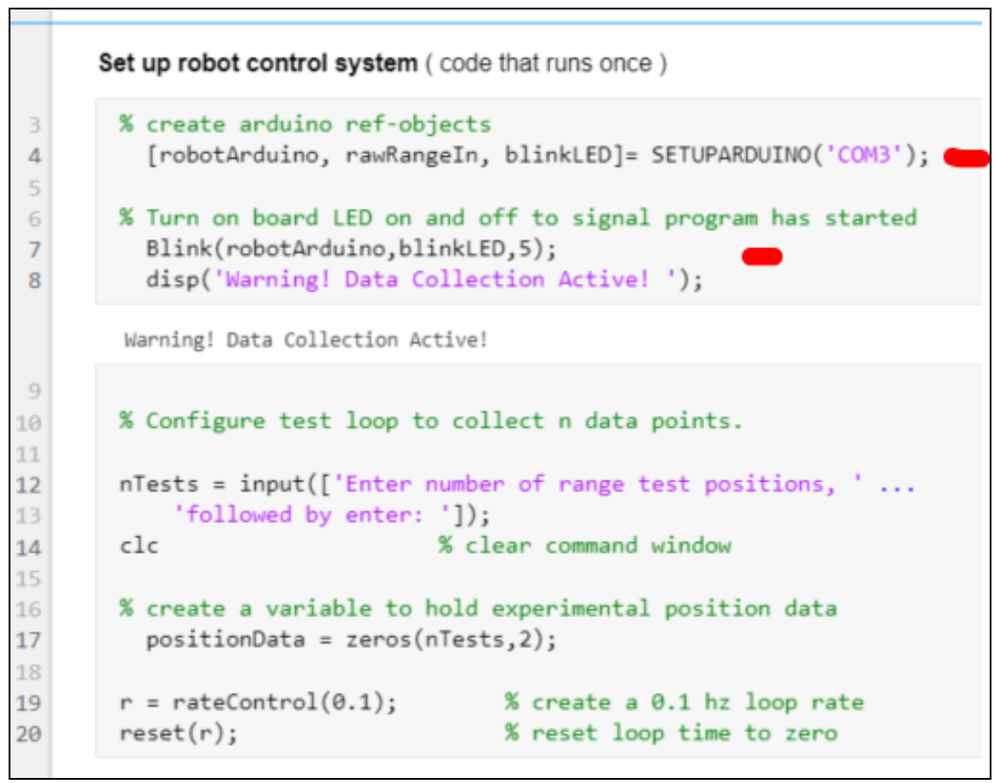
Let's take on each of these tasks sequentially. First open up a copy of your Matlab Robot Template:



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:



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) 2021

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
```

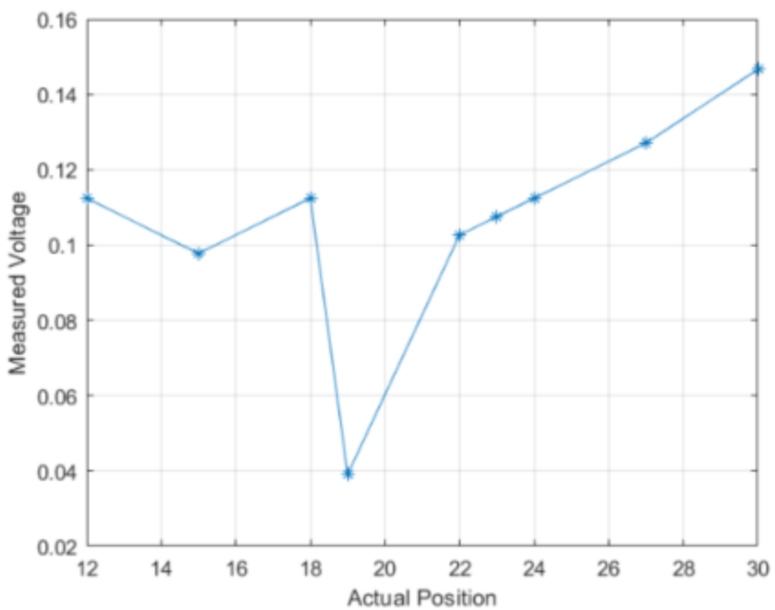
After collecting your data, you will want to process and see it, so please add the following section to make a pretty plot:

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

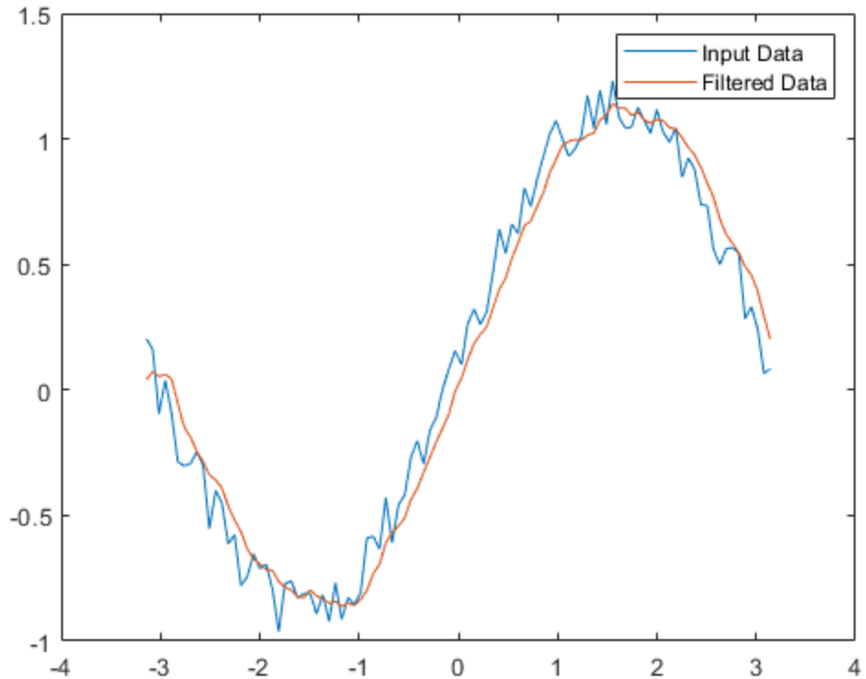
With this code complete, you've done the hard part of sensing, namely getting the raw data in from the environment around the Robot. The next steps are to linearize it and perhaps filter it so that it is clean enough to work with. Then you can calibrate it so that your sense functions produce ranges 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

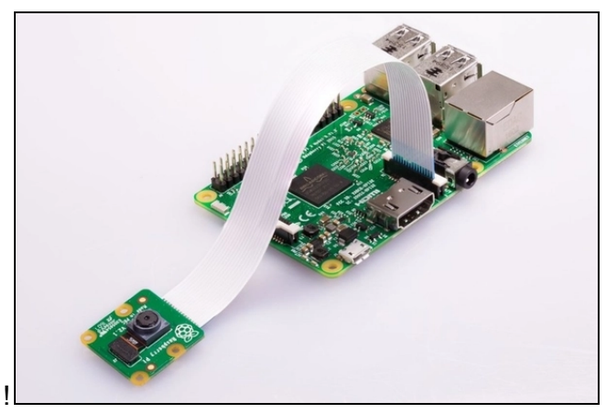
The Raspberry Pi has many awesome talents, but directly reading the output of Analog sensors isn't one of them. The Pi doesn't have analog input ports like an Arduino. To overcome this shortcoming we are going to give you a i2C bus Analog Input device and a pre-written Matlab function to call it. With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

Pi Camera V2 Target Tracking Sensors

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.

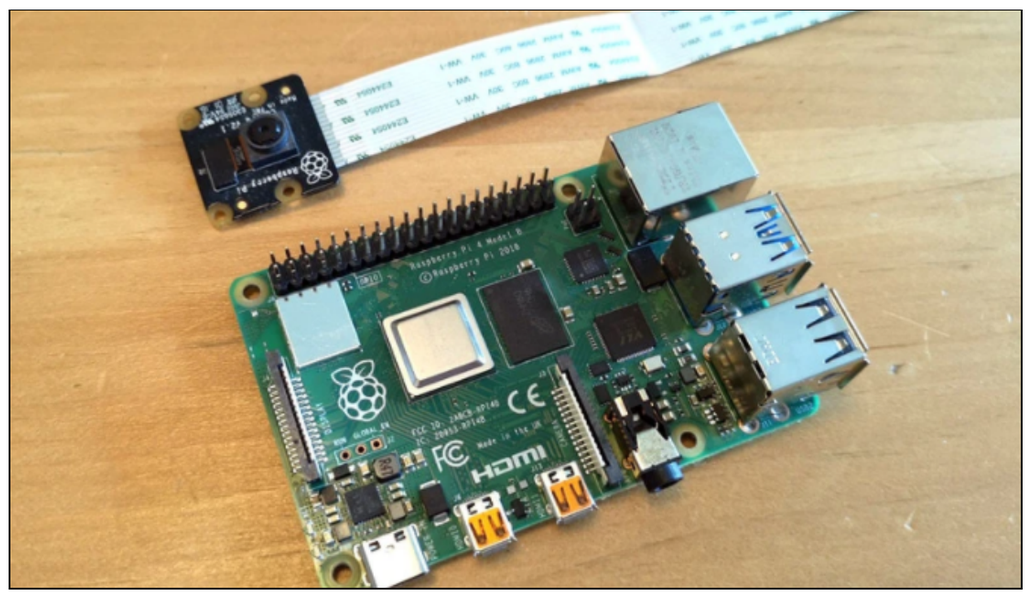
! 

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.

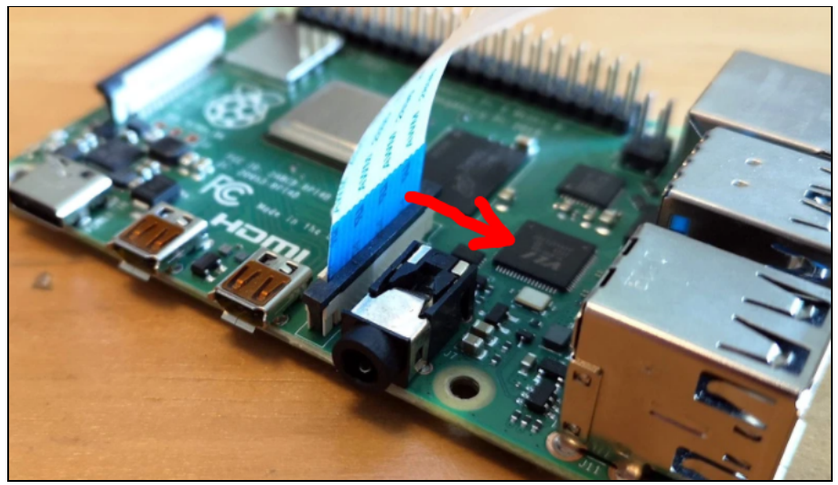
Before you take your Camera Module out of the box, be aware that it can be damaged by static electricity. Make sure you have discharged yourself by touching an earthed object (e.g. a radiator, PC Chassis or similar). Please make sure your Pi is off and unplugged from the power supply.

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.

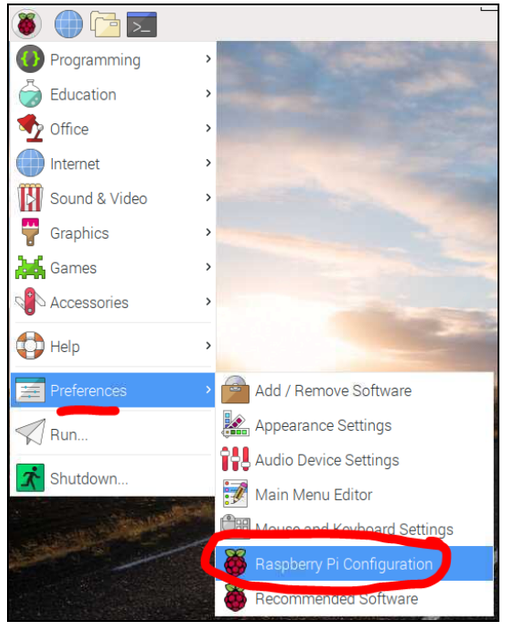


Install the Raspberry Pi Camera module by inserting the cable into the Raspberry Pi camera port. The cable slots into the connector situated between the USB and micro-HDMI ports, with the silver connectors facing the micro-HDMI ports.

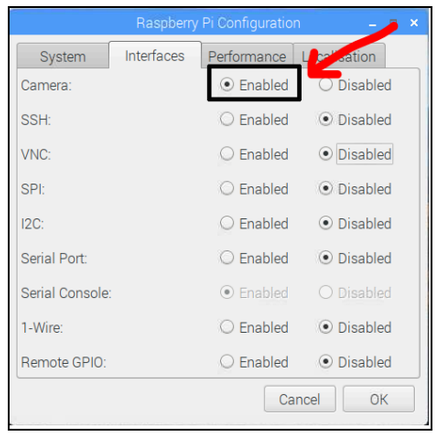
If in doubt, just make sure the blue part of the cable is facing the USB ports on the Raspberry Pi:



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



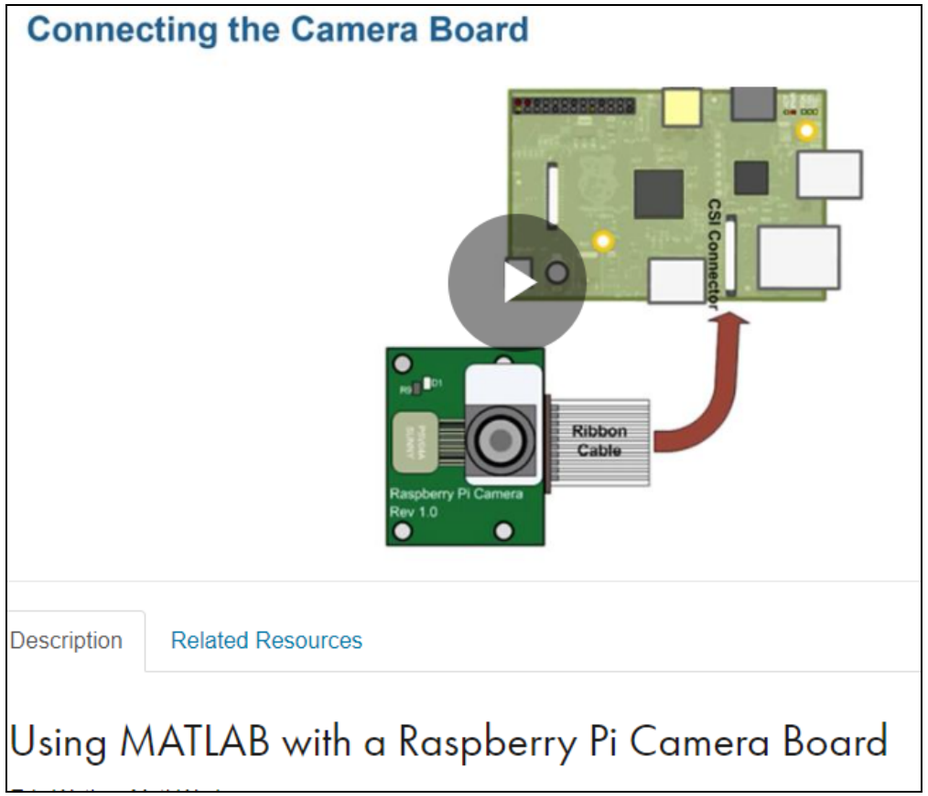
Select the **Interfaces** tab and ensure that the camera is **Enabled**:



Reboot your Raspberry Pi.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

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 (hopefully) 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
```

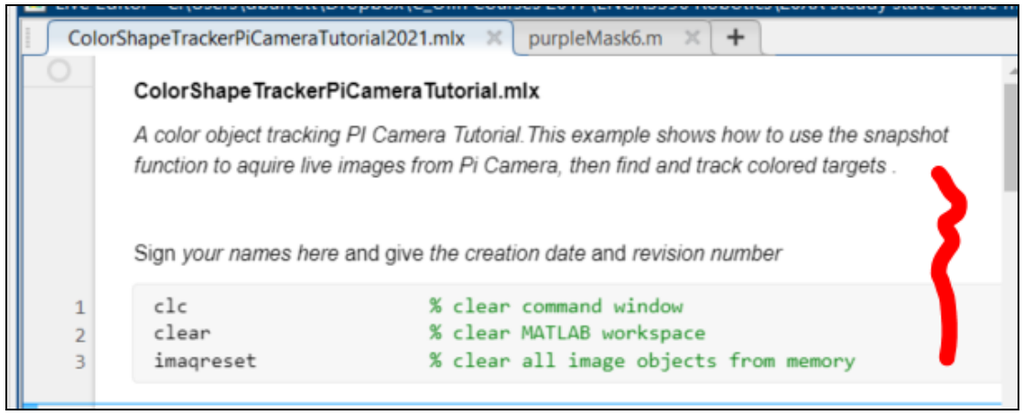
There is a lot to learn to begin using this powerful set of image processing tools well. For now we will have you focus on some simple ones and leave plenty of room for expansion if you get interested in the Computer Vision space.

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.

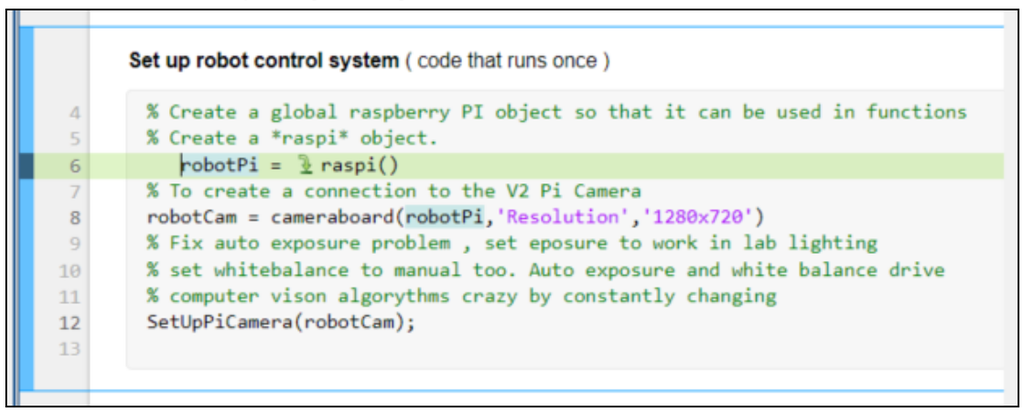
Next, let's write your first Pi image processing program.

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

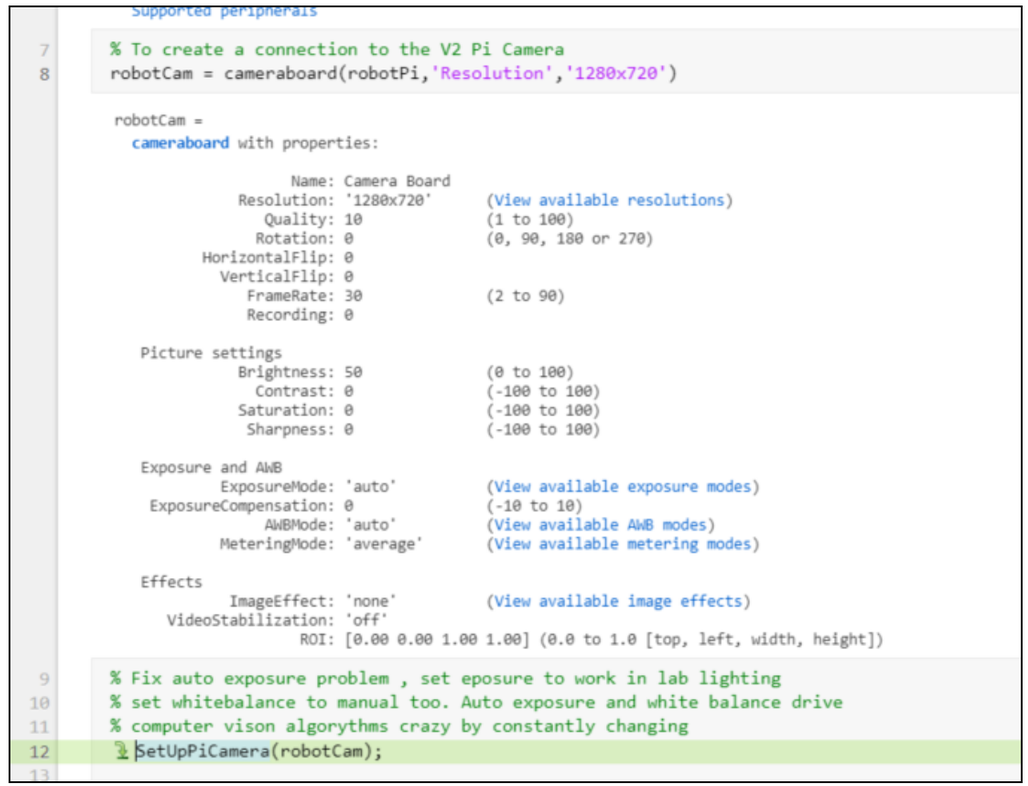
Please start with your Robot Control Template, save a copy as **ColorShapeTrackerPiCameraTutorial** and then enter the code below, section by section and we will walk you through collecting and inspecting your first webcam images.Please modify as shown:



Then create a Raspberry Pi object in code that runs once



The next lines of code connect your Pi Cam:



Before moving on, please click on the **View available** options for your Pi camera to get an idea of the **Resolutions**, **Exposure modes**, **Automatic White Balancing** modes and **Metering** modes as well as **Imaging Effects** available to you. The Pi cam has its own quite powerful internal processor and can do all sorts of useful things to your captured images, even before your Pi get a chance to process them. A bit later on you can go back in and play with these to get a better whale tracker out of the camera.

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 your initial system. You will do this in the function **SetUpPiCamera.** Please enter code as shown:

Sense Functions (store all Sense related local functions here)

```
73  function []= SetUpPiCamera(robotCam)
74      % creates and configures an Pi V2 Camera to be a simple robot
75      % vision system. It requires a standard Pi V2 cmaera attached to
76      % your Raspberry piand takes the picam object name as sole input
77      % You need to set your cameras unique parameters to optimize picture
78      % D. Barrett 2021 Rev A
79
80      % Fix  exposure set it to respond quickly, set auto whitebalance to
81      % fluorencent and then set brightness to get a good high contrast image
82      robotCam.ExposureMode = 'sports';
83      robotCam.AWBMode = 'fluorescent';
84      robotCam.Brightness = 30;
85  end
```

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  % 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;
```

Because the original snapshot function has a bug in it that stores 5 frames before updating it output to the Pi, we are going to use the modified one below:

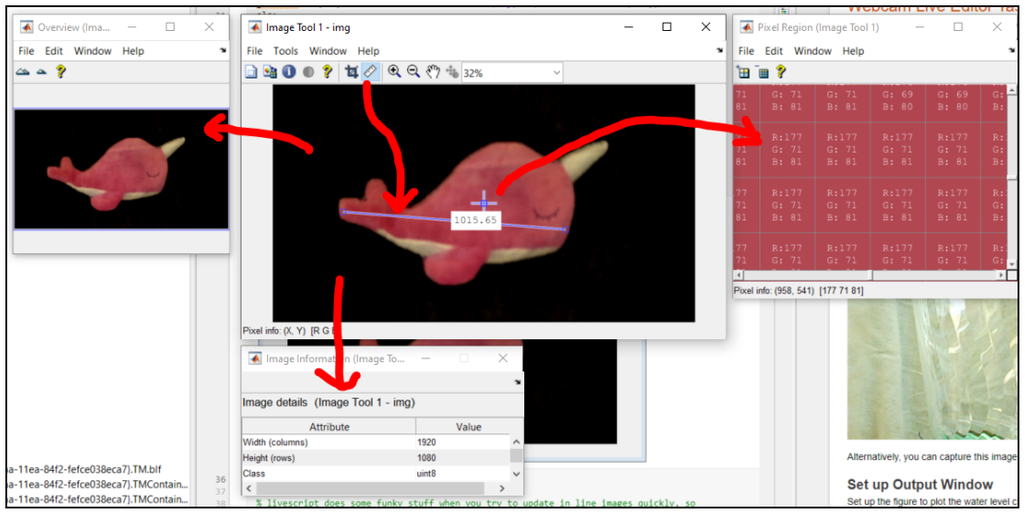
```
87  function imgOut = snapshotCustom(w)
88      % snapshotCustom: clears the buffer and get latest frame
89      % INPUT:
90      % w - webcam obj
91      %
92      % OUTPUT:
93      % imgOut - image data
94      % function provided by Matlab to solve 5 images in buffer problem.
95      % Feb 17 2021
96
97      % Buffer size
98      i = 5;
99      while(i>0)
100        imgOut = snapshot(w);
101         i = i -1;
102     end
103 end
```

So that we can get one new frame with each snapshot call.

This code takes an original image, asks if you've re-tuned your color filter today (more on this in a bit, for now just say **g enter** for go) and then dumps the image into the stand alone **imtool.**

We will set up that filter in a few sections, this request for operator input is there to remind you to set up and tune your color filter each time you come back to this program. Color filter based object detection schemes tend to be quite sensitive to lighting and color balance.

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 look at the size and content of the image.



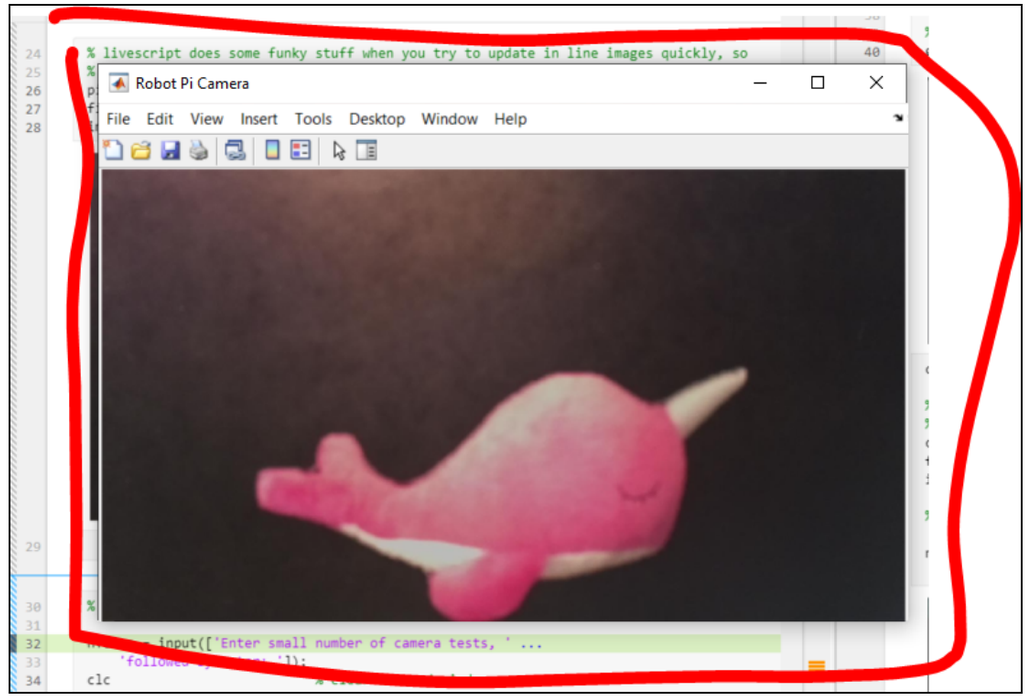
Play around with the tools until you are familiar with them then type in **G** (for go) followed by the enter key.

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

```
% explore image with tool
gCamTest= input('experiment with tool, type G, then hit ENTER ','s');
clc;

% 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
camWindow = figure('name','Robot Pi Camera','NumberTitle','off','Visible','on');
figure(camWindow)                          % 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:

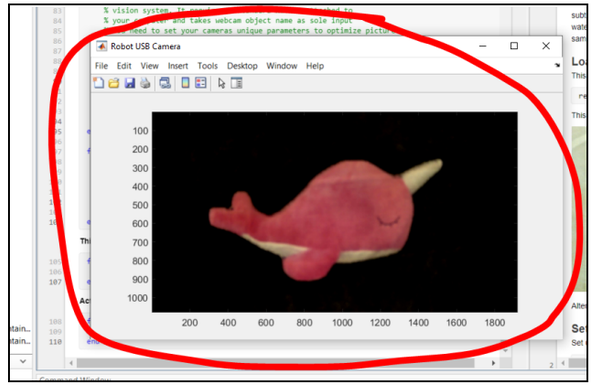
```
Run robot control loop ( code that runs over and over )
controlFlag = 1;
% create a loop control
while (controlFlag < nTests+1)   % loop till ntests data captured

    % 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
```

In this example code, the loop is pretty simple.the THINK and ACT functions are empty placeholders.

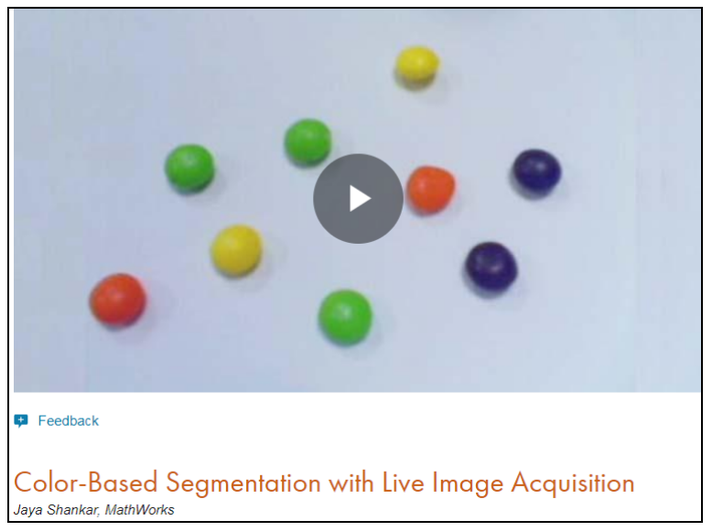
The SENSE function pulls an image off the USB webcam and there's a little logic to pause the loop to let you move targets between images. Taking a look at the SENSE function:

```
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 = snapshotCustom(robotCam);

   figure(camWindow)                          % go to camWindow for imshow
   imshow(robotImage)
   camtest= input('check out target image, type G, then hit ENTER ','s');
   clc;
```

It just takes a snapshot of what is in front of the Pi camera and returns that as a **robotImage**. It then will display the image you just captured in the Pi camera window. It is always helpful to display the image each step of the way as you do image processing. It will give you much insight as to what is happening so you can debug easier. You can then comment these image displays out after you get it all working the way you want it to.



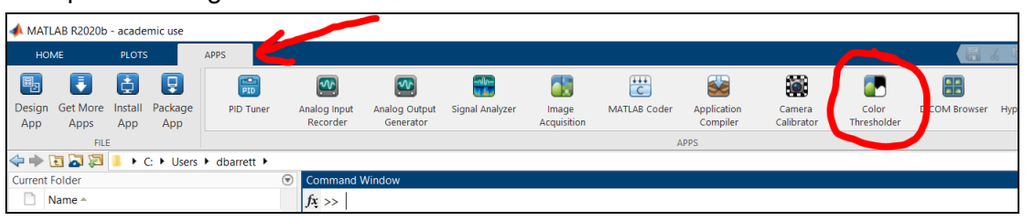
After collecting images, it would be good to be able to find and identify multiple different colored targets. Please view this short video tutorial on how to do so: https://www.mathworks.com/videos/color-based-segmentation-with-live-image-acquisition-68771.html



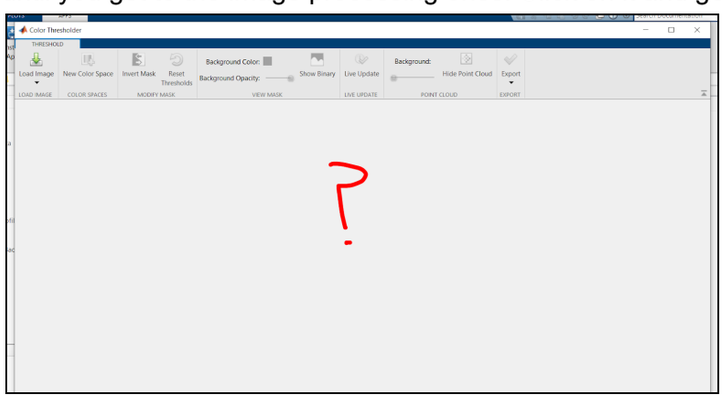
And then, drawing heavily from the tools shown in the above tutorial and with the guide below, let's write a MATLAB function that lets you find and identify the multiple colored targets on the test station cart.

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.

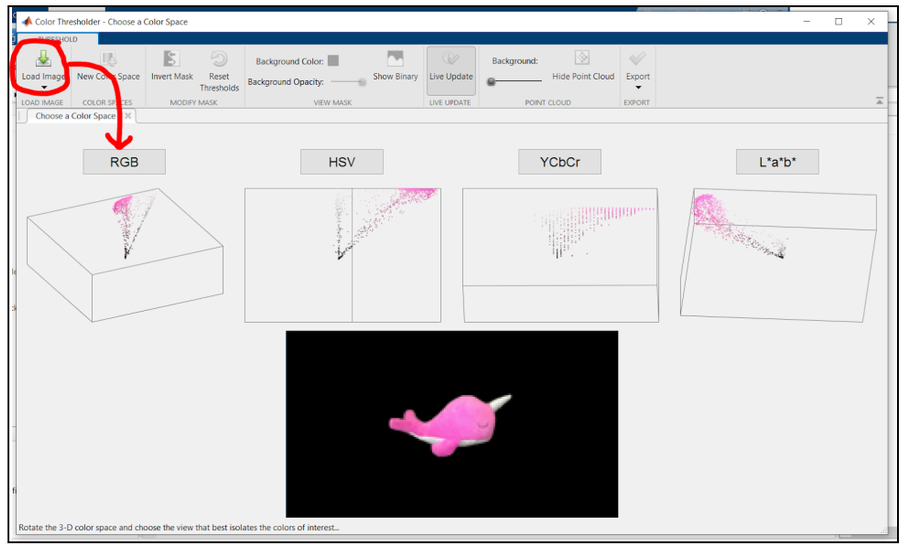
You might want to use this tool multiple times to build several color filters, one for each potential target. Go to **APPS tab** then **ColorThresholder**



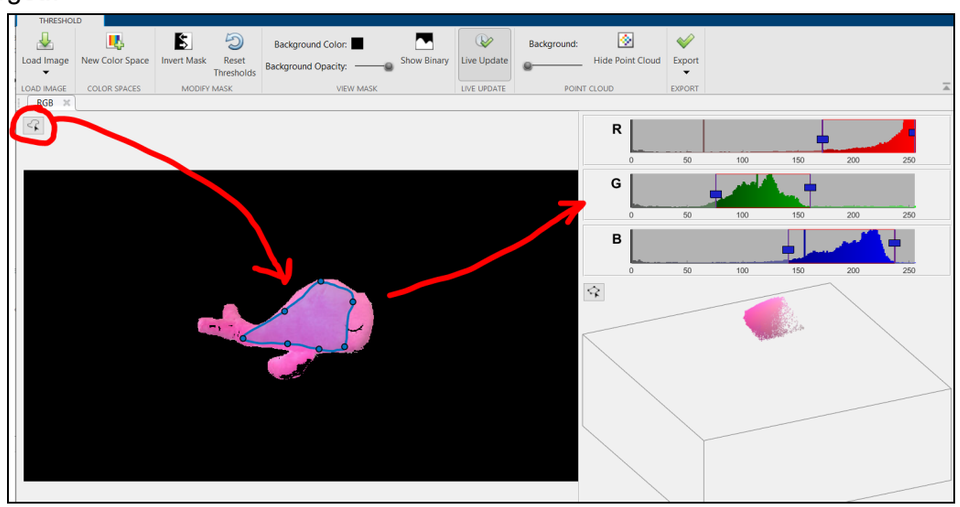
There are a lot of APPS in 2020b, you may need to scan down the full page of APPS until you get to the image processing ones. Click on it and get:



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

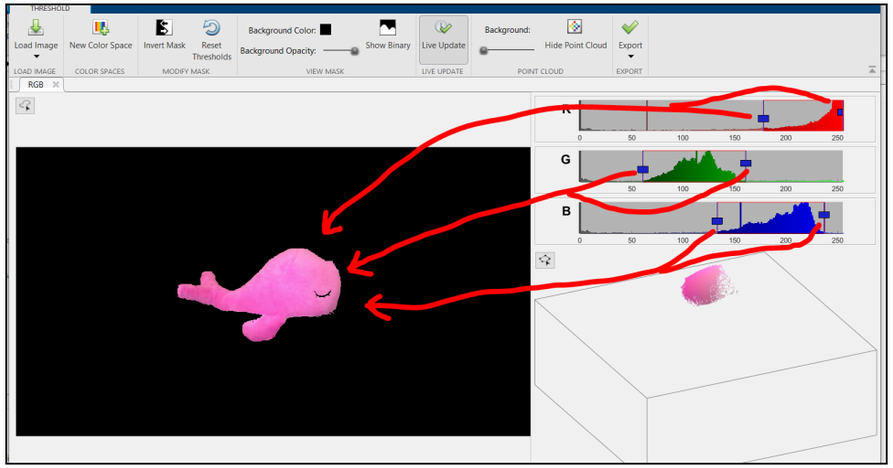


Select RGB color space (we will start out with RGB, but downstream you might find other color spaces let you better distinguish between similar colors) and then use the **lasso tool** in the top left of the image workspace to lasso the colored area of your target.:

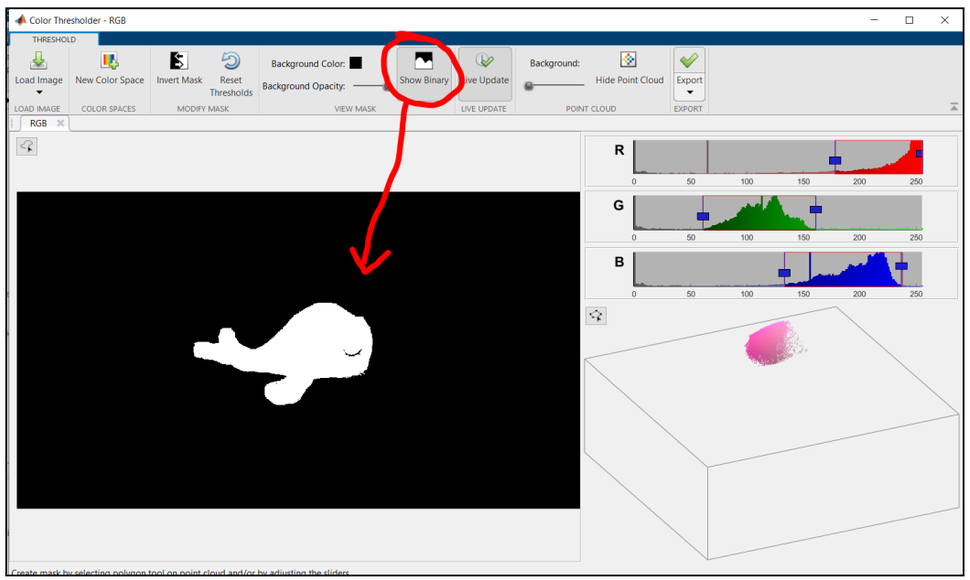


ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

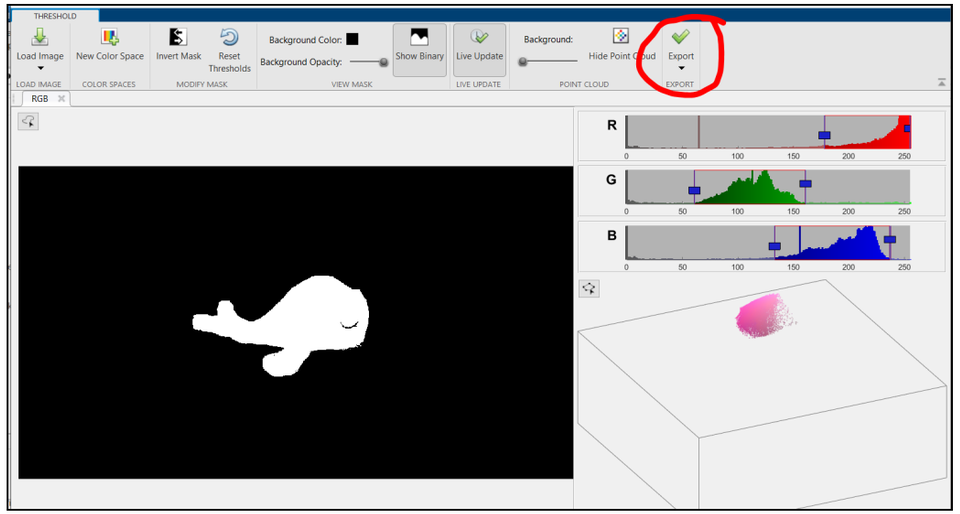
Use the two bar sliders on the R G B tracks to fine tune the filter to capture the best segmentation of the image that you can.



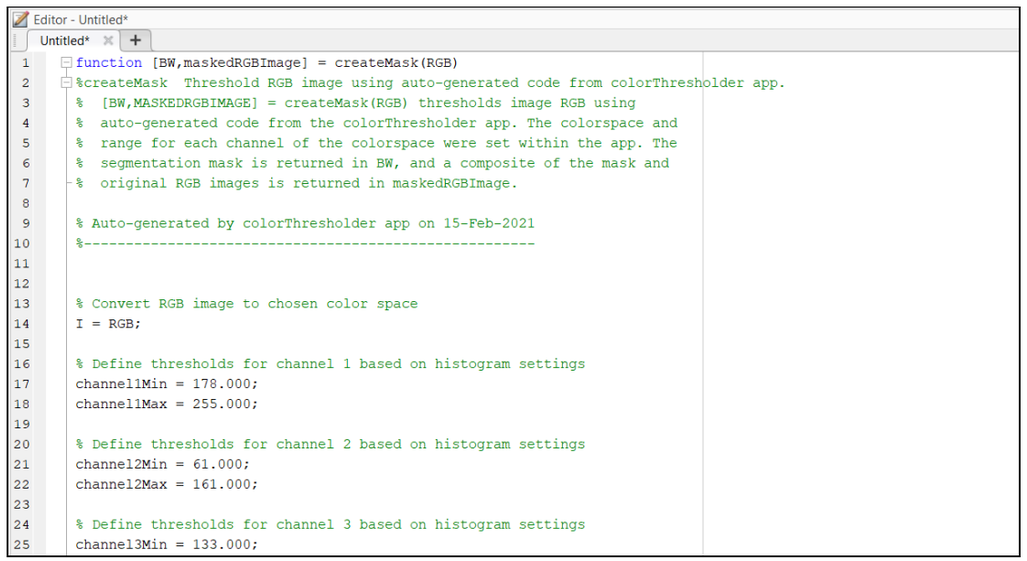
You can click on **Show Binary** button to see the full binary mask that you just created with this App tool:



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



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



ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

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

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

% Auto-generated by colorThresholder app on 15-Feb-2021
%------------------------------------------------------


% Convert RGB image to chosen color space
```

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

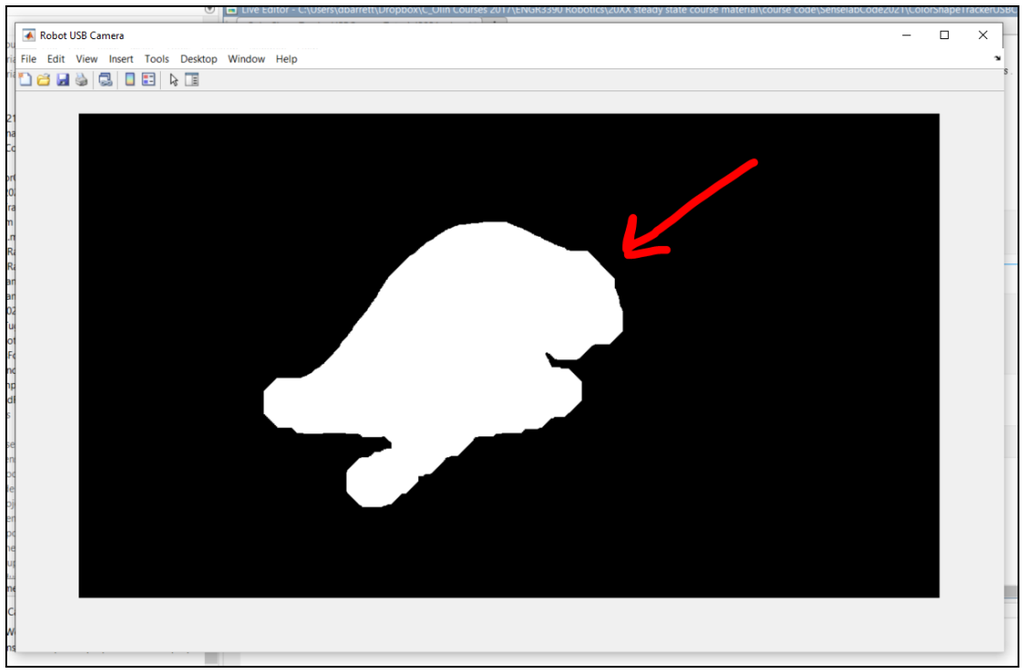
Diving right into the new SENSE function, Add purplemask function:

```
        figure(camWindow)                       % go to camWindow for imshow
        imshow(robotImage)
        camtest= input('check out target image, type G, then hit ENTER ','s');
        clc;

        %% 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]= purpleMask6(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.

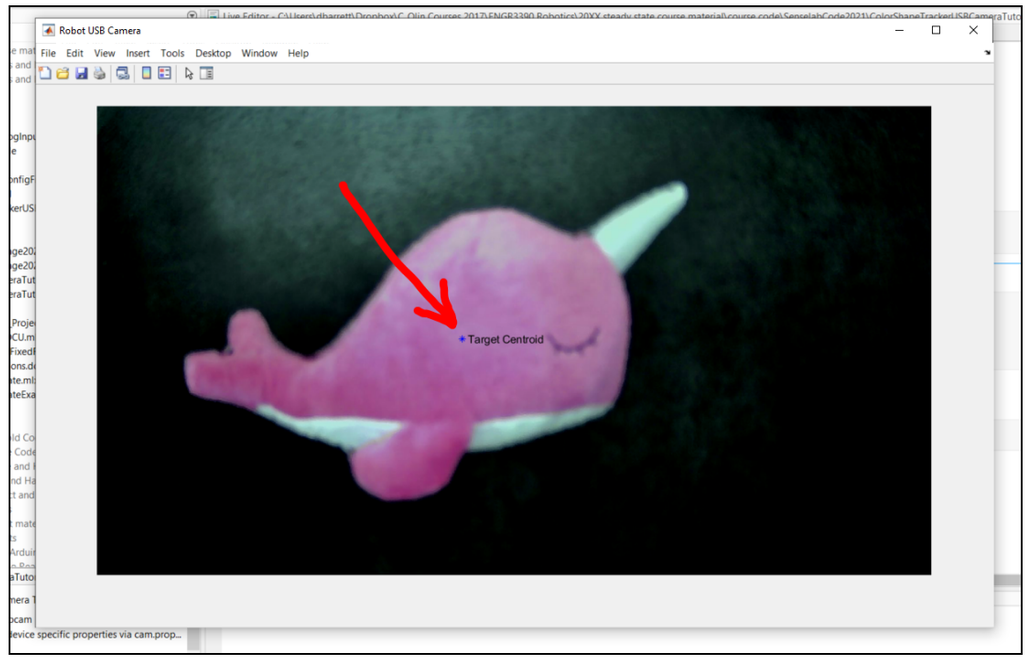
You will take that BW image and remove much of the noise from it:

```
        camtest= input('check out masked image, type G, then hit ENTER ','s');
        clc;

        %% 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;

        %% Calculate the centroid of the white part of masked area (target)
```



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
% 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 your whale finder work?

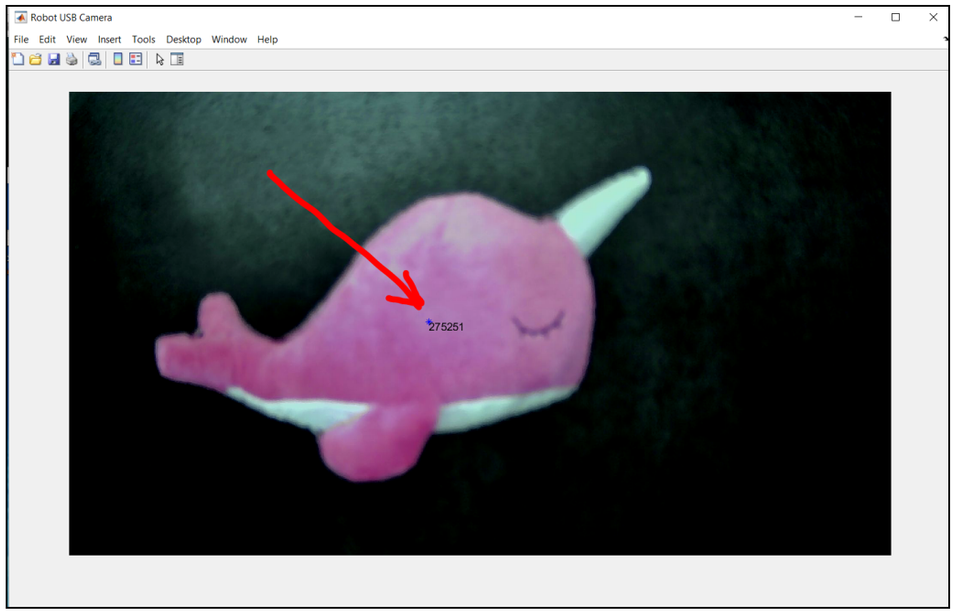
The SENSE function returns the coordinates of the target back up to the main program. In a full robot control system (like the THINK lab), you could use the x-component of this centroid to steer the robot Tug toward the NarWhal.

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

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

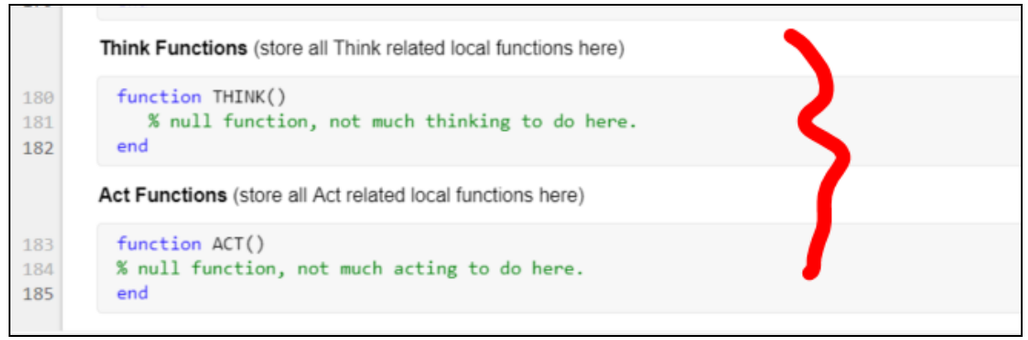
```
157     %%  Calculate the area of the target
158     targetArea = regionprops(cleanImage,'area');
159     if isempty(targetArea)                      % if no target found
160           area = 0;                  % place centroids at 0 0
161     else
162     area=targetArea.Area;
163     end
164     % 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 close it gets bigger. You could use this information to avoid running into it (driving into NarWhals is just bad and probably illegal) !

The Think and Act functions are just placeholders:



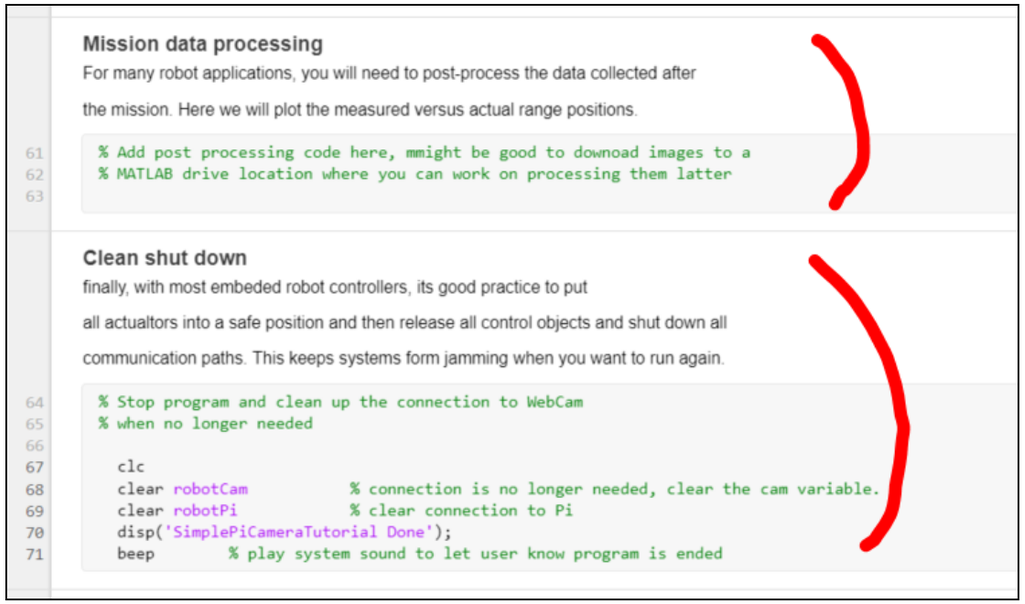
Think Functions (store all Think related local functions here)

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

Act Functions (store all Act related local functions here)

```
183 function ACT()
184 % null function, not much acting to do here.
185 end
```

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

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 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 to the targets as well as identify which target it is.
3. Stretch goal. Write demo code to track and give 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 bearings 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 bearings 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.

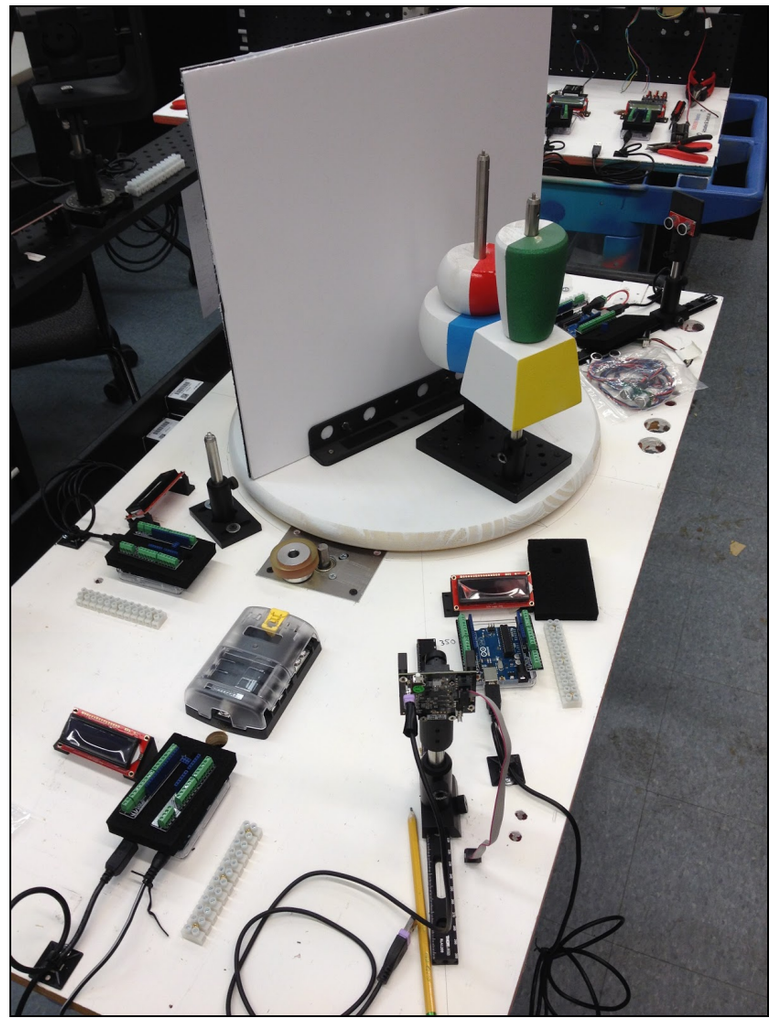


Figure: PixyCam test station

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.

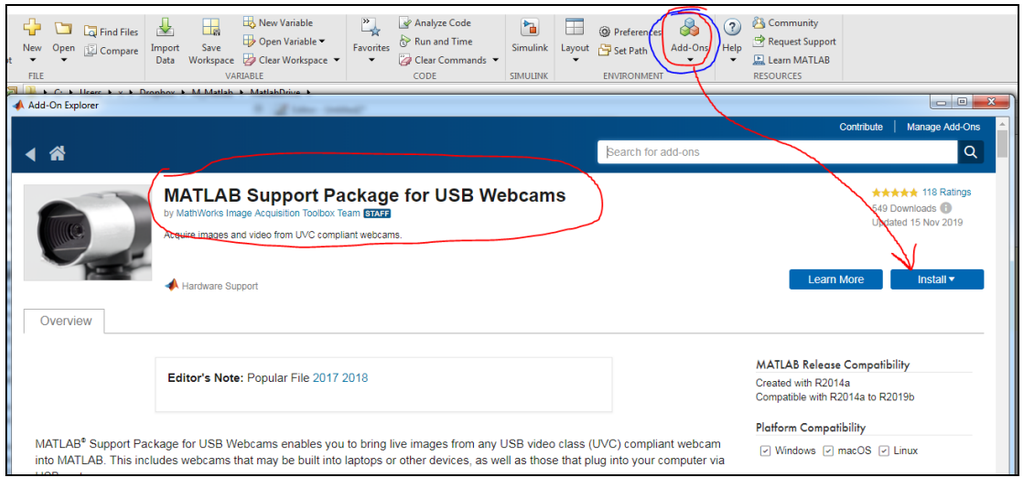
MATLAB has a very deep and quite wide collection of image processing functions and can pretty much do any robot sensing application you might be interested in.

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.

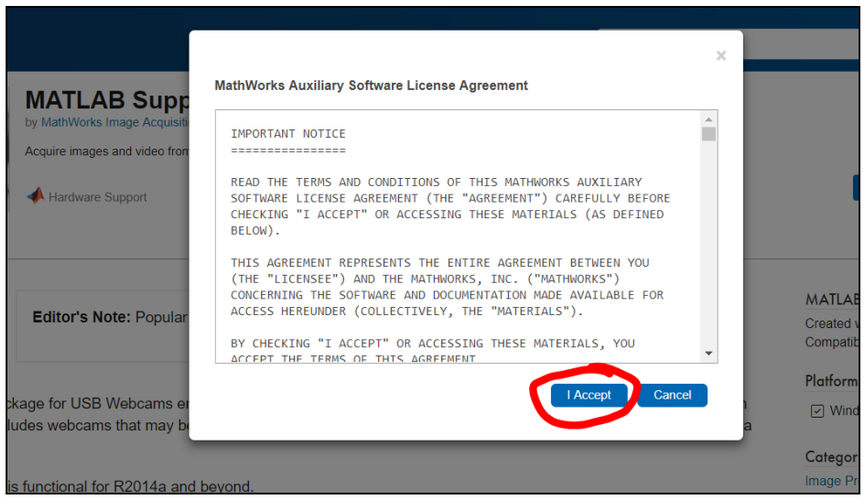
First, Plug the Microsoft webcam in and then wait for it to load the USB driver (may take a few minutes) and follow the on screen instructions to get it hooked up and added to your WIndows operating system

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

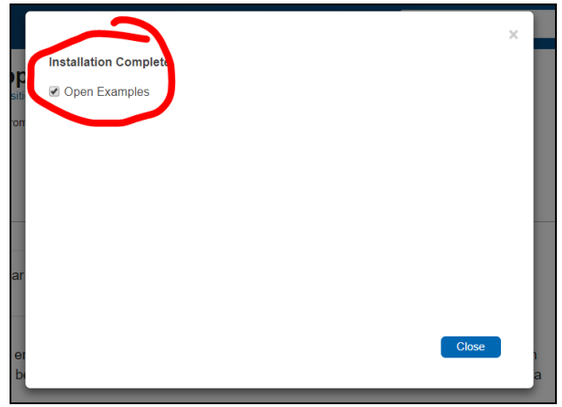
Next go up to the top MATLAB **Home** menu, click on **Add-Ons, Get Add-ons,** search for and then find the **MATLAB Support Package for USB Webcams:**



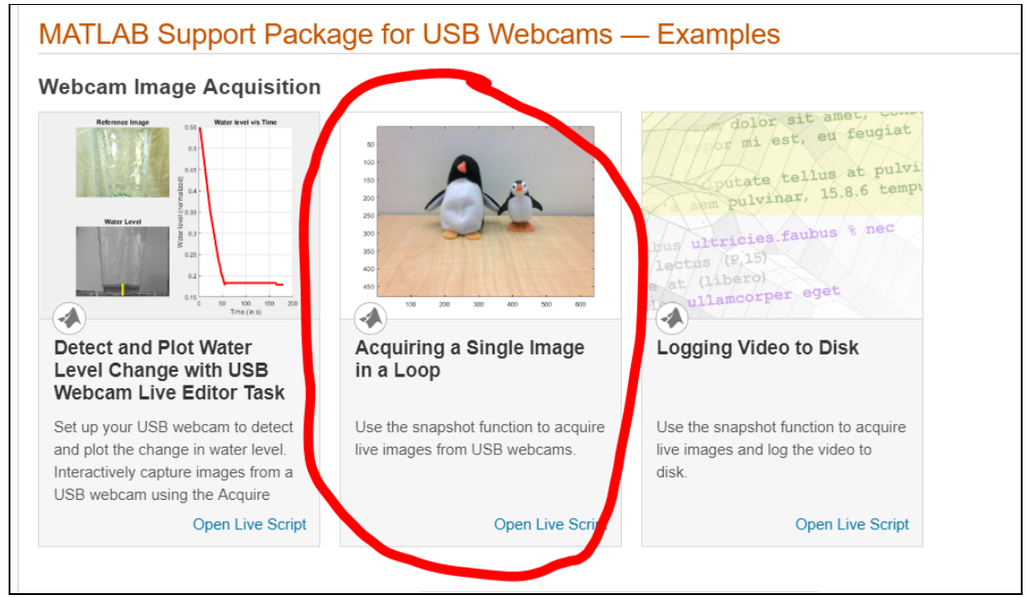
Click on the Blue Install button. Accept Auxiliary Software License agreement.



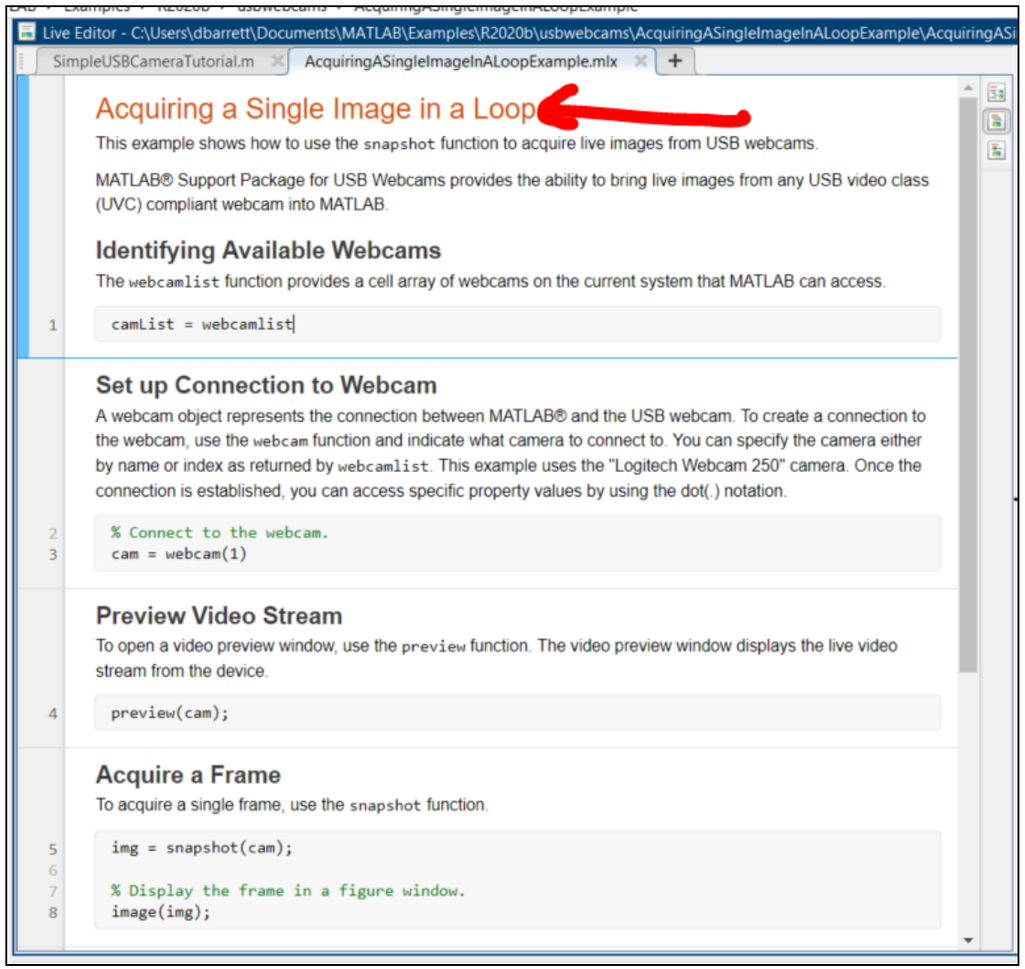
Wait a bit for it to download and install the USB camera package. Open up the Examples:



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

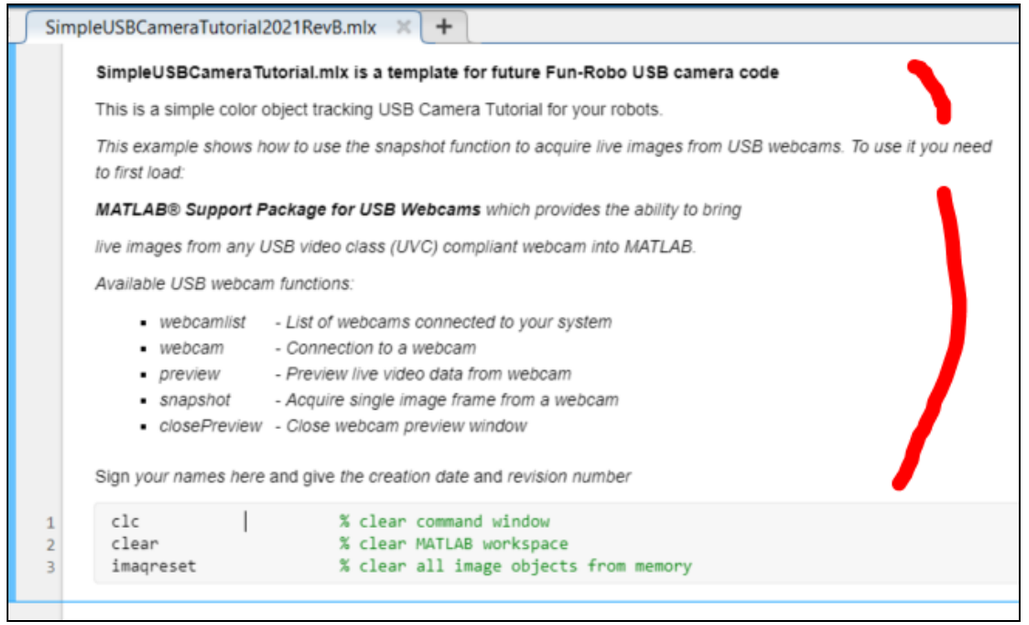


ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

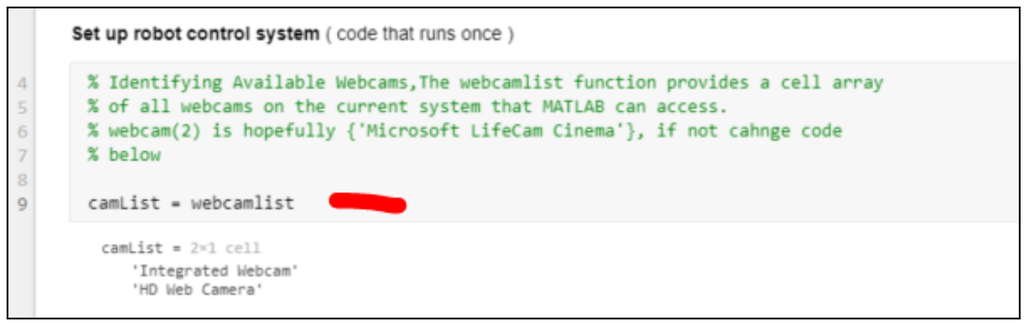


Recommend running in debugger mode, line by line, to get a good grasp of what it is doing and how all of the functions work. This short script will connect you to your (or the labcart's) webcamera, take a few pictures and store them.

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;
```

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

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:

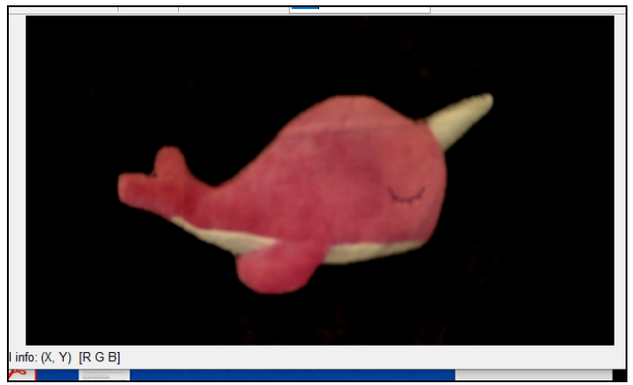
Sense Functions (store all Sense related local functions here)

```
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:

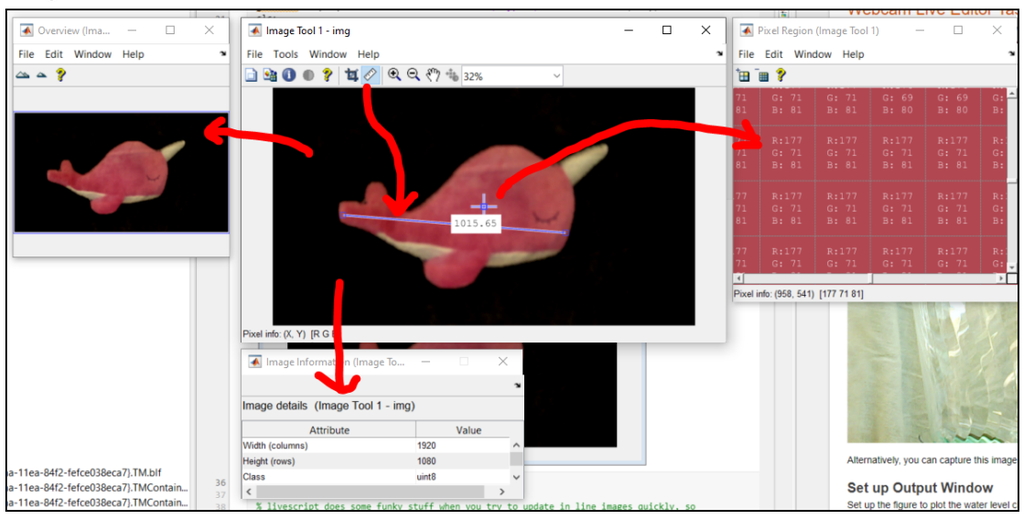
Sense Functions (store all Sense related local functions here)

```
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');
```

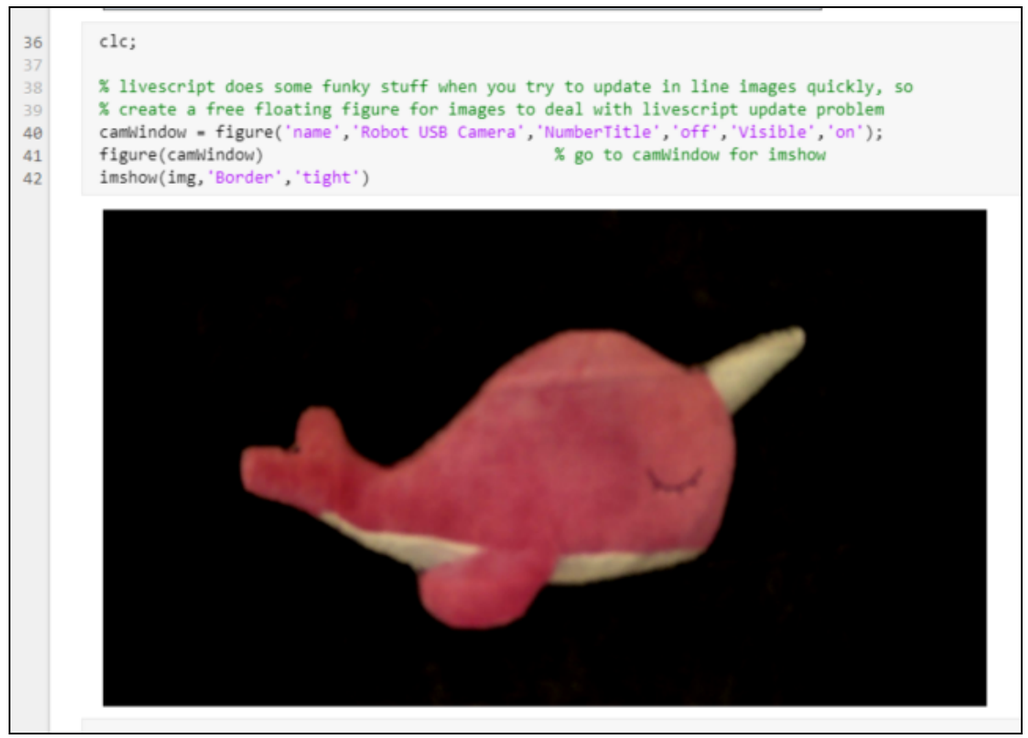
The **imtool** lets you probe the actual RGB pixel values in a region, measure objects on the screen to tell how many pixels they span and the size and content of the image.



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

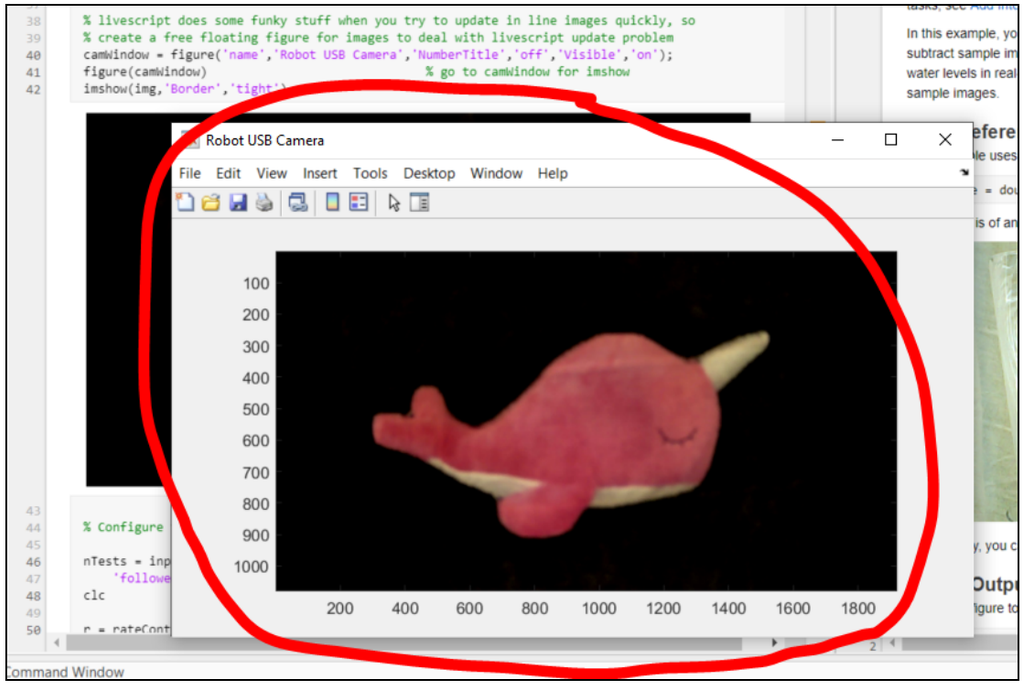
The next set of code creates an external figure to drop images in:

```
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')
```

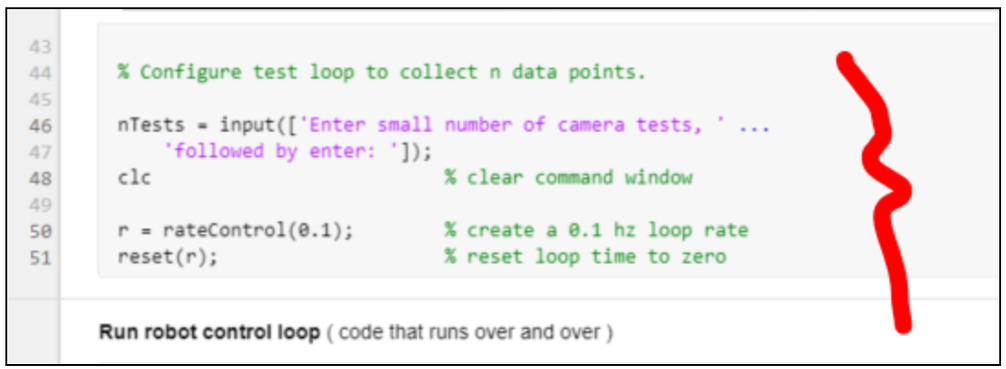


Because of some program oddness in the Java engine behind Live Scripts, it sometimes does weird things with in-line figures in the Matlab code window. To overcome this and to also give you a semi-permanent record of the image after the script completes, this code will create an external stand alone figure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

The figure looks like this:



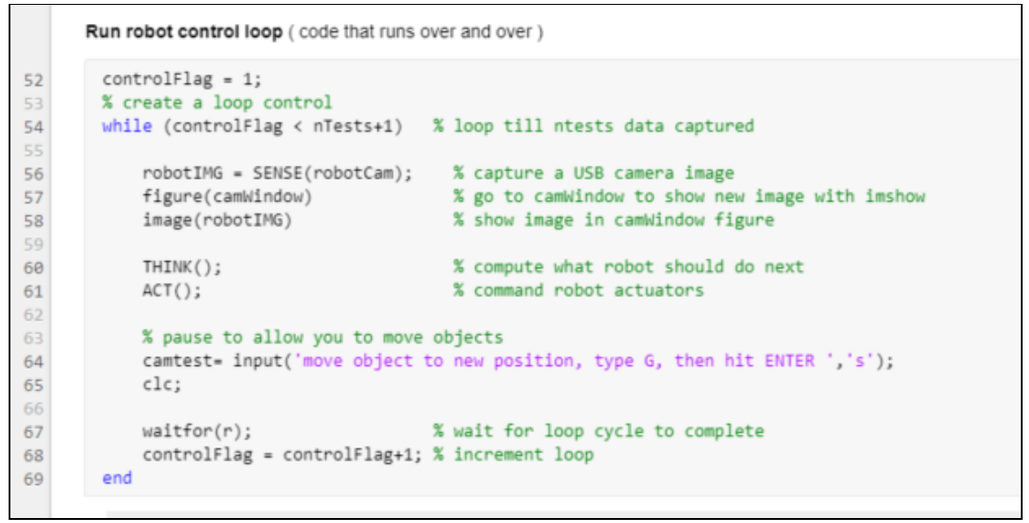
Next there is a little piece of code that sets up the number of experimental test to do:



```
% 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)

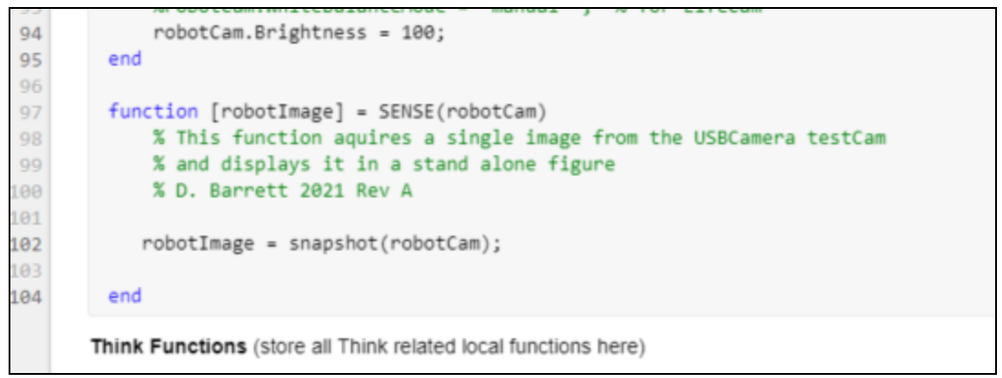
Followed by the robot control loop:



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:



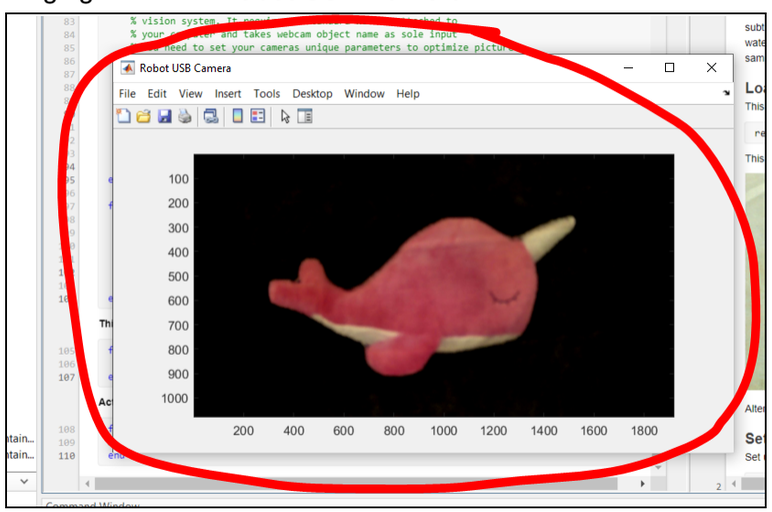
```
    robotCam.Brightness = 100;
end

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) 2021

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:

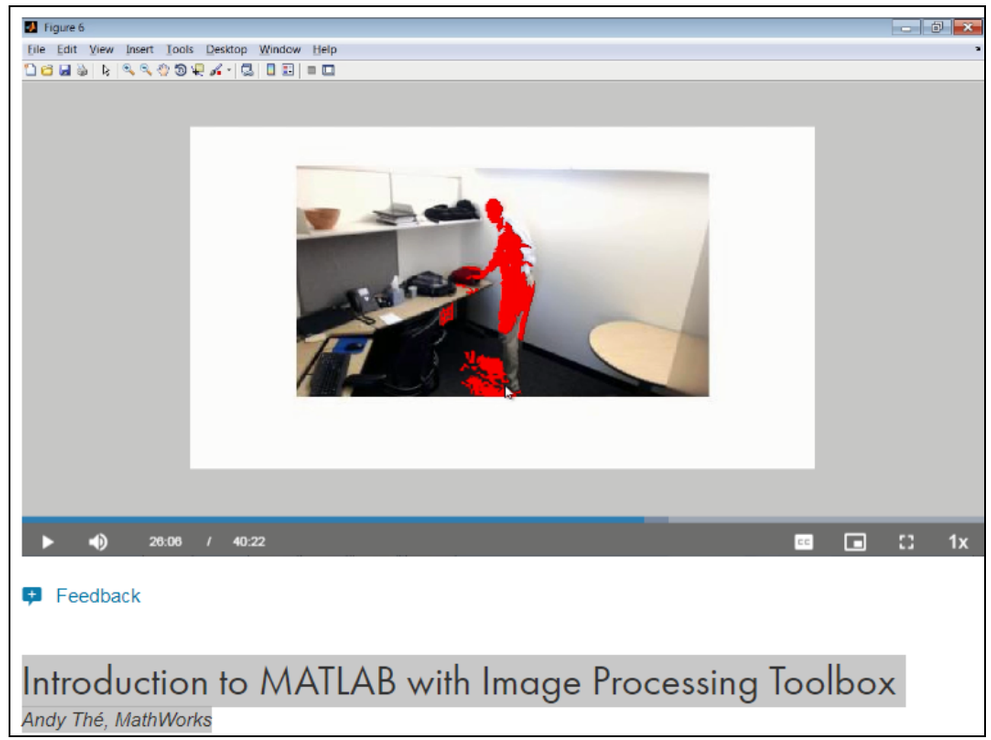
Introduction to MATLAB with Image Processing Toolbox *Andy Thé, MathWorks*
https://www.mathworks.com/videos/introduction-to-matlab-with-image-processing-toolbox-90409.html

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.



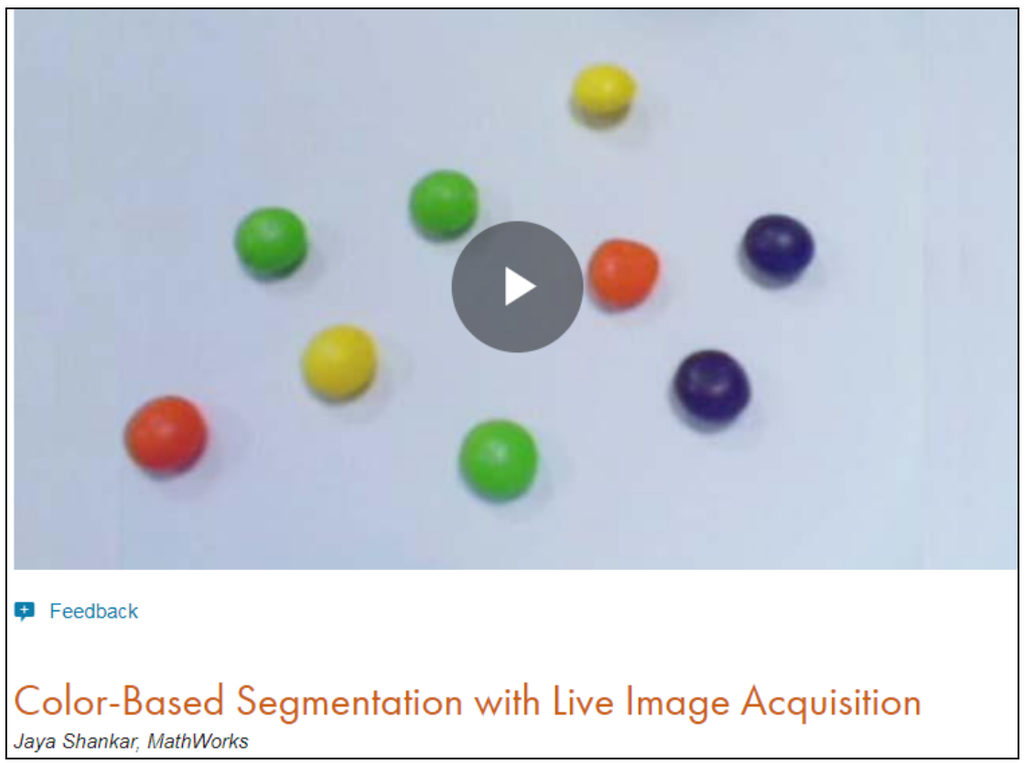
This tutorial is good at finding what has changed in a video workspace and that is pretty much exactly what you will need to do for the first part of the USB lab. Please:

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) 2021

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:

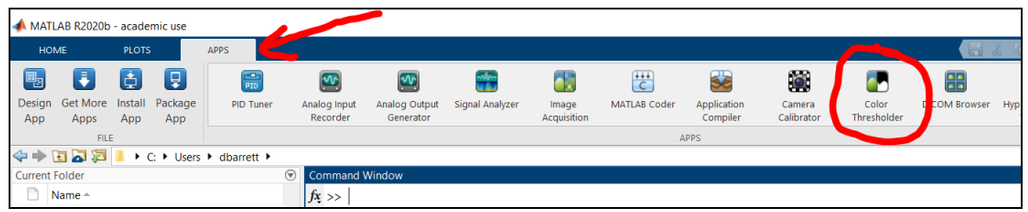
https://www.mathworks.com/videos/color-based-segmentation-with-live-image-acquisition-68771.html



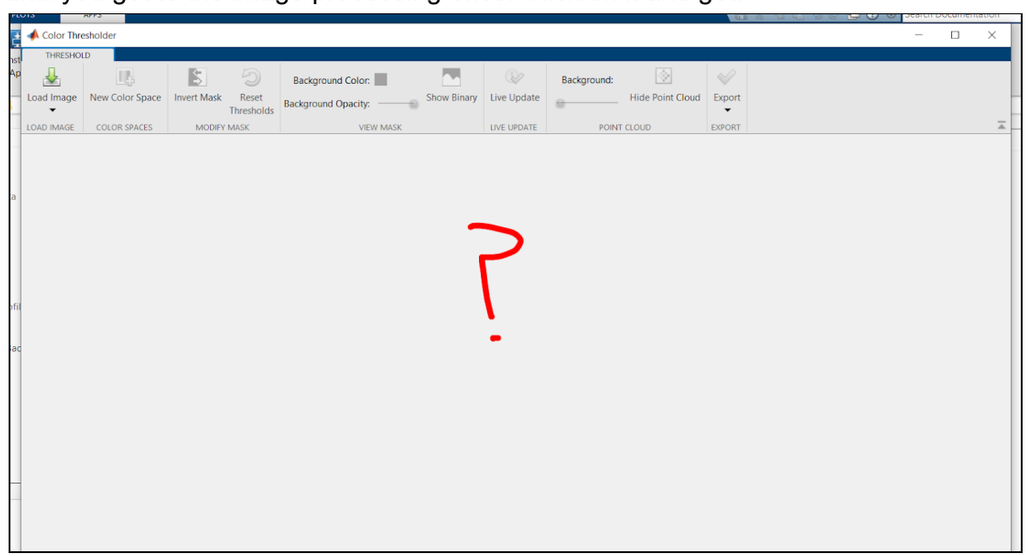
And then, drawing heavily from the tools shown in the above tutorial and with the guide below, let's write a MATLAB script that lets you find and identify the multiple colored targets on the test station cart.

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.

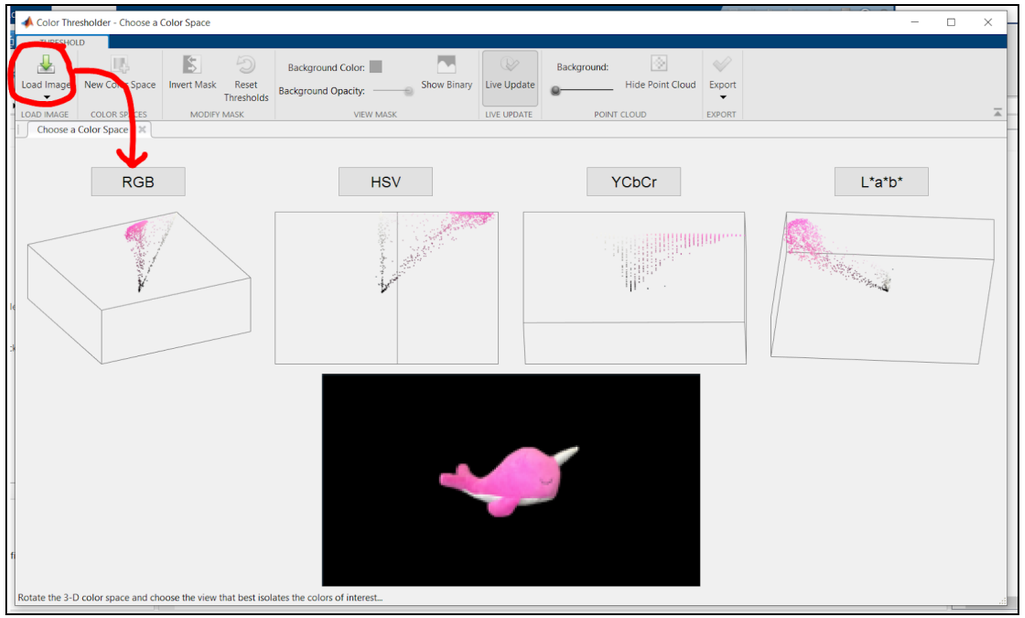
You might want to use this tool multiple times to build several color filters, one for each potential target. Go to **APPS tab** then **ColorThresholder**



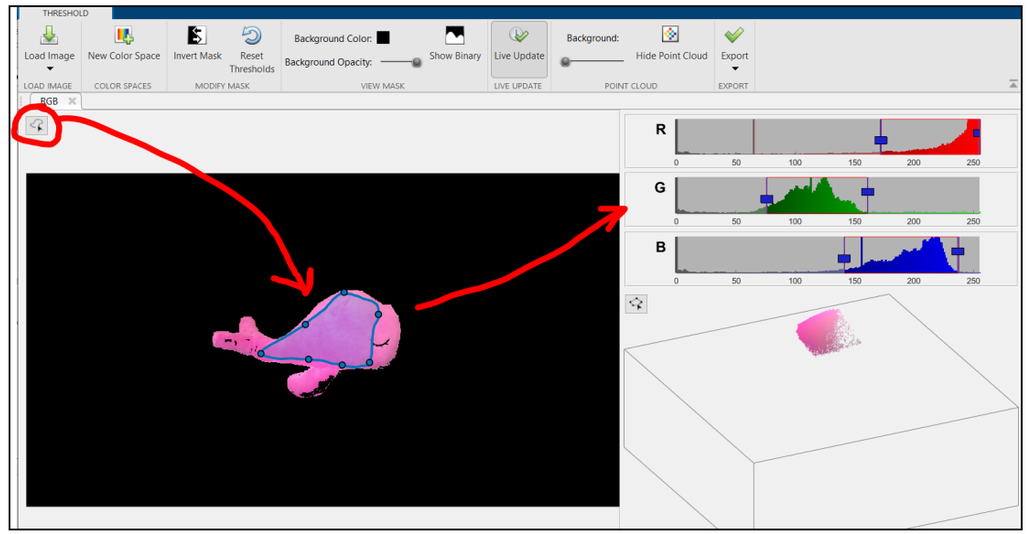
There are a lot of APPS in 2020b, you may need to scan down the full page of APPS until you get to the image processing ones. Click on it and get:



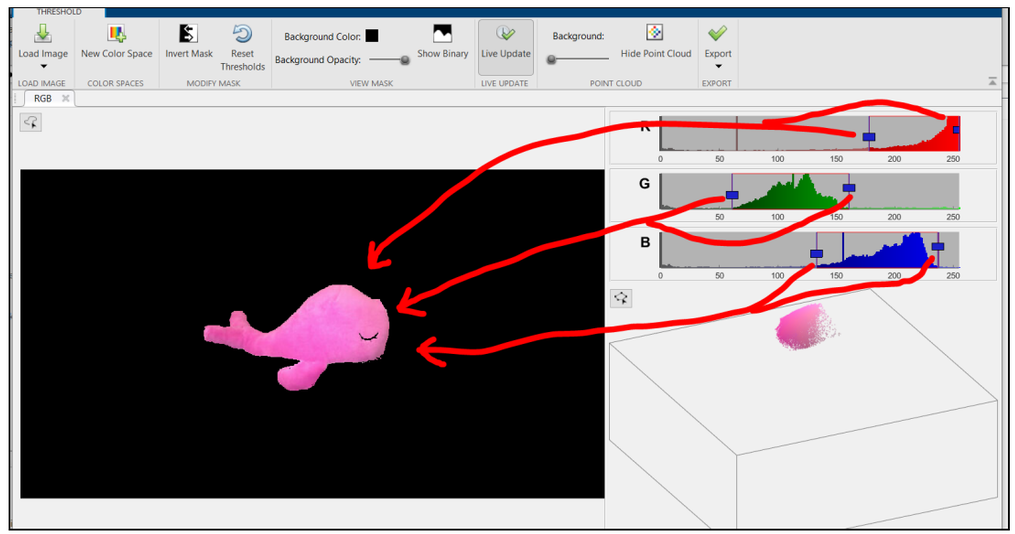
Click on **Load Image** and load one of your saved images from the previous section of this tutorial, it will appear in the center work space:



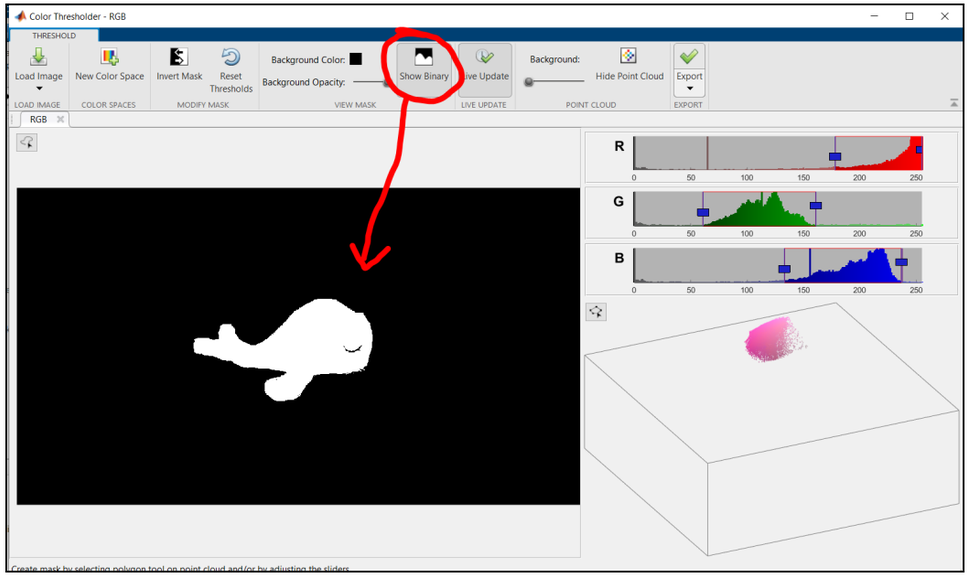
Select RGB color space (we will start out with RGB, but downstream you might find other color spaces let you better distinguish between similar colors) and then use the **lasso tool** in the top left of the image workspace to lasso the colored area of your target.:



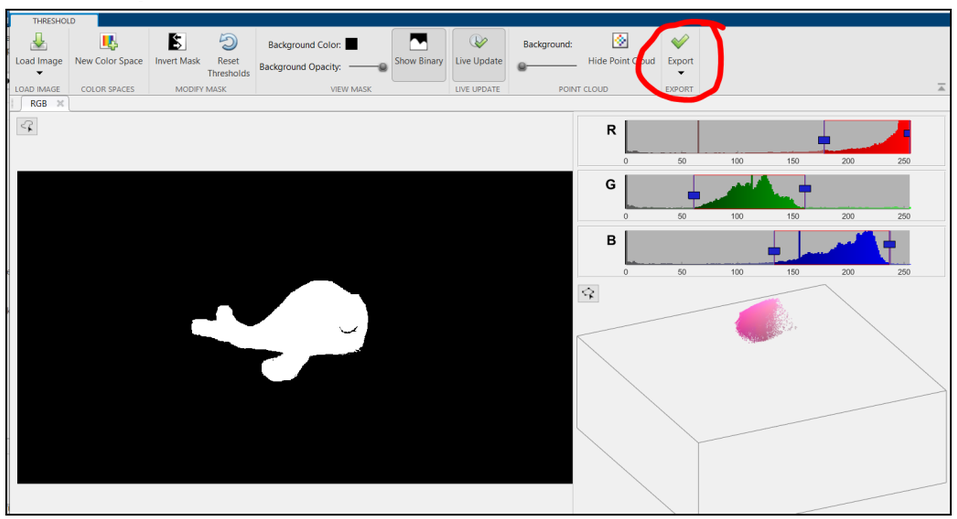
Use the two bar sliders on the R G B tracks to fine tune the filter to capture the best segmentation of the image that you can.



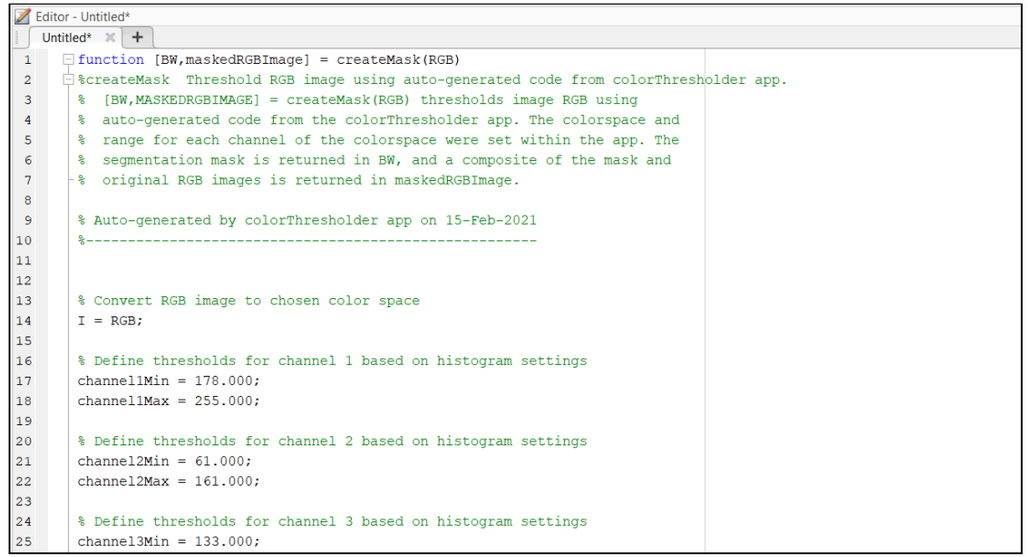
You can click on **Show Binary** button to see the full binary mask that you just created with this App tool:



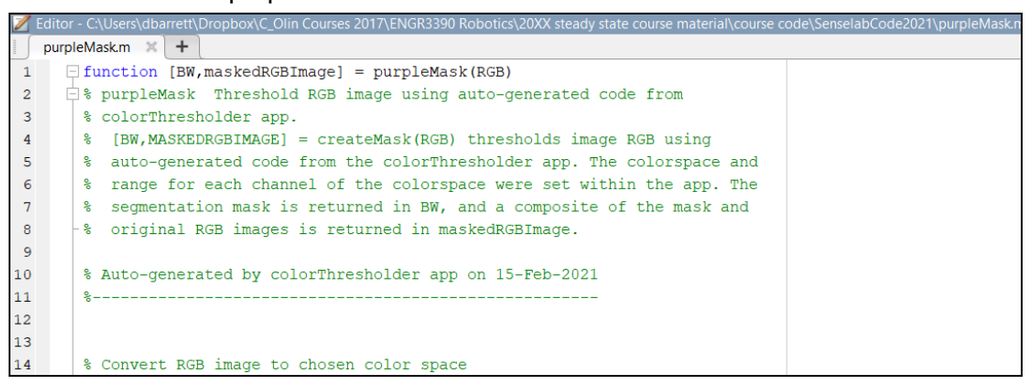
Click on **Export** choose **Function**



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



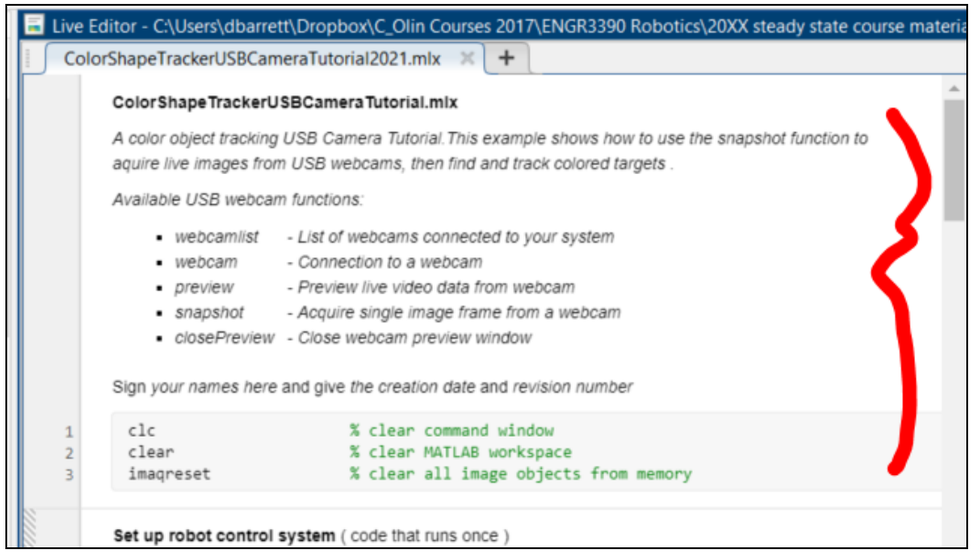
Edit with updated comments and save it into your working Matlab Drive directory with a useful name like purpleMask.m:



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

ENGR3390: Fundamentals of Robotics Tutorial (SENSE - Simple sensors) 2021

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



Most of the code stays the same:

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

% close Preview window
closePreview(robotCam);
```

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:

```
% Acquire a Frame
img = snapshot(robotCam);

% Go off and use colorMask APP to make filter based on this image
% just load image img from Matlab workspace
gCamTest= input('todays filter done?, type G, then hit ENTER ','s');
clc;

% Display the frame in the imaqtool window
% imaqtool launches an interactive GUI to allow you to explore, configure,
% and acquire data from your installed and supported image acquisition devices.
imtool(img);

% explore image with tool
gCamTest= input('experiment with tool, type G, then hit ENTER ','s');
clc;

% 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
camWindow = figure('name','Robot USB Camera','NumberTitle','off','Visible','on');
figure(camWindow)                              % go to camWindow for imshow
imshow(img,'Border','tight')

% 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
```

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) 2021

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)

```
57  controlFlag = 1;
58  % create a loop control
59  while (controlFlag < nTests+1)   % loop till ntests data captured
60
61      % SENSE finds centroid and area of target
62      [centroids, targetArea] = SENSE(robotCam, camWindow)
63      THINK();                     % compute what robot should do next
64      ACT();                       % command robot actuators
65
66      % pause to allow you to move objects
67      camtest= input('move object to new position, type G, then hit ENTER ','s');
68      clc;
69
70      waitfor(r);                  % wait for loop cycle to complete
71      controlFlag = controlFlag+1; % increment loop
72  end
```

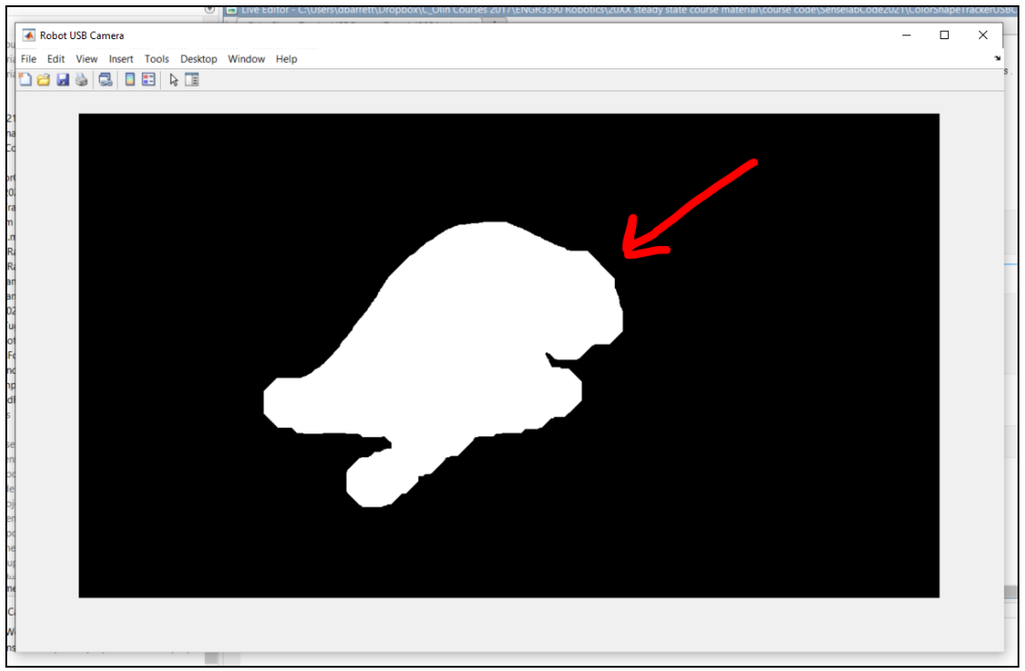
Diving right into the new SENSE function, please modify as we go:

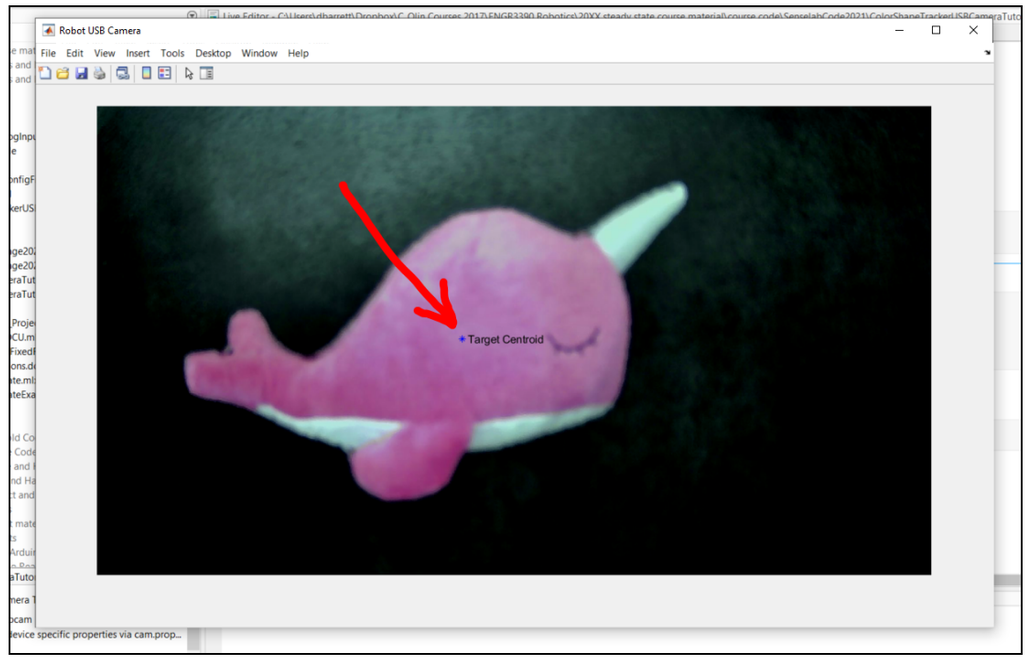
```
100 function [centroids, targetArea] = SENSE(robotCam, camWindow)
101    % This function aquires a single image from the USBCamera testCam
102    % finds and returns centroid of purple area in image
103    % D. Barrett 2021 Rev A
104
105    % capture image
106    robotImage = snapshot(robotCam);
107
108    %% Use Color Threshold App to create a colorMask function (purpleMask)
109    %  and place function in same directory as this script. Your functions
110    %  will have different names depending on color of target you want to
111    %  track, apply colorMask Function to captured image
112    %  this will create two new images
113    % [BW,maskedRGBImage] = purpleMask(RGB)
114    %  BW is binary mask of image
115    %  colorMaskeImg will be the color image inside the mask
116    [targetMask,purpleImg]=purpleMask2(robotImage);
117    figure(camWindow)                          % go to camWindow for imshow
118    imshow(targetMask)
119    camtest= input('check out masked image, type G, then hit ENTER ','s');
120    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:

```
120     clc;
121
122     %% Preprocess image to remove noise
123     se = strel('disk',50);               % structured element erosion function
124     cleanImage= imopen(targetMask, se);  % Morphologically open image
125     figure(camWindow)                                % go to camWindow for imshow
126     imshow(cleanImage)
127     camtest= input('check out cleaned image, type G, then hit ENTER ','s');
128     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;
```

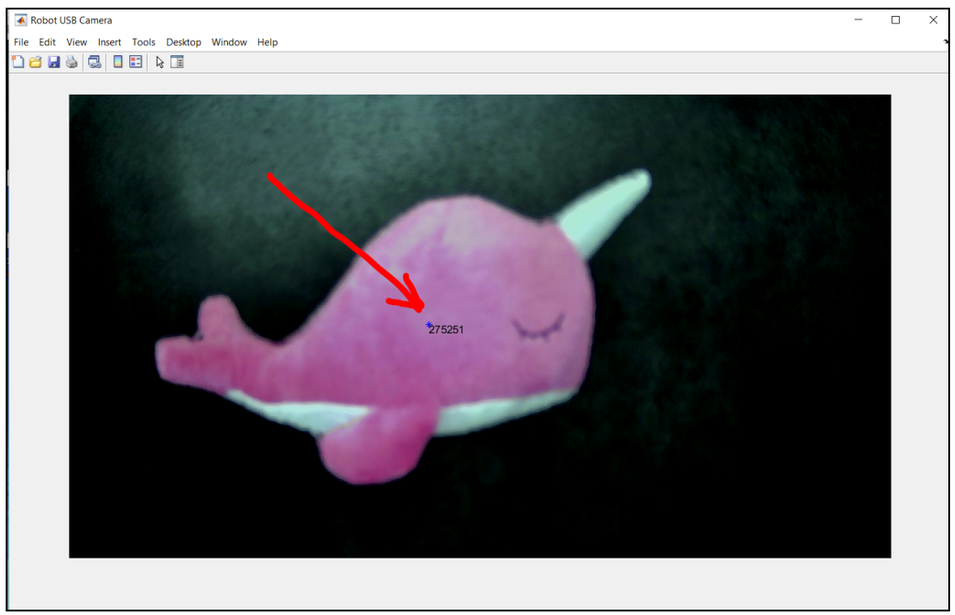
Having found the centroid, the code draws and labels its calculated location onto the original image. In image processing it is always good to do this, namely to draw your final results onto the original image as both a sanity check and as a way of understanding how the algorithms work under a variety 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:

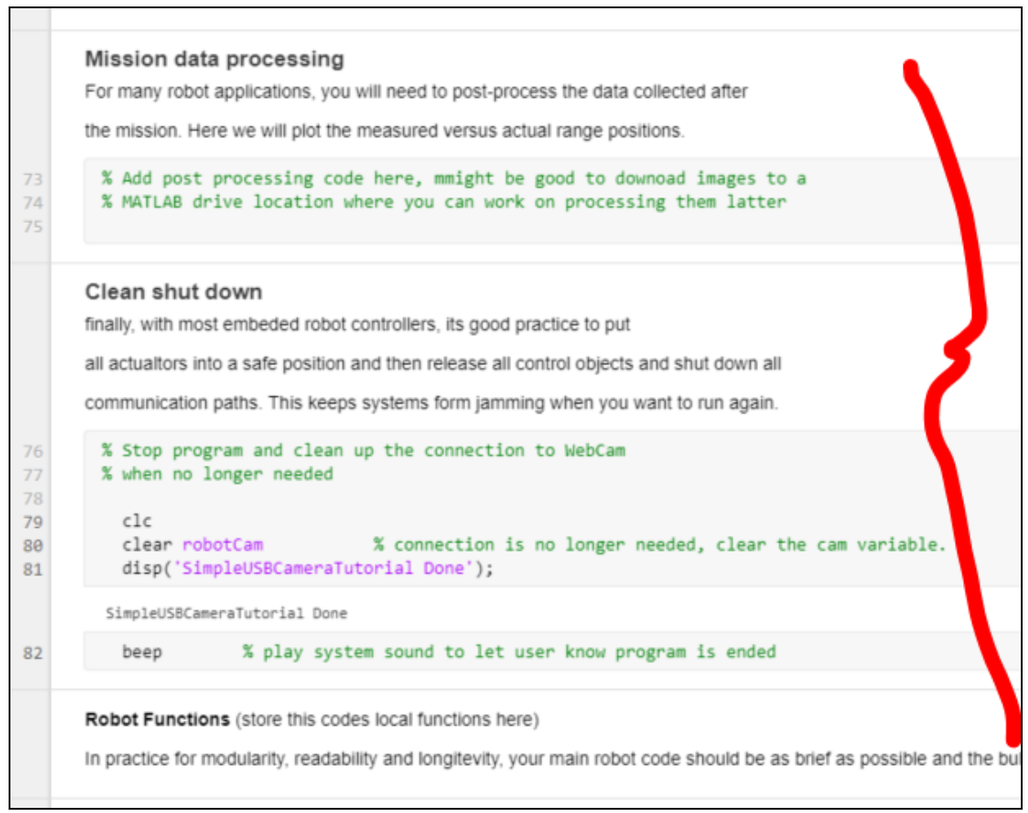
```
142     clc;
143
144     %%  Calculate the area of the target
145     targetArea = regionprops(cleanImage,'area');
146     area=targetArea.Area;
147     % Store the  x and y coorinates in a two column matrix
148     centroids = cat(1,targetCenter.Centroid);
149     % Display the binary image with 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 task, 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:

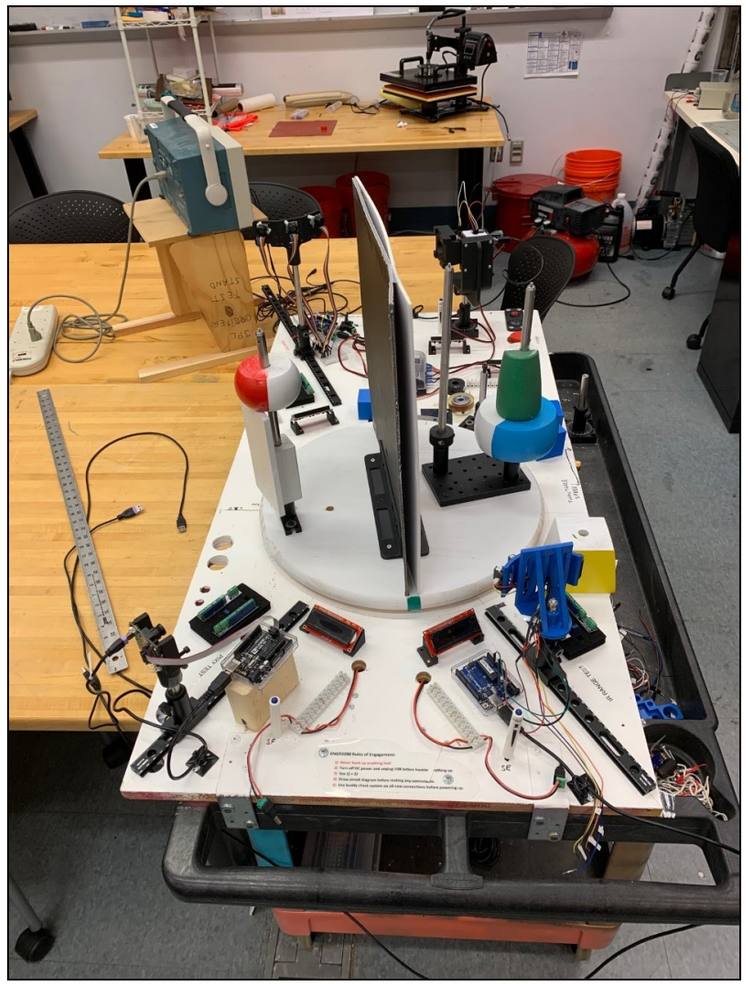
Find Target:

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).

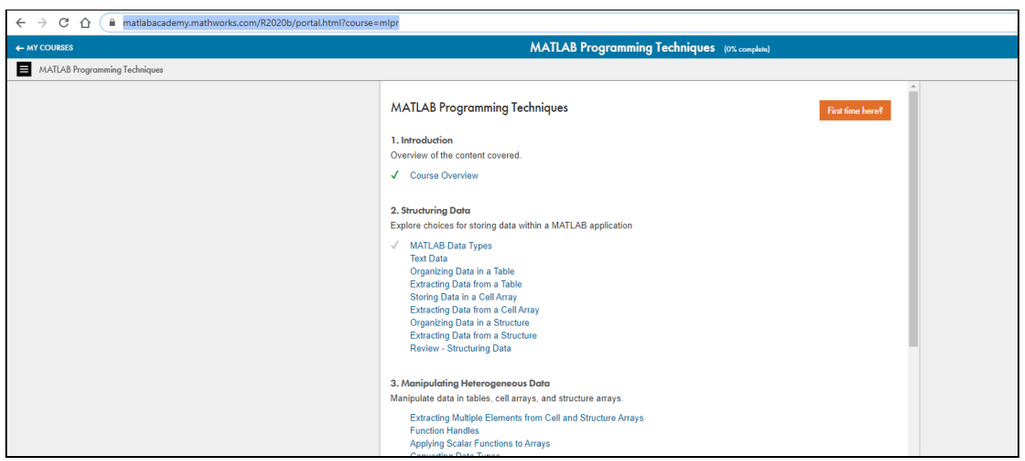
If you have time you can take on a stretch goal and try to build your code into an App with the App building skills you acquired in the MATLAB tutorial. As always, please see an instructor or Ninjas for help with any part of the above.



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.

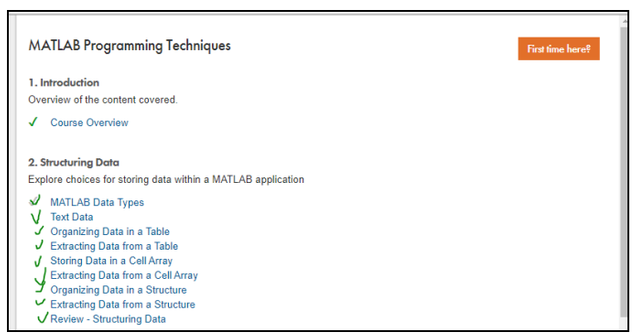
Please see a course instructor or Ninjas for MATLAB help as needed.

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