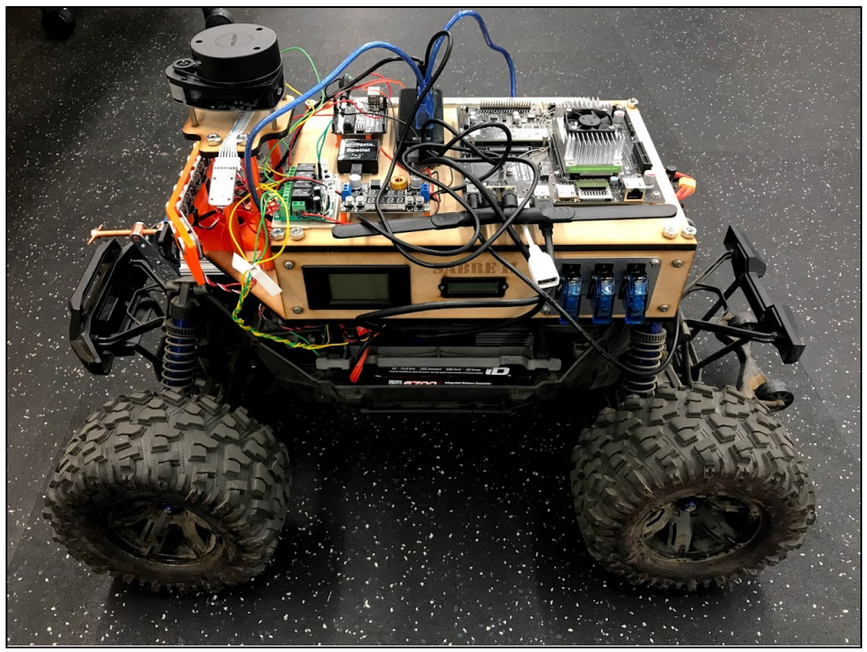
ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

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.

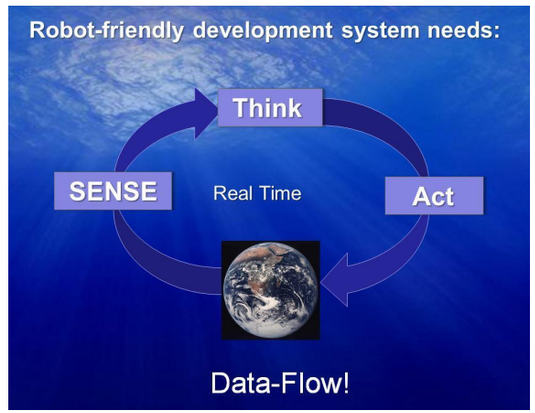


Figure : Sense-Think-Act Control Loop

ACT: involves both designing and adding actuators {motors, servo-motors, linear actuators, transmissions, etc.} and on-board operator communications devices {behavior lights, LCD displays, speakers, etc.} to a robot to allow it to move around in the real world, do useful work on the real world, and to indicate to any people in its immediate area what it is doing. The goal of this ACT lab is to develop your skills in choosing the appropriate actuator, learning to electrically hook it up into your robot’s existing circuitry, writing the correct position, velocity, light-pattern or sound software to control it, and commanding that actuator to perform one of the appropriate robot actions described 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 common robotic actuators connected to your Arduino and robot controllers, as well as advancing your MATLAB programming skills on your robot laptop controller.

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

ACT Lab

The ACT lab contains contains 6 commonly used robot actuator test stations, each housing a commonly used actuator mounted on a common MechE friendly breadboard that would allow you to explore adding simple robot actuator transmissions (as small stretch subprojects, if you are interested to do so).

Motion actuators generally have two main system functions; **Move to a location** or **Run at a speed**. The first lets you move a fin, arm, sensor-platform, tool, etc. to a desired location or orientation; the second lets you move your robot via wheels, propellers, treads, legs, jets, etc. toward a desired goal somewhere in space. Operator communication actuators use colored-LED-Lights, LCD-displays or speaker based-sounds to alert people near the robot as to what it is or intends to be doing.

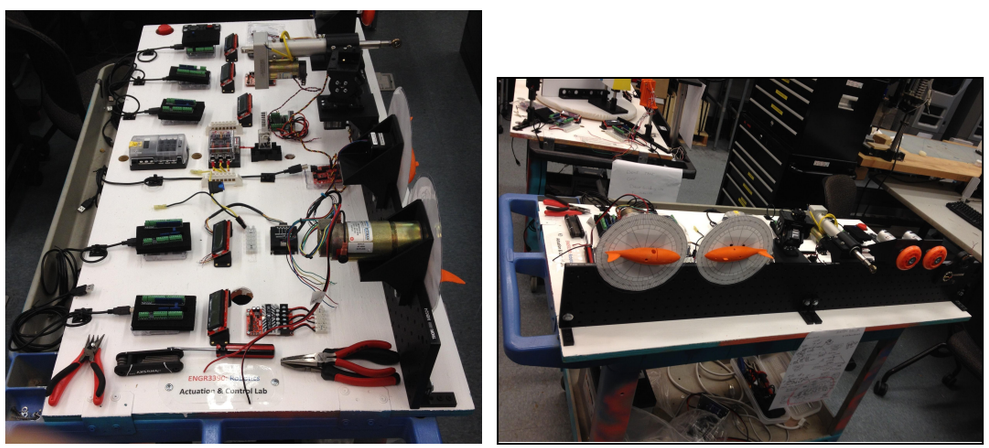


Figure: Act Lab Hardware

Position Control: Brushed DC Gearmotor/encoder with RoboClaw Drive

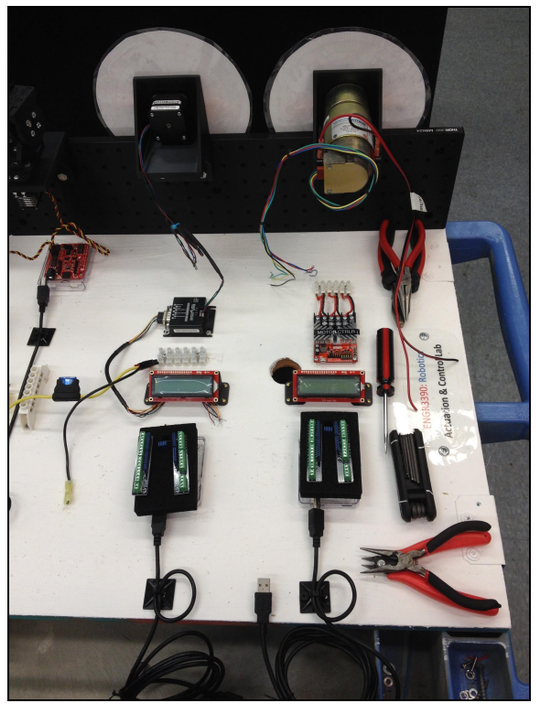


Figure: Right side, DC gearmotor actuator

The motor we are using is a brushed Pittman DC motor with a gear ratio of 65.5:1. The motor has an encoder attached to it. This means that we can read out the position of the motor through a signal from this encoder. In the datasheet of the motor (online or via Canvas) you can find the function of each of the motor wires.

The technical specs for this motor can be found on this webpage: http://www.servocomponents.com/Pittman-Gear-Motor-GM14000-series/14902s018-r1

At the top is the voltage the motor takes, which we see is 12V.

Assembly Data	Symbol	Units	Value
Reference Voltage	E	V	12
No-Load Speed	S_{NL}	rpm (rad/s)	59 (6.2)

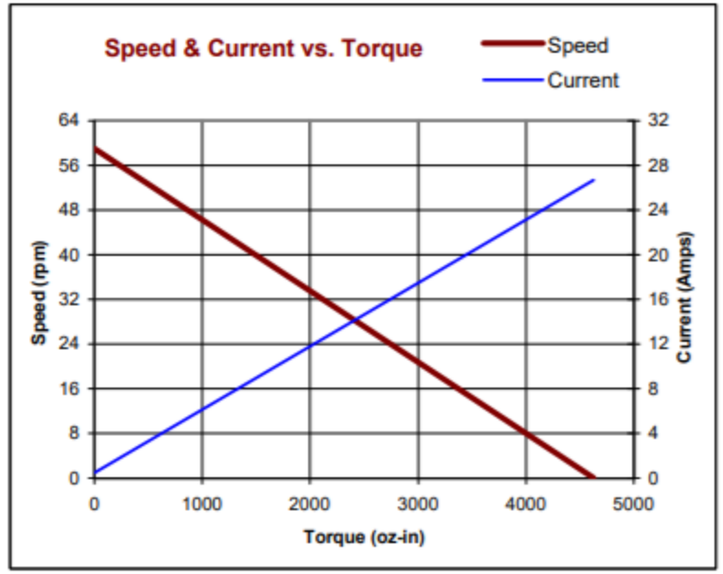
More useful information includes the specs for current. When under no load (not trying to drive anything), the motor still pulls almost half an amp. This is important when determining how long your robot will last on battery power. Also included is the stall current, which we will translate into stall torque in a moment. This is also a useful number to keep in mind when designing the electrical system and picking out a fuse.

No-Load Current	I_{NL}	A	0.49
Peak Current (Stall)[2]	I_P	A	26.7

One last thing we want to check out is the reduction ratio and efficiency. Since the RoboClaw will be handling the processing of the encoder values, we're not too worried about the reduction ratio. However, it is nice to know the efficiency of the motor.

Gearbox Data			
Reduction Ratio			65.5
Efficiency[3]			0.80
Maximum Allowable Torque		oz-in (N-m)	500 (3.53)

More importantly, at the bottom is a Speed & Current vs. Torque curve. From there you can determine how much torque the motor provides at a given rotation speed, as well as how much current it will pull at those torques.

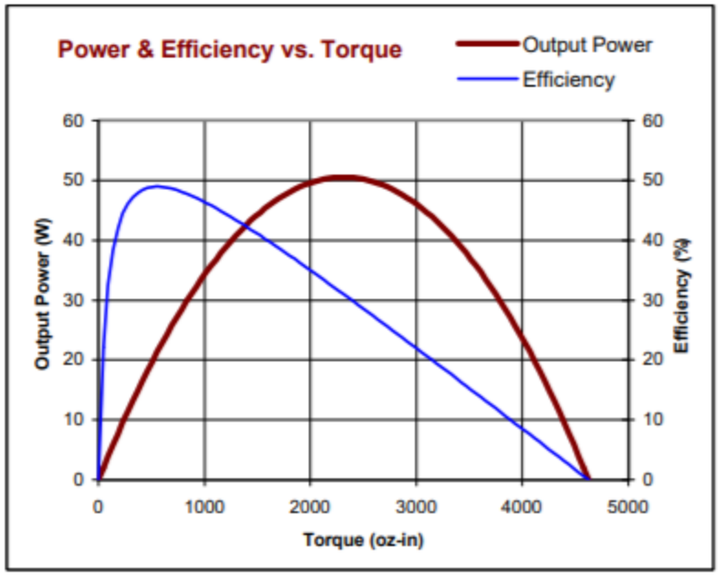


Next to that graph is the Power & Efficiency vs Torque graph. Notice that the highest efficiency occurs when the motor is using about 500 oz-in of torque. When being powered by a battery, as mobile robots tend to be, efficiency is more important than power output. Keep in mind that the motor's efficiency decreases as the torque exceeds 1000 oz-in.

The Encoder

The encoder mounted to the end of the motor has five wires. There's a red wire for power and a black wire for ground. This particular encoder is a quadrature encoder, meaning it has two steady pulses, one on the A wire, and an opposite pulse on the B wire. With the two of those, it's possible to determine whether the motor is going forwards or backwards. The Index line provides a pulse every time the motor spins;

due to the reduction ratio, the Index line will send 65.5 pulses per rotation of the shaft.

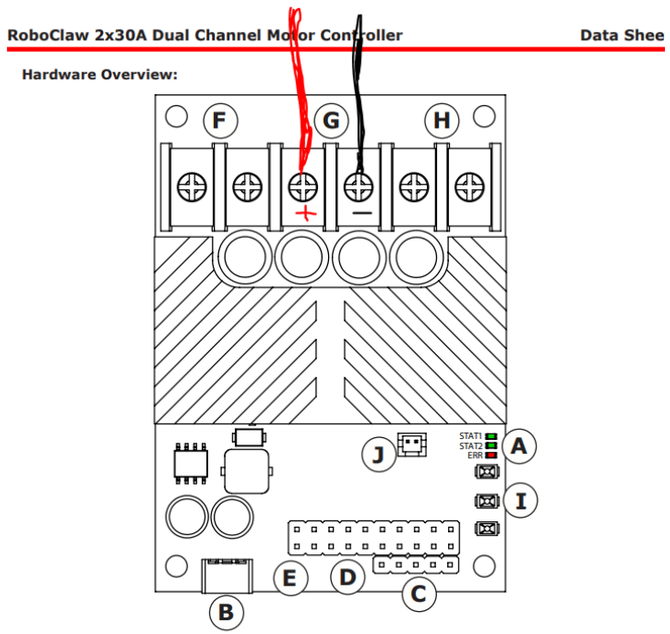


RoboClaw (the Motor Controller)

You will notice that under the motor is a RoboClaw controller. The datasheet and the user manual for this controller can be located in Canvas Act file and at: http://downloads.ionmc.com/docs/roboclaw_datasheet_2x30A.pdf and http://downloads.ionmc.com/docs/roboclaw_user_manual.pdf.

Another very useful feature to control the motor driver and to change the settings is the program provided by RoboClaw: IonStudio. To use this you have to install the USB drivers and the program itself. The RoboClaw is a complex microprocessor based “smart” robot controller. It can do position, velocity and coordinated motion of up to two brushed DC servomotors. Setting it up will take multiple steps, so just proceed carefully step by step until you are done. We will have you set up RoboClaw first, then add in your Arduino using MATLAB code to finally control it.

You cannot power the RoboClaw (or any large actuator) via the USB-port of your computer (it just doesn’t have enough amps) so you need 12V power from a big Robot battery. To power the board from the 12V from the battery under the test stand, you need to carefully connect 12VDC and Ground wires to the **+** and **- screw terminals** (with power OFF!). You will need to power it up to program it.



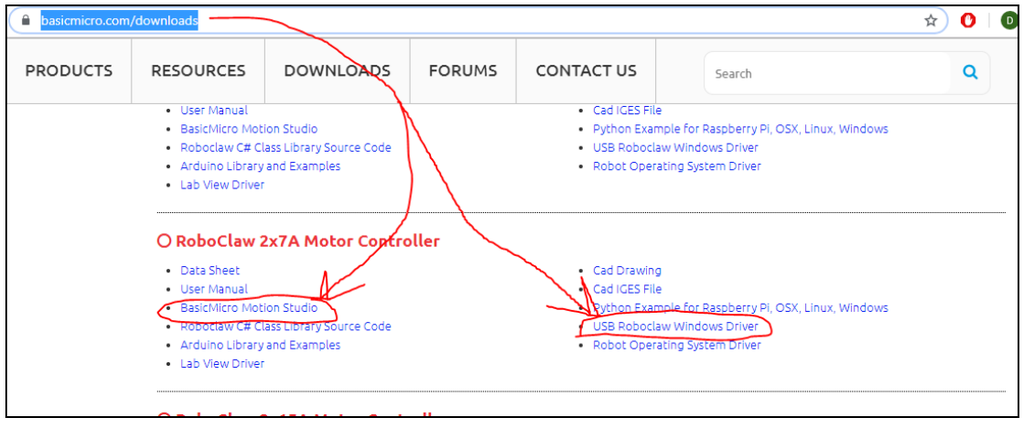
ID	Function	DESCRIPTION
A	Status LEDs	Provides RoboClaw status information.
B	USB Port	Communicate with RoboClaw via USB.
C	Control Inputs	S1,S2,S3,S4 and S5 control inputs.
D	Encoder Inputs	Dual encoder input and power pins.
E	Logic Battery	Logic battery jumper setup and logic battery power input.
F	Motor Channel 1	Motor driver output screw terminals for channel 1.
G	Main Battery	Main battery screw terminal input.
H	Motor Channel 2	Motor driver output screw terminals for channel 2.
I	Setup Buttons	Configure RoboClaw. Can bypass and use IonMotion PC setup utility.
J	Fan Control	Automatic fan control. 5VDC Fan. On at 45°C and off at 35°C

Please use care when hooking up wires. **Never, never-ever plug in wires hot!** Always shut down all power when making or unmaking any connections. After you

connect your wires, get someone else to double check you before powering up. Next let's get your RoboClaw hooked up.

Step 1. Download RoboClaw Motion Control Studio

Please go to https://www.basicmicro.com/downloads and download the BasicMicro Motion Studio application. This will let you set up the PD control for and check out the encoder on your finger servos.



Step2. Download the USB Roboclaw Windows Driver

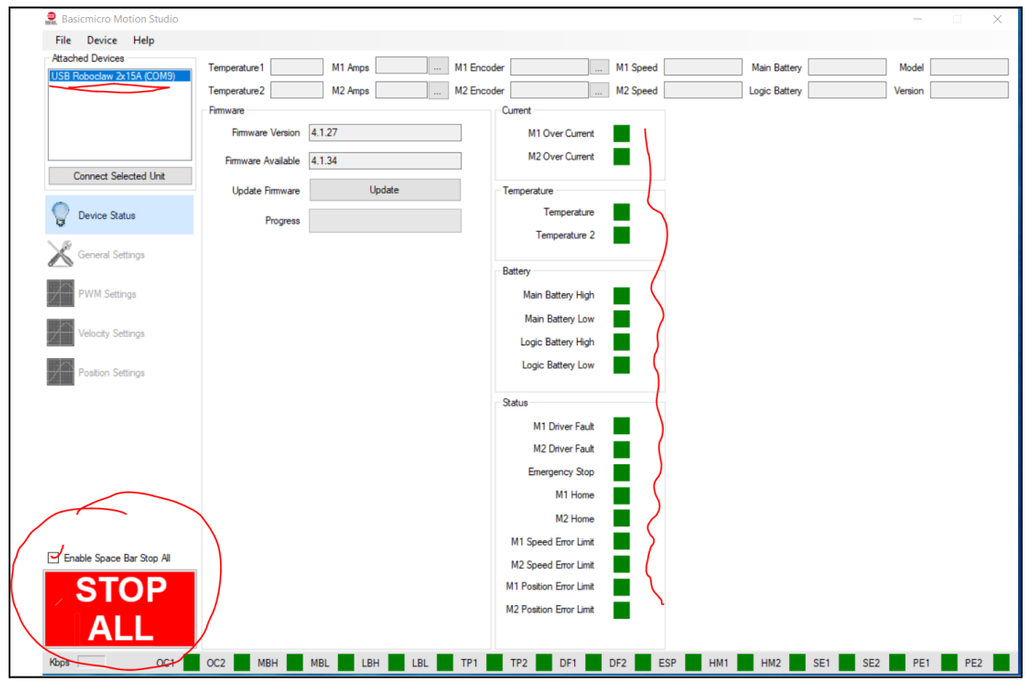
Please download the USB windows driver from the same location, your Laptop will need this driver to communicate with the mico-controller on the Roboclaw. After downloading, install driver. See course instructors for help if needed.

Step3. Setup and Tune RoboClaw for position control

The BasicMicro Motion Studio has extensive on-line documentation support. You can find it here:

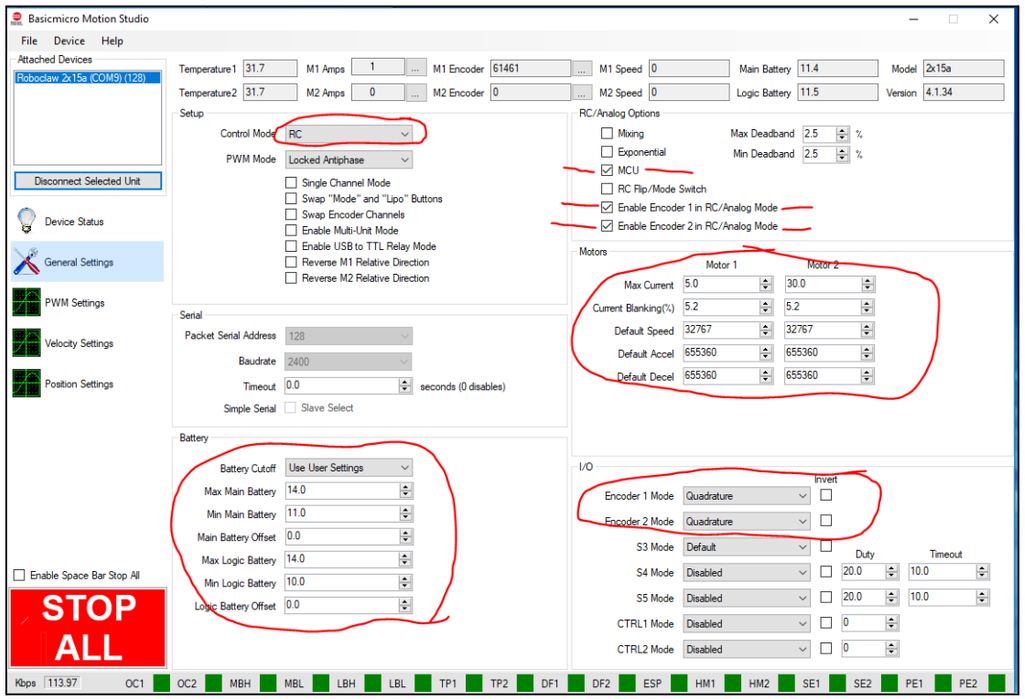
https://resources.basicmicro.com/layout-and-common-operations-in-motion-studio/

Open up the **Motion Control** Application, plug the provided mini-USB cable into you laptop, select your Roboclaw and click on **Connect Selected Unit** to get:

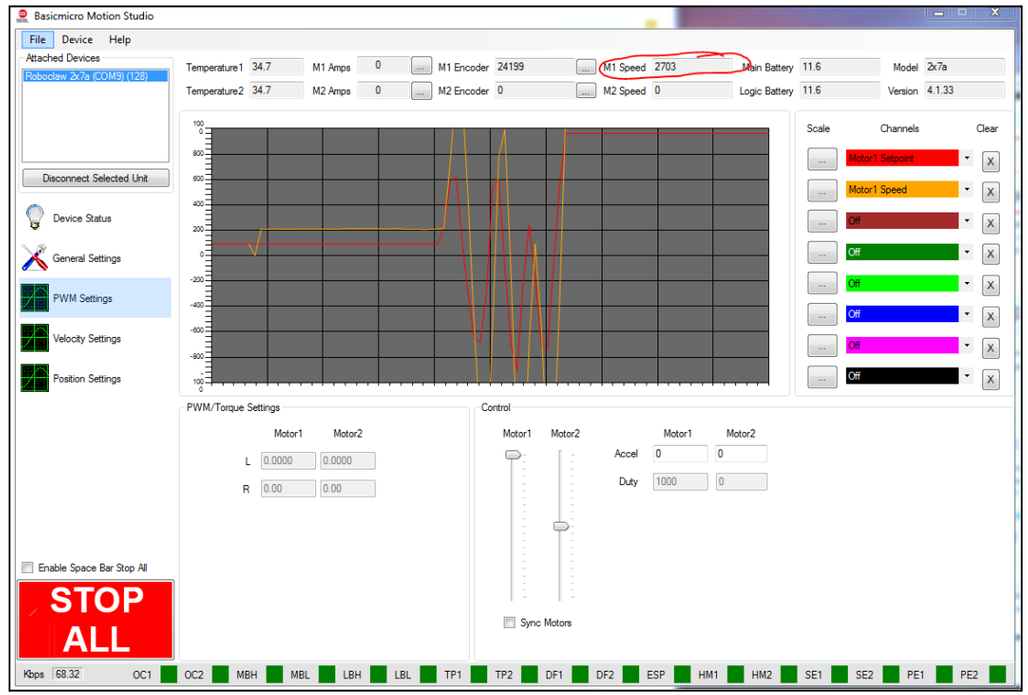


Set the **Enable Space Bar Stop All** check box. This will be your **EmergencyStop**.Then click on the left hand **General Settings** and set the **Control Mode** to **RC,** Set the **Battery voltages** and **Max Current** as shown, select the two **Enable Encoder 1 and 2 in RC/Analog Mode** radio boxes as shown, and set the two encoders as **Quadrature** as shown. (if you want to really know how to do this type of work independently and what all these control settings mean, please take a

look at their giant hundred page **User Manual** that you can get to online under the **Help** pulldown menu of the **Basic Motion Studio Application**).



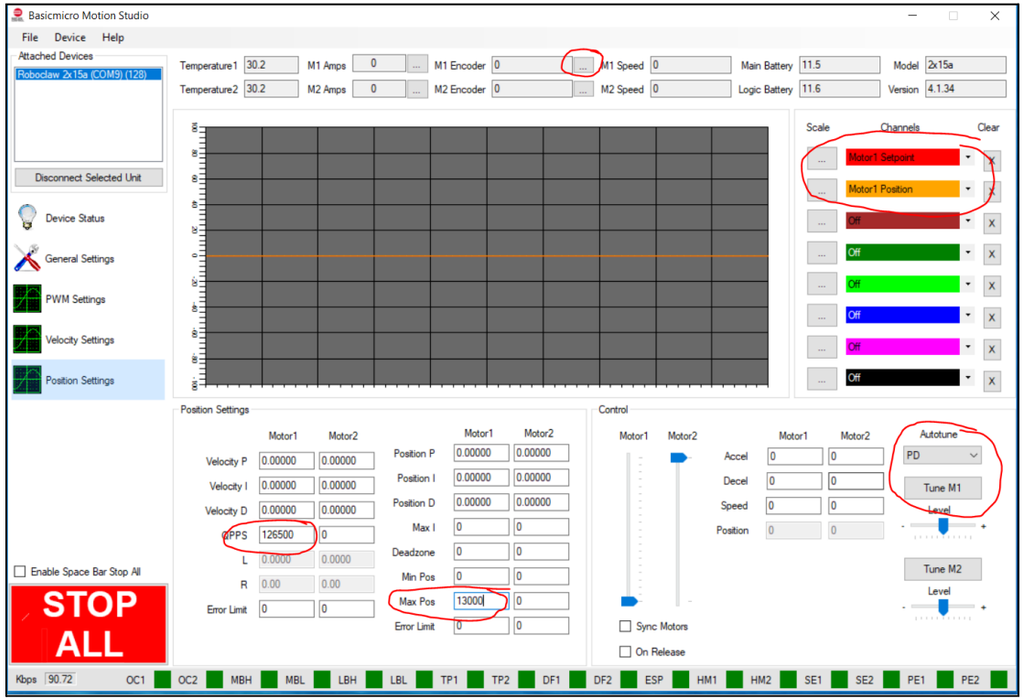
slowly rotate the motor to see how many counts are in one full revolution (around~131000).



On the **PWM Setting** Tab, set up the first channel of digital scope to be **Motor1 Setpoint** and the second channel to be **Motor1 Speed**. Select the **Motor1 slider** and move it up, if your encoder is wired correctly the **M1 Encoder** value should increase in a positive direction and the **M1 speed** should be positive. If it's not, your Encoder channel A and B are probably backward. Please re-wire them. Next take the **Motor1 speed slider** and move it all the way to the max, read the **M1 Speed** value (in Quadrature Pulses per Second QPPS) and write it down for use in setting the Position servo values (should be ~126500). Next zero the **M1 Encoder** with **… button** (three ellipses in box) to the right of the encoder readout) and then very

Next go to the **Position Setting** tab. Enter the max QPPS speed you measured on the previous tab (~126500) into the **QPPS** box as shown. Enter the **Max Pos** for one full revolution as shown (~131000) in the **Max Pos** box. Then select the **Motor1 Setpoint** and the **Motor1 Position** as the first two channels on the digital scope as shown. Zero the **M1 Encoder** (**press …button**). Disconnect any transmission (you may have added to the motor breadboard) on the motor and select the **PD** control option under **Autotune**, and then when all fingers are clear hit **Tune M1.** Motor M1 will go through a set of oscillations and then the application will set an optimum set of

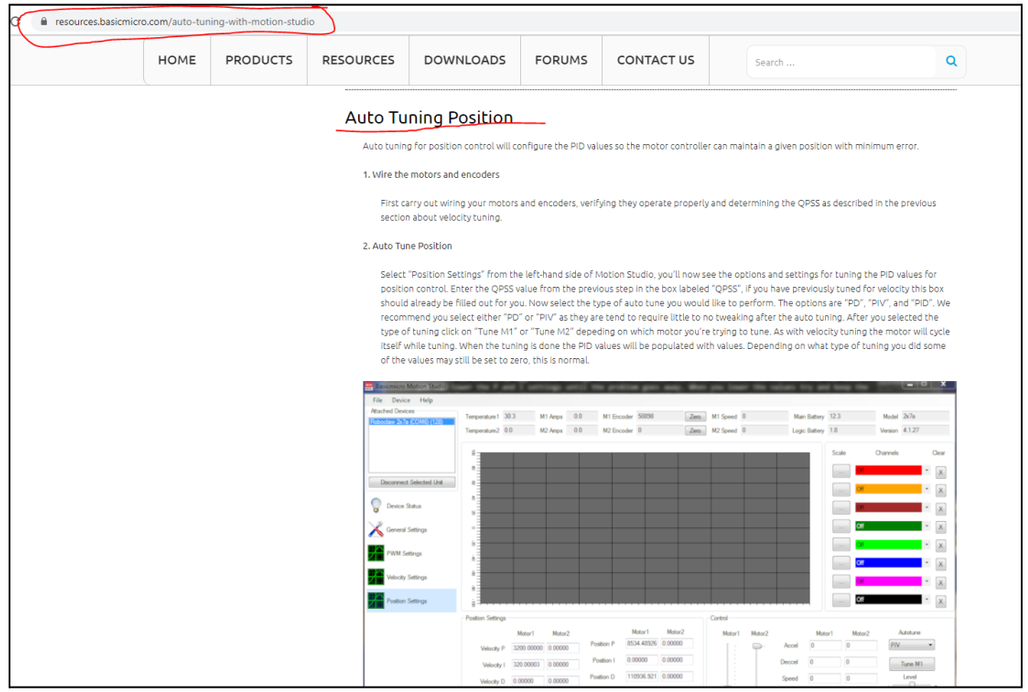
position servo control values for it. You can then play with **Motor 1 Position Slider** to move your motor from position to position.



Your Servo Motor will oscillate through a little waggle dance and then autoset the correct PD gains for a safe stable position control loop. Compacting a semester's worth of controls course into a single button push, you've just optimally tuned your servo system for accurate, responsive position control.

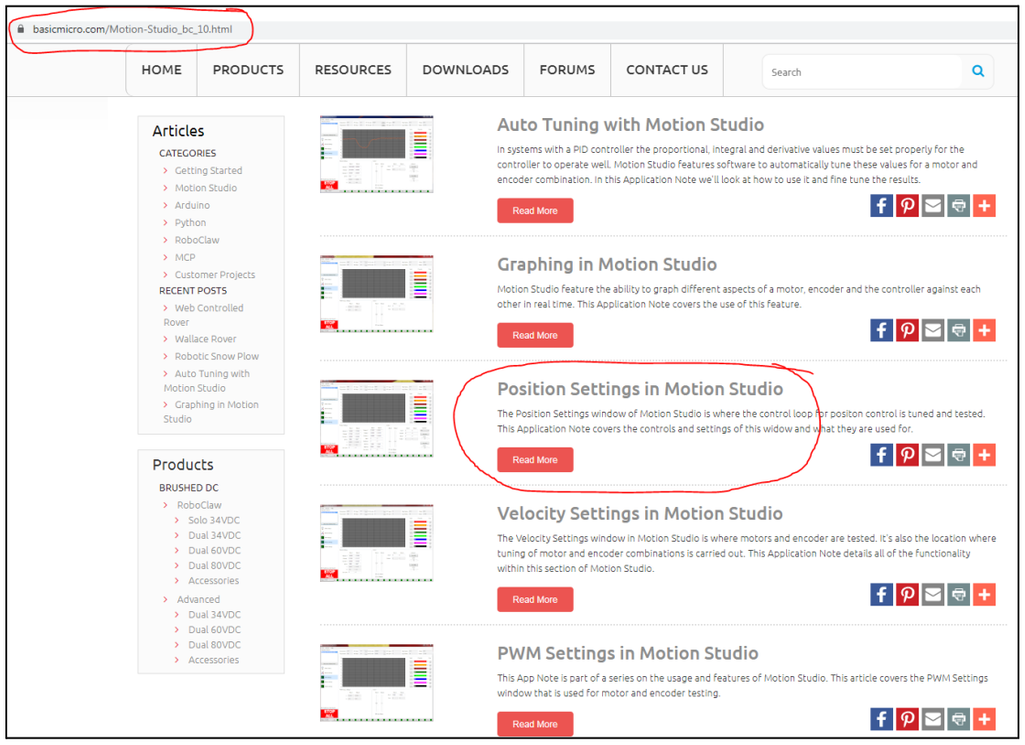
For a more indepth on-line tutorial on optimizing these control parameters, and the PID and PIV options see:

https://resources.basicmicro.com/auto-tuning-with-motion-studio

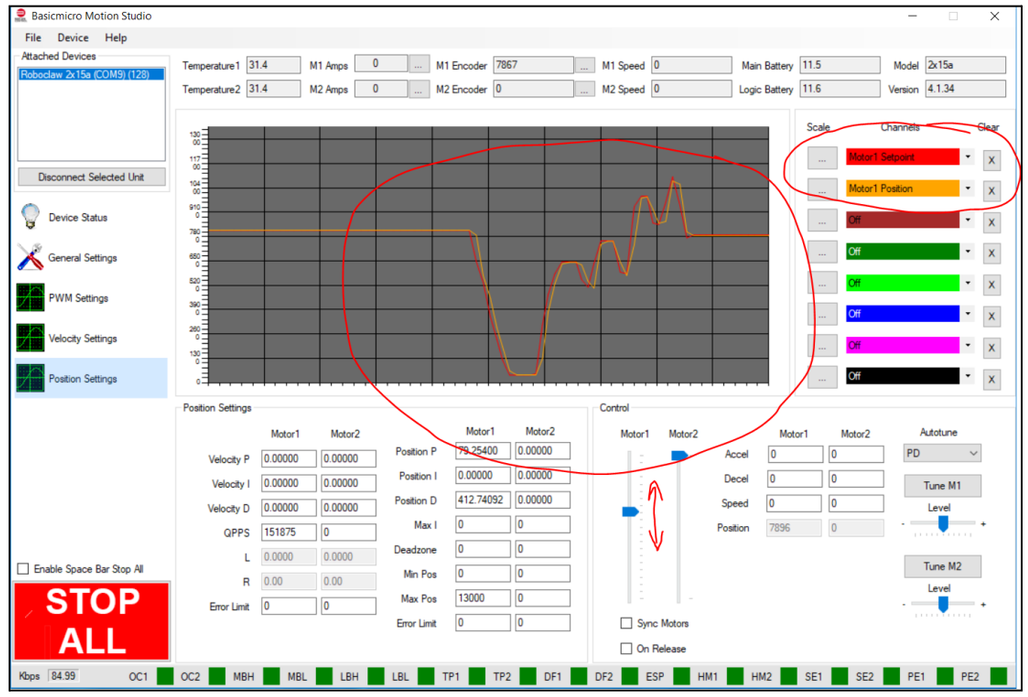


The Roboclaw is an awesome little servo control system for all kinds of robot applications. It dramatically simplifies adding DC gearmotors to any robot. For many more detailed on-line tutorials about the many other ways you can use this very versatile roboClaw motor controller, please see the tutorials at::

https://www.basicmicro.com/Motion-Studio_bc_10.html



Please note, your motor has a relative incremental encoder on it. This means that it will treat its location on power up as zero and then count either up or down from that position. The Roboclaw will now limit the motion of the controller to the space between o and your Max Pos and will truncate commands given to it above or below them. You should zero encoder on power startup, each time you use it and try to leave the motor at a known zero position when you shut it down (more on using limit switches to overcome this latter on). Now take the Motor 1 position slider and move it up and down, the motor should follow your desired position commands with just a little lag and not much overshoot. To learn more about this whole servo motor control world, please consider taking Olin's Controls course.

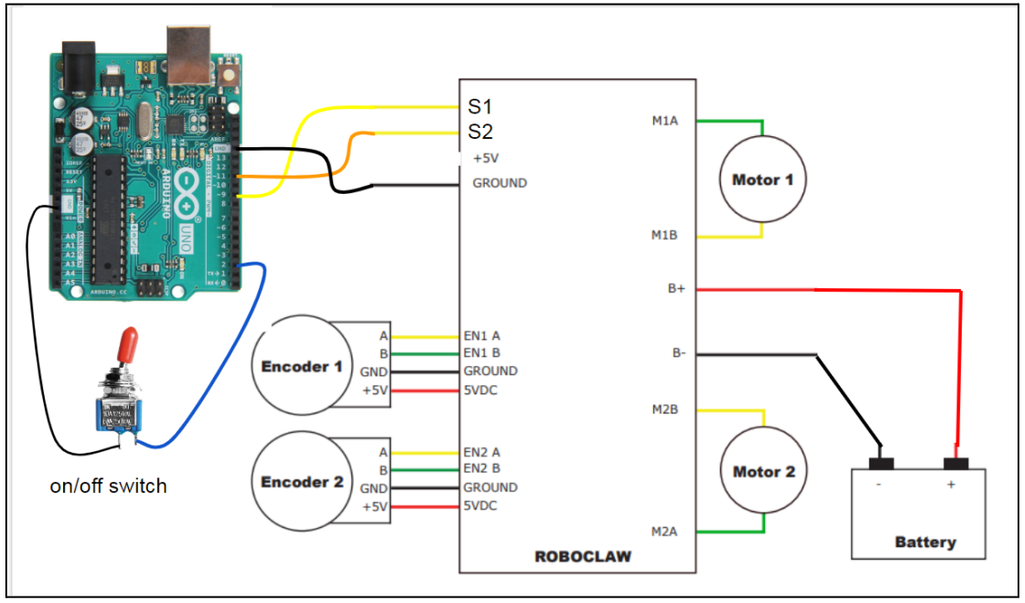


When you have your DC servo-gearmotor control all set up, please select **File> Save Settings** to create a backup file of it on your laptop. Then select **Device>Write Settings** to permanently download these settings to your RoboClaw. It will then always start up with the correct control setting loaded and you will no longer need the Motion Studio package to run your robot motor.

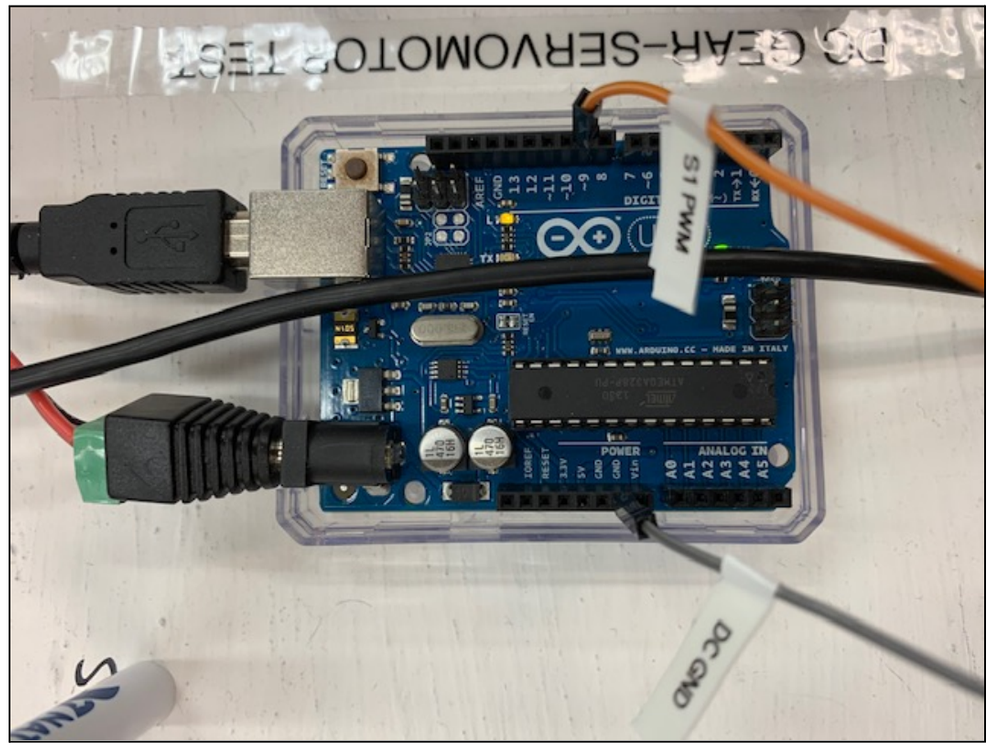
Normal operating procedure is to tune your Roboclaw with the **Basicmicro Motion Studio package** and then to internally save its parameters for use with your robots Arduino or Raspberry-Pi control system.

To drive the motor as a position servo motor, you now need to send a position PWM signal (needed by the RC interface) from a PWM-enabled pin on your Arduino or Raspberry-Pi to the S1 pin on the RoboClaw and provide a wire to connect their two reference grounds.

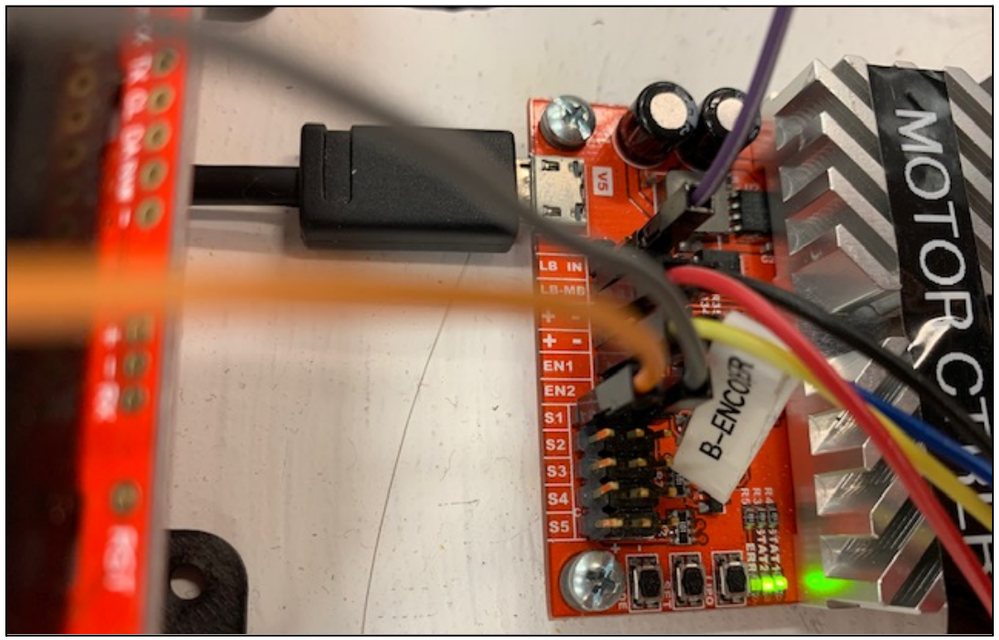
Let's look at the Arduino controller hook up first (please shut off all power when hooking up any wiring). To proceed, wire your Arduino to your Robo Claw as shown:



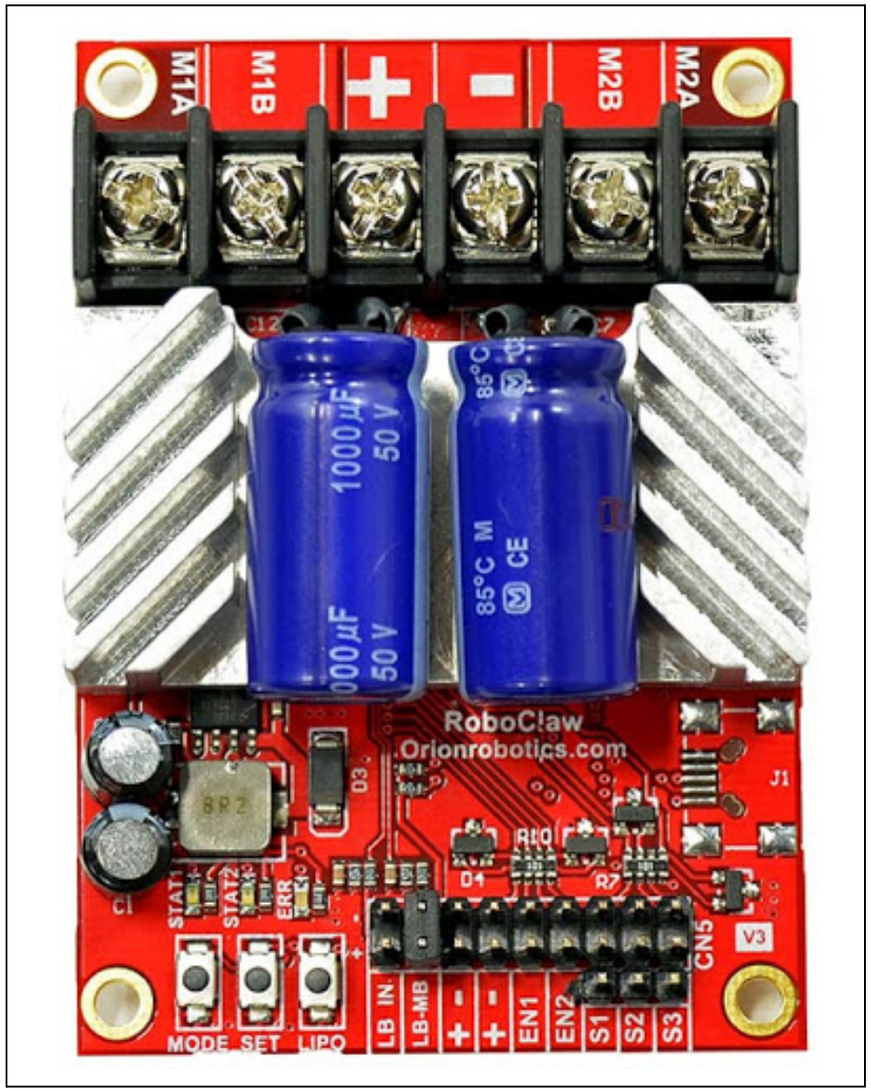
If you had two DC servomotors, you could hook up both as shown, we will focus on just Motor 1 in this lab.



Wire the S1 pin on RoboClaw to Pin 9 on Arduino. Ground on Arduino to reference ground on RoboClaw (- pin next to S1), see details above and below:



Please have your pair programming buddy check your wiring. Burning out your Arduino will put a damper on your day. Board is small and it's a bit hard to see where pins are, so large copy provided here:



Actual pinouts, listed in RoboClaw manual and provided here for convenience:

Control Inputs

S1, S2, S3, S4 and S5 are setup for standard servo style headers I/O(except on ST models), +5V and GND. S1 and S2 are the control inputs for serial, analog and RC modes. S3 can be used as a flip switch input when in RC or Analog modes. In serial mode S3, S4 and S5 can be used as emergency stop inputs or as voltage clamp control outputs. When set as E-Stop inputs they are active when pulled low and have internal pullups so they will not accidentally trip when left floating. S4 and S5 can also optionally be used as home signal inputs. The pins closest to the board edge are the I/0s, center pin is the +5V and the inside pins are ground. Some RC receivers have their own supply and will conflict with the RoboClaw's 5v logic supply. It may be necessary to remove the +5V pin from the RC receivers cable in those cases.

Wire 6 encoder wires from motor into RoboClaw pin sockets, for 5VDC power, gnd, Encoder Channels, as specified in RoboClaw document. Please consult color codes

Encoder Inputs

EN1 and EN2 are the inputs from the encoders on pin header versions of RoboClaw. 1B, 1A, 2B and 2A are the encoders inputs on screw terminal versions of RoboClaw. Channel A of both EN1 and EN2 are located at the board edge on the pin header. Channel B pins are located near the heatsink on the pin header. The A and B channels are labeled appropriately on screw terminal versions.

When connecting the encoder make sure the leading channel for the direction of rotation is connected to A. If one encoder is backwards to the other you will have one internal counter counting up and the other counting down. Refer to the data sheet of the encoder you are using for channel direction. Which encoder is used on which motor can be swapped via a software setting.

and wire really carefully. Please don't burn out the encoder, we only have one!

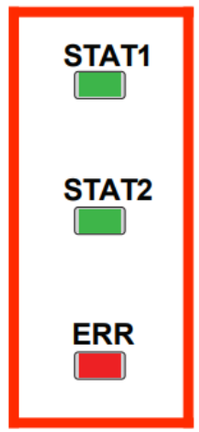
Have someone else on your team double check your wiring before powering the system up for the first time. When attached to the Arduino, Motion Studio lets the Arduino view the motors state but the Motion Studio USB command system no longer works (slides on GUI are disconnected), but you can see the encoder values and you may have to unplug from Arduino to re-tune both the Motion Studio control parameters and your MATLAB code to get the design outcome you seek.

If something goes wrong the three LEDs on the RoboClaw can help you debug it:

Status and Error LEDs

RoboClaw includes 3 LEDs to indicate status. Two green status LEDs labeled STAT1 and STAT2 and one red error LED labeled ERR. When the motor controller is first powered on all 3 LEDs should blink briefly to indicate all LEDs are functional.

The LEDs will behave differently depending on the mode. During normal operation the status 1 LED will remain on continuously or blink when data is received in RC Mode or Serial Modes. The status 2 LED will light when either drive stage is active.



The number and sequence of flashes can help put a fine point on the error that is currently stopping your robot from working.

Error light codes here:

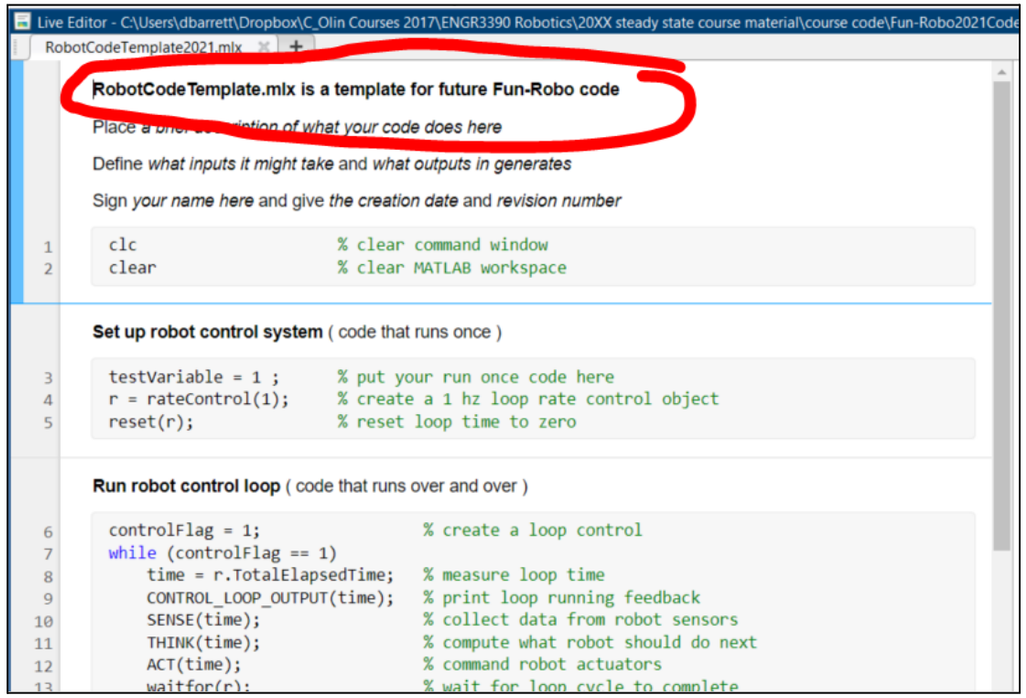
LED Blink Sequences

When a warning or fault occurs RoboClaw will use the LEDs to blink a sequence. The below table details each sequence and the cause.

LED Status	Condition	Type	Description
All three LEDs lit.	E-Stop	Fault	Motors are stopped by braking.
Error LED lit while condition is active.	Over 85c Temperature	Warning	Motor current limit is recalculated based on temperature.
Error LED blinks once with short delay. Other LEDs off.	Over 100c Temperature	Fault	Motors freewheel while condition exist.
Error LED lit while condition is active.	Over Current	Warning	Motor power is automatically limited.
Error LED blinking twice. STAT1 or STAT2 indicates channel.	Driver Fault	Fault	Motors freewheel. Damage detected.
Error LED blinking three times.	Logic Battery High	Fault	Motors freewheel until reset.
Error LED blinking four times.	Logic Battery Low	Fault	Motors freewheel until reset.
Error LED blinking five times.	Main Battery High	Fault	Motors are stopped by braking until reset.
Error LED lit while condition is active.	Main Battery High	Warning	Motors are stopped by braking while condition exist.
Error LED lit while condition is active.	Main Battery Low	Warning	Motors freewheel while condition exist.
Error LED lit while condition is active.	M1 or M2 Home	Warning	Motor is stopped and encoder is reset to 0
All 3 LED cycle on and off in sequence after power up.	RoboClaw is waiting for new firmware.	Notice	RoboClaw is in boot mode. Use IonMotion PC setup utility to clear.

As always, please see an instructor or Ninjas for help with connecting up any part of the above. After successfully wiring up your DC servomotor, position it carefully so that the fish nose pointer is at zero, the shut down power to it. The RoboClaw will use that zero position as encoder zero the next time it is powered up. Next, fire up MATLAB. You will now use the Matlab Arduino tools you developed in the previous tutorials to write a (reusable, your-toolbox) function to drive the position DC servomotor to precise rotational positions.

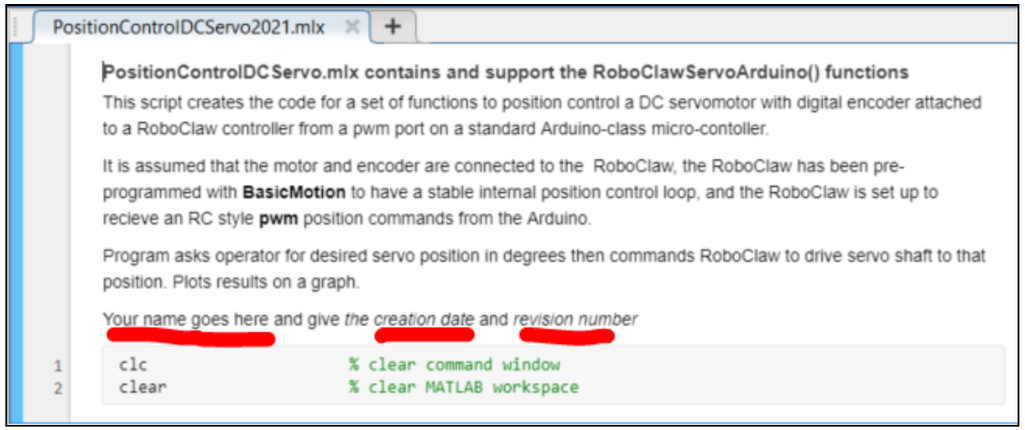
Please start up your MATLAB and open up your **RobotCodeTemplate:**



We will use and reuse this template to support building your robot Toolbox functions for each test station on this lab bench. Our joint goal is for you to write portable, flexible functions to control each actuator so that your future team can quickly reuse them for your final project robot, without having to go back and redevelop the wheel from scratch.

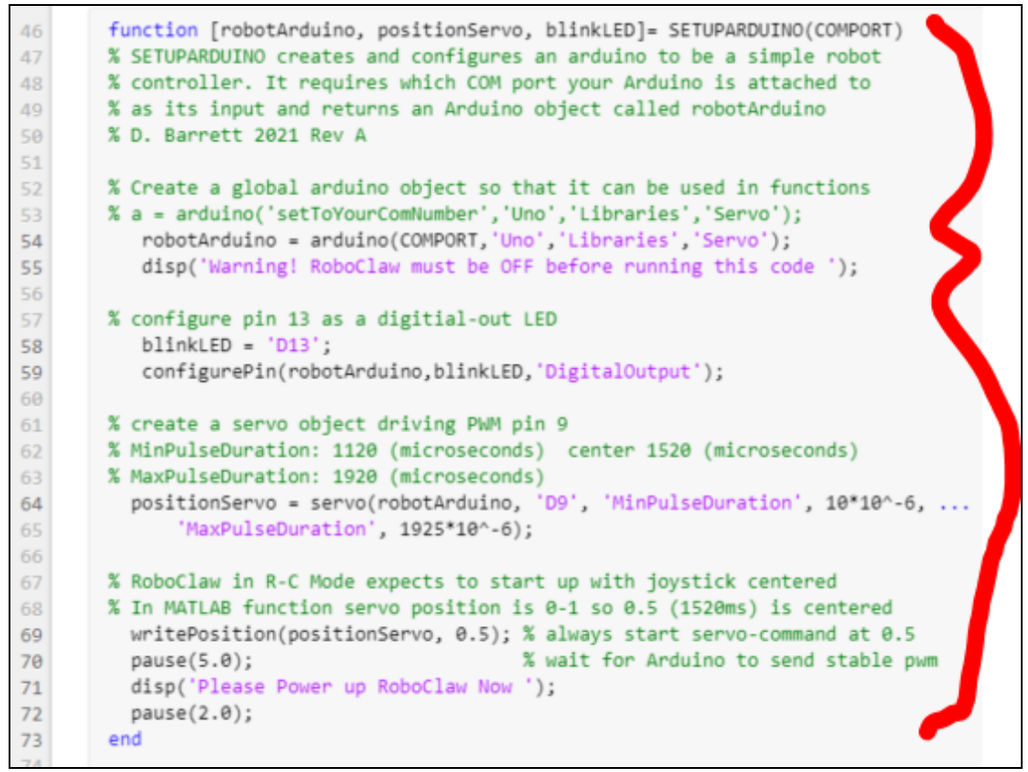
ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

Open editor, modify file and save in a MATLAB-Drive team repository so your pair programming partner and the rest of your team can have easy access to it. Please edit the first section as shown below:



In keeping with the policy of moving all complex code down into the **functions** part of your program, please move down to the functions area;

And add your first function in the general local function area. This function takes care of all of the details of setting your Arduino up to act like a robot controller:



It creates an Arduino object, sets a blinky light LED on Arduino for user feedback, creates a single pwm **servo** object on **pin D9** and centers the output of servo to the RoboClaw so that it will start up in midrange and not on one of its end of travel stops. There is a lot of text in the comments to explain how it all works. A good team coding style is 50% comment and 50% working code. This will help other folks figure out what you intend the code to do and help you figure out what you hoped your code would do, 6 months from now when you forget how you coded all of this!

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

The next general program function handles blinking your Arduino feedback light:

```
    disp('Please Power up RoboClaw Now ');
    pause(2.0);
end

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
```

Sense Functions (store all Sense related local functions here)

It's always good operator interface practice to put a blinky light on your robot so that you can tell at a glance if it's running or if something is stuck. You can reuse this **Blink** function over and over again.

Having created these two general program service functions to handle all the detailed setup of the Arduino, and placed them down in the **reusable functions** part of your code, you have achieved three goals. First, you've dramatically cleaned up the front end of your main control code so that your team and partners can get right to the robot part without scrolling through 30 lines of routine Arduino set-up code. Second, you've isolated this code so that if you change or upgrade your Arduino to something more powerful, you only need to replace this function, the rest of your working code stays untouched. That is a really powerful feature of modular reusable code. And third, you can reuse this function, lock, stock and barrel on your next robot project without needing to rewrite it from scratch. Writing code for reuse will save you hours and hours in this course and on future robot projects.

Moving back up to the top of your code, let's now use your two new functions in the section of "run-once" code:

Set up robot control system (code that runs once)

```
% create arduino and arduino-position servo ref-objects
  [robotArduino, positionServo, blinkLED]= SETUPARDUINO('COM3');
```

```
Warning! RoboClaw must be OFF before running this code
Please Power up RoboClaw Now
```

```
% Turn on board LED on and off to signal program has started
  Blink(robotArduino,blinkLED,3);
  disp('Warning! Position Servo Active! ');
```

```
Warning! Position Servo Active!
```

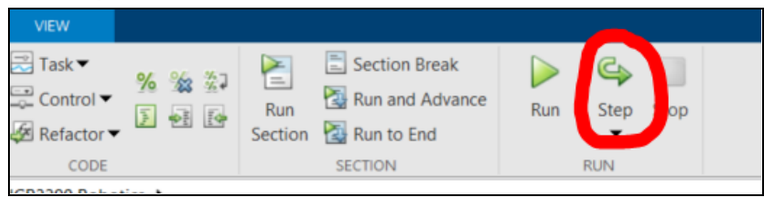
```
% Configure test loop to collect n data points.

nTests = input(['Enter number of servo test positions, ' ...
    'followed by enter: ']);

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

r = rateControl(0.25);                % create a 0.25 hz loop rate
reset(r);                             % reset loop time to zero
```

Please run and debug this section using the **Step** line by line debugger to make sure it all works before you move on:



Next, you will write the **SENSE-THINK-ACT** control code, before proceeding there, go back down to the **Functions** section and add the following Sense and think functions:

```
86      end

        Sense Functions (store all Sense related local functions here)

87      function commandAngle = SENSE()
88        commandAngle = input('Enter desired servo angle in degrees:  ');
89      end

        Think Functions (store all Think related local functions here)

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

        Act Functions (store all Act related local functions here)
```

Since this lab is mostly about Act, the Sense function just asks for operator command input, basically; where do you want motor to go? As there is no cognition involved yet, you can just create a null (doesn't do anything) Think function.

Moving back up to the **Robot Control Loop** part, add your two new functions in as shown:

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

18      controlFlag = 1;                % create a loop control
19      while (controlFlag < nTests+1)  % loop till ntests data captured
20
21          commandAngle = SENSE();     % collect deired position from operator
22          THINK();                    % compute what robot should do next
23          % command servo motor to desired position
24          ACTRoboClawServoArduino(positionServo, commandAngle);
```

Note: the functions lead to a clean readable robot control flow. Next go back down to the **Functions** zone and create the ACT function:

```
        Act Functions (store all Act related local functions here)

93      function ACTRoboClawServoArduino(positionServo, commandAngle)
94      % is an example of a function to control a RoboClaw DC motor
95      % controller in closed-loop, encoder-feeback driven position servo mode.
96      % Need to run SETUPARDUINO function first to create arduino and
97      % position servo objects.
98      % hook Arduino PWN pin 9 into Roboclaw digital input S1, hook Arduino
99      % Ground to RoboClawSI Ground.
100     % Input is (desired position in degrees)
101     % Output is motion of servo motor shaft
102     % dbarrett 1-13-20  RevC
103
104         % scale desired angle to 0-1 for writeposition Arduino command
105         % 0 =0 degs on servo, 1 = maximum servo angle (for DC motor=360)
106         scaleComAngle = commandAngle/360;
107         writePosition(positionServo, scaleComAngle);
108
109     end
```

This code takes the desired command angle input from the human operator, scales it to match the output range of the arduino **Servo** function and then moves (**Act**) the servo motor's shaft to that angle. In this case the RoboClaw is doing teh heavy PID lifting, reading the encoder and getting you fast servo performance. In other cases you will want or need to do that yourself within this body of code.

There is a lot of hookup and use documentation in the body of the Act function so that you can pop it out of this code and reuse it in another robot, without losing any of the critical implementation data. Document well once on creation, and you won't spend all of your time doing forensic code probing to figure out how the darn thing works the next time you need it!

Note: the control loop is done in three clean lines of code Sense-Think-Act

```
21      commandAngle = SENSE();     % collect deired position from operator
22      THINK();                    % compute what robot should do next
23      % command servo motor to desired position
24      ACTRoboClawServoArduino(positionServo, commandAngle);
```

Because this is an experimental setup, we will want to collect some experimental data to determine if our robot is acting the way we design it. It's pretty standard to record and save internal performance data when a robot is being built, to verify its operation and when it is running to aid in debugging complex system failures. To get you a little experience doing both, please add the remaining lines of control oop code:

```
26      % store experimental commanded versus actual data
27      positionData(controlFlag,1)= input('Enter actual position degrees: ');
28      positionData(controlFlag,2)= commandAngle;
29      Blink(robotArduino,blinkLED,1);
30      waitfor(r);                 % wait for loop cycle to complete
31      controlFlag = controlFlag+1; % increment loop
32  end
```

When you command a new position, the servo will move there. Using the built-in protractor on the test bench, you can then measure where it went. Both are stored in the **positionData** variable for future data analysis. Looking at full loop:

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

```
18      controlFlag = 1;                % create a loop control
19      while (controlFlag < nTests+1)  % loop till ntests data captured
20
21          commandAngle = SENSE();     % collect deired position from operator
22          THINK();                    % compute what robot should do next
23          % command servo motor to desired position
24          ACTRoboClawServoArduino(positionServo, commandAngle);
25
26          % store experimental commanded versus actual data
27          positionData(controlFlag,1)= input('Enter actual position degrees: ');
28          positionData(controlFlag,2)= commandAngle;
29          Blink(robotArduino,blinkLED,1);
30          waitfor(r);                 % wait for loop cycle to complete
31          controlFlag = controlFlag+1; % increment loop
32      end
```

The control loop will run **nTest** times. On each cycle, it will ask you to input a commanded angle (we recommend 0, 90, 180, 270, 360 to start). After you enter an angle, the RoboClaw will drive the motor to that angle, and wait for your next command. When the actuator has stopped moving, enter the actual angle it arrived at when you are requested to do so.

To proceed with a live motor test, first power up the RoboClaw. Fire up the **Basic Micro Motion** application. Make sure good motor parameters are loaded and use application to position the motor shaft at 0 degrees. Zero the encoder there. Power off the RoboClaw. Then run the positionDCServo.m file code on your Arduino. Your code has a delay built in to allow a stable 1474 microsecond pulse to be sent out by the Arduino PWM pin to sync up with the “joystick in neutral position” that the RoboClaw is expecting from a Radio Control (RC) unit.

Note: always turn on the robot controller, before turning on the roboclaw. You want tight control on the input to any servo controller, before powering up the servo amplifier. This will prevent the servo from flying widley out of control and damaging something based on a random floating input. If the RoboClaw servo was attached to a car’s steering wheel, you DO NOT WANT IT TO FLY BOUT UNCONTROLLED!

Once the program has sent out the correct null PWM pulse, it will prompt the user to turn on RobClaw. It will then ask you how many data points to sample, pick something like 10. Choose 90 degrees as a first servo position to test. The whale pointer on the position servo should move about 90 degrees. The program will ask you to enter the actual position that it went to. Read it from the motor protractor and then enter it.

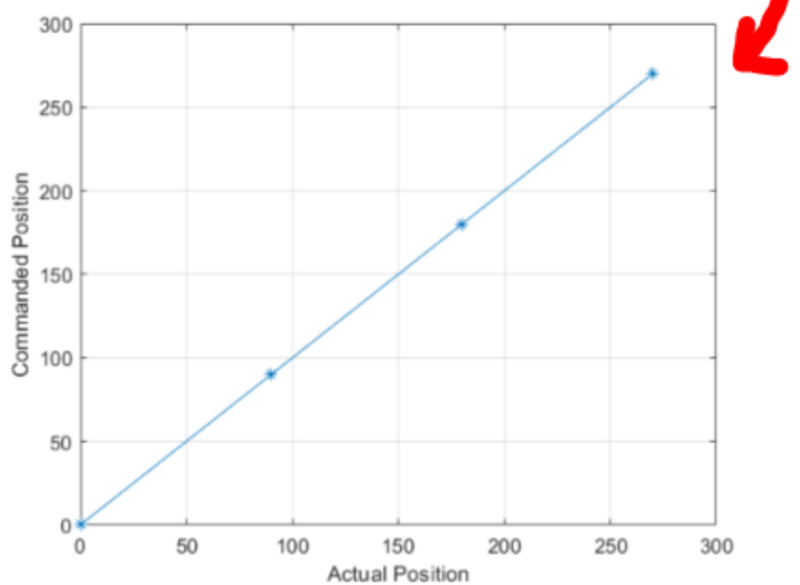
Repeat this last procedure for the ntests you chose to do. Please use the **Step** debugging command and walk line by line through your new code. You should get an output that looks like this:

After running experiential tests, when the control loop terminates, it's very useful to actively plot out the data that is collected to enable a clean review of system performance, it's also useful to store that data in a file for further use down the road. Please add the following code to do both and run it:

Mission data processing

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

```
33  % Plot and store commanded position data vs. actual position
34    plot(positionData(:,1), positionData(:,2), '-*')
35    xlabel('Actual Position')
36    ylabel ('Commanded Position')
37    grid
```




```
38    save positionServo.dat positionData -ascii
```

In the best case, this will be a clean straight line and you will have taken the first step toward reliable DC ServoMotor position control. If line is a little off, you may need to retune your RoboClaw servo based on the position data you got experientially.

Finally, in any active robot code involving embedded controllers, communication between processors or radios, it's critical you shut down each system cleanly and release their resources so that you don't leave anything in a jammed or wedged software state that will inhibit use the next time you turn it on.

Please add the following clean shutdown code, right before Function section:

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.

```
39  % Stop program and clean up the connection to Arduino
40  % when no longer needed
41    writePosition(positionServo, 0.5);      % always end servo at 0.5
42    clc
43    disp('Arduino program has ended');
```

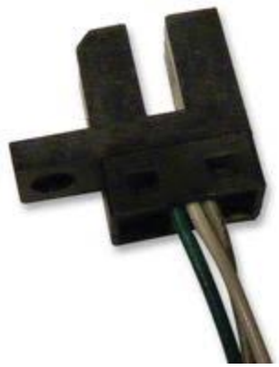
```
Arduino program has ended
```

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

Robot Functions (store this codes local functions here)

Two week demo: Please write this code, debug it, and be prepared to demo commanding servo to 4 locations, having it move to those locations and plotting out a clean performance curve showing the motor is under precise linear control.

Optional, add an optical home switch
As an optional stretch task, you can hook up the provided optical interruption sensor to provide a fixed home location for your motor. A QVE00112 optical switch has been added to the lab breadboard to support this.



The rotating whale has been given a short tail extension, which when passing through the optical switch, trips it, and this information can be used as an absolute positional feedback mechanism to a digital input pin on your Arduino. Its technical specs can be found in Canvas Act folder and here:

https://www.digchip.com/datasheets/parts/datasheet/161/QVE00112-pdf.php

You could easily write a short section of code that slowly rotates the motor until the break beam interrupt is seen and then zeros your motors position there.

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

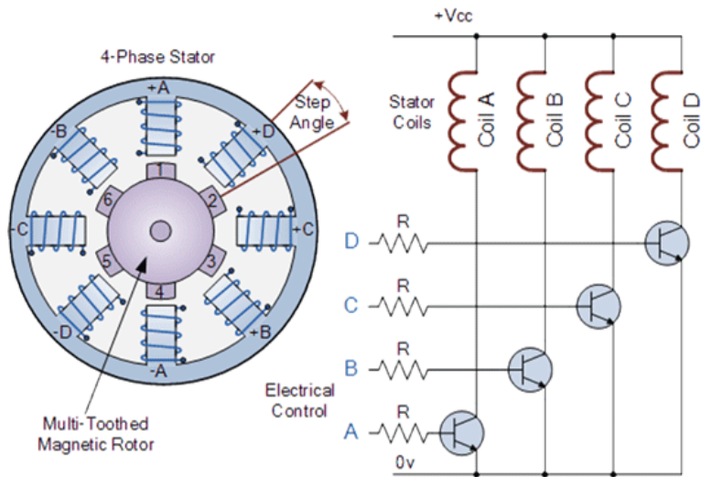
Two week demo: Please write this code, debug it, and be prepared to demo commanding servo to 4 locations, having it move to those locations and plotting out a clean performance curve showing the motor is under precise linear control.

.

.

Position Control: DC Stepper Motor

Stepper motors are DC motors that move in discrete steps. They have multiple electromagnetic coils that are organized in groups called "phases" around a central magnetically stepped rotor. By energizing each coil phase in sequence with a pulse, the motor can rotate one step at a time.



With a microcontroller it is possible to achieve very precise control on the position and/or speed of a stepper without the need for a servo or any other feedback mechanism. This makes it relatively simple to design and build with and is thus widely used in precision motion control applications. Stepper systems should be considered for applications requiring:

- High positional accuracy
- Precision low speed control
- Low speed high torque applications.

There are however certain limitations associated with the stepper when compared to other DC motors w/o servos.

- Higher vibration in operation (steppers are acoustically noisy).
- Limited high speed torque
- No feedback mechanism
-

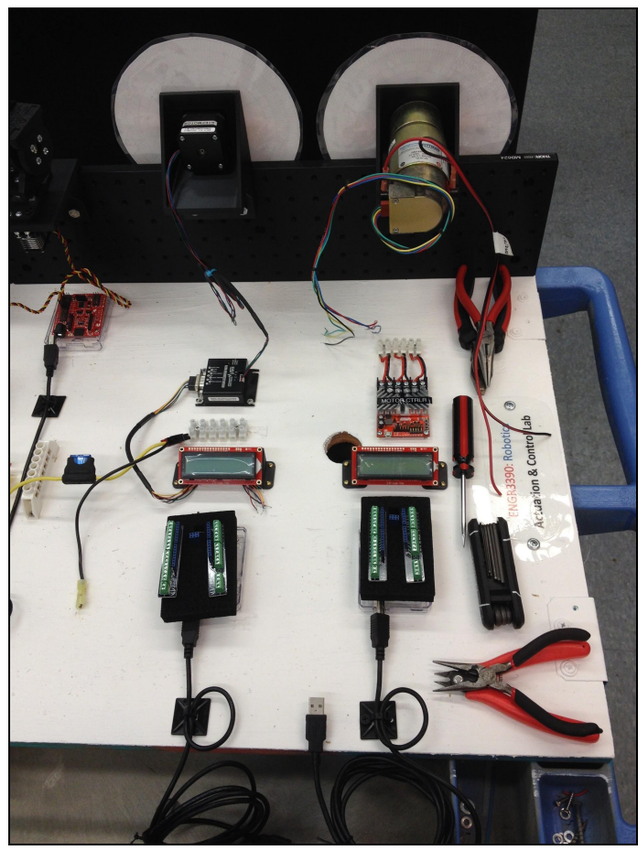


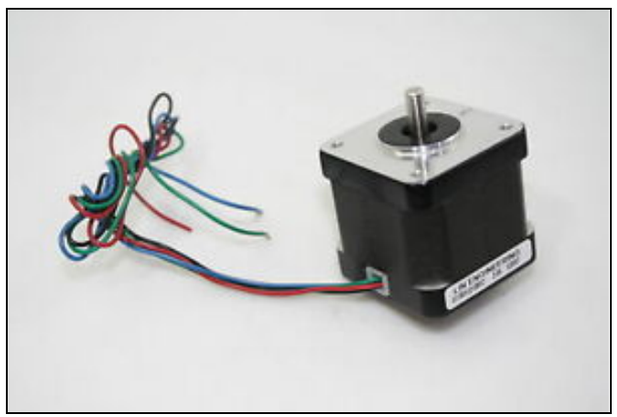
Figure : Left side, Stepper Motor Actuator

The Stepper Motor

The stepper motor in the Act lab setup is a Lin Engineering **4118M-01RO** High torque, Bipolar, two phase stepper. Its technical specs can be found here:

https://www.linengineering.com/products/stepper-motors/hybrid-stepper-motors/4118-series/4118M-06P/WO-4118M-06P

It has a rating of 1.7A, 1.5 ohm, 3mH, and has a full-stepping resolution of 1.8° (or there are 200 steps per revolution).



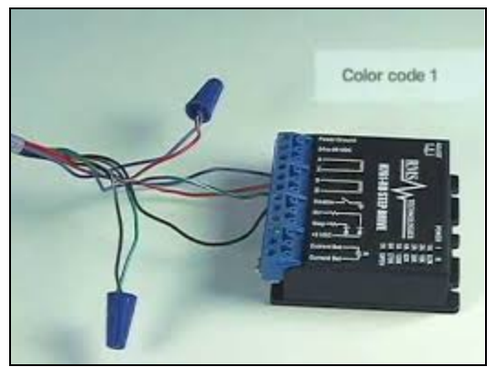
Bipolar steppers are the strongest type and generally more expensive than unipolar ones. Bipolar steppers require slightly more complicated electronics to be able to reverse the direction while unipolar steppers simply have a center tap that only operates half a coil for each direction. It is not critical to understand the inner wiring of the stepper, but if needed please refer to this article:

https://www.circuitspecialists.com/blog/unipolar-stepper-motor-vs-bipolar-stepper-motors/

The Microstepping Driver

The Act lab setup has a **R208 microstepping driver**. For more details please see**:**

https://www.linengineering.com/products/value-add/drivers-and-controllers/r208



A microstepping driver enables the stepper to take steps finer than 1 full-step, by providing it with partial pulses. This driver enables full, half, 1/4th and 1/8th steps which correspond to stepping angles of 1.8°, 0.9°, 0.45°, 0.225°. The driver has an optically isolated logic circuit and uses the 12V power source to control power to the stepper coils. The isolated logic circuit takes in digital outputs from the microcontroller for Vcc, ground, Enable/disable switch, direction switch, step resolution switch 1, step resolution switch 2, and a pulse signal.

Connecting two digital output pins from your Arduino to the step and direction input wires to the R208 will allow you to control the direction and the number of steps that your stepper motor will take. At the falling edge of an Arduino digital out pulse (minimum step width is 1 microsecond), the stepper rotates by 1 step or 1.8 degrees.

The controllers wire colors and pinouts are:

PIN #	COLOR (#26 AWG Lead)	FUNCTION	DESCRIPTION
1	**Red**	+V	Motor Supply Voltage. +12 to +24 VDC
2	**Black**	SR1 Input	Step Resolution 1. Pins 2 & 3 are used to preset the step resolution by selective contact to ground (Pin 7)
3	**Brown**	SR2 Input	Step Resolution 2. Pins 2 & 3 are used to preset the step resolution by selective contact to ground (Pin 7)
4	**Black/White**	Enable/Disable Input	This input is used to enable/disable the output of the driver
5	**Orange**	Direction Input	This input is used to change the rotation direction of the motor
6	**Green**	Power Ground	The ground or return of power supply connects here.
7	**White**	Logic Ground	Used to ground to the logic functions (i.e. step resolution)
8	**Blue**	Opto Supply (Input)	+5 VDC input used to supply power to the isolated logic inputs**
9	**Yellow**	Step Clock	Connects to the open-collector drive.

Table 1: Pin Assignments

WARNING! - Do not apply differential voltage on SR1 and SR2, this will damage the driver.

** The Resistors shall be connected in series with each Input: Pin 4 (Disable), Pin 5 (Direction), and Pin 9 (Step) if you are to use an opto supply of more than +5VDC. See Section 7.

Refer to the wiring diagram:

How to Connect

To connect the R208 according to *Figure 4*, begin by connecting your step motor to the R208, for details on how to make that connection refer to *Section 8 – Motor Connections*. Next, connect the DB-9 mating connector to the R208.

If using a function generator, take the positive end and connect it to Pin 9 (Step) and connect the negative end of the generator and connect it to the negative of the +5 VDC power supply. Take the positive end of the +5 VDC power supply and connect it to Pin 8, the Opto Supply. (Refer to *Table 6* for Opto supply resistor values)

Finally, connect the external +12 to +24 VDC power supply. Connect Pin 1 (Power) to the positive terminal of the power supply. Connect Pin 6 (Power Ground) to the negative terminal of the power supply.

Remember that the default Current is set to 1.4 Amp peak. **BE SURE NOT TO BURN YOUR STEP MOTOR WITH THIS DEFAULT CURRENT SETTING**. In order to change this value, please see the section above called "Adjusting the Output Current".

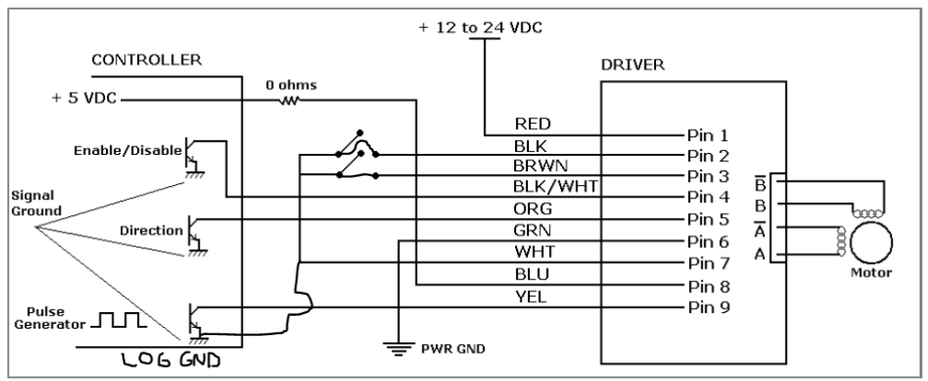
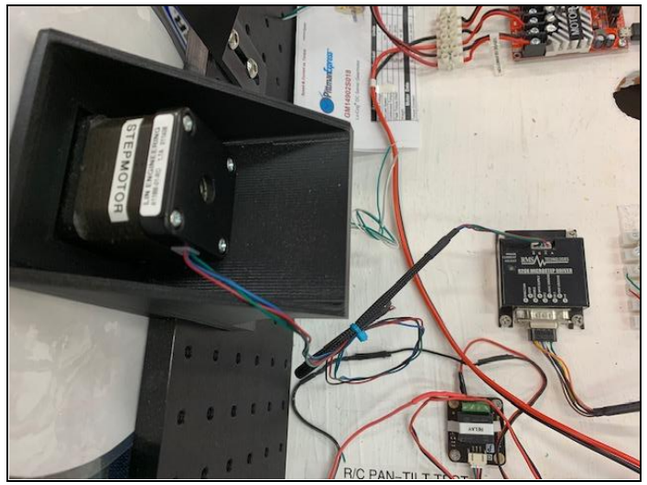


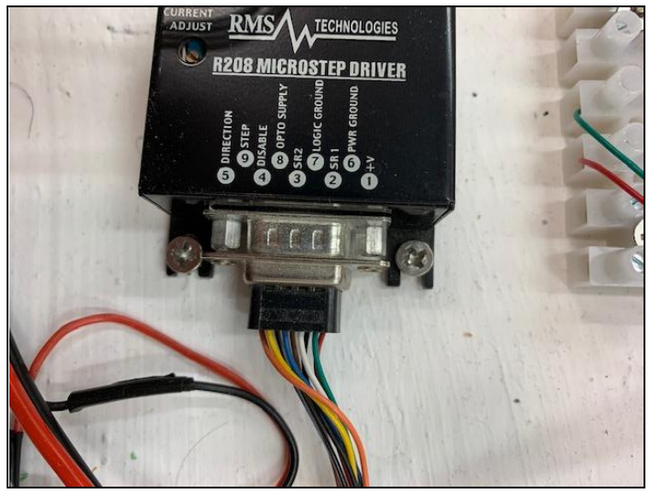
Figure 4: Connections Diagram

WARNING! Avoid touching Pin 1 (+12 to 24 VDC) to Pins 2, 3, 4, or 5. These four pins are connected to optocouplers which can only handle a few milliamps of current.

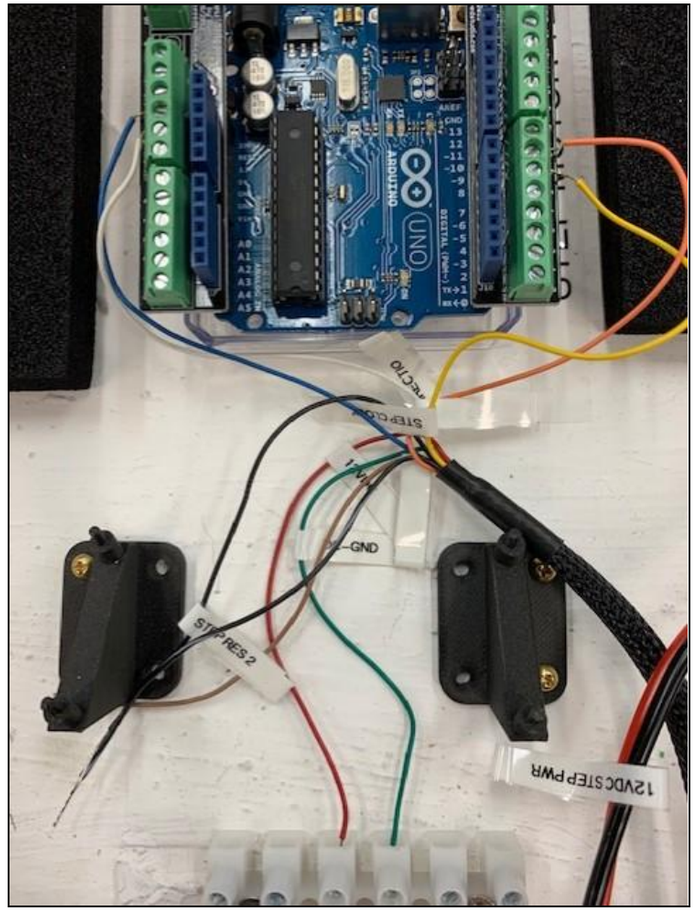
In practice, the stepper motor plugs directly into its controller with the premade factory cable on one end of the R208:



One the other end is a RS-232 connector that has a set of pigtail wires you must connect to your Arduino.



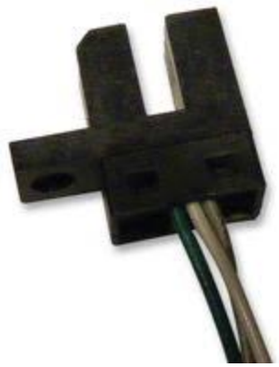
To wire up your Arduino to the stepper motor, please carefully connect as shown below:



For your first hookup, unplug Stepper Fuse. Please hook red and green wires (12VDC and Gnd) to the provided 12VDC power strip (picture bottom). Next hook up the blue (5VDC) and white,black,and brown wires (LogicGnd) to your Arduino. Last, hook up the yellow and orange wires (step and direction) to Arduino pins 6 and 7.

For your first pass through, leave the remaining wires disconnected and not touching anything.
You can hook up and enable the remaining wires if you want to pursue more sophisticated control of your stepper and be able to change step size or enable and disable the driver from your MATLAB program. You can reinsert the fuse and then move on to programming.

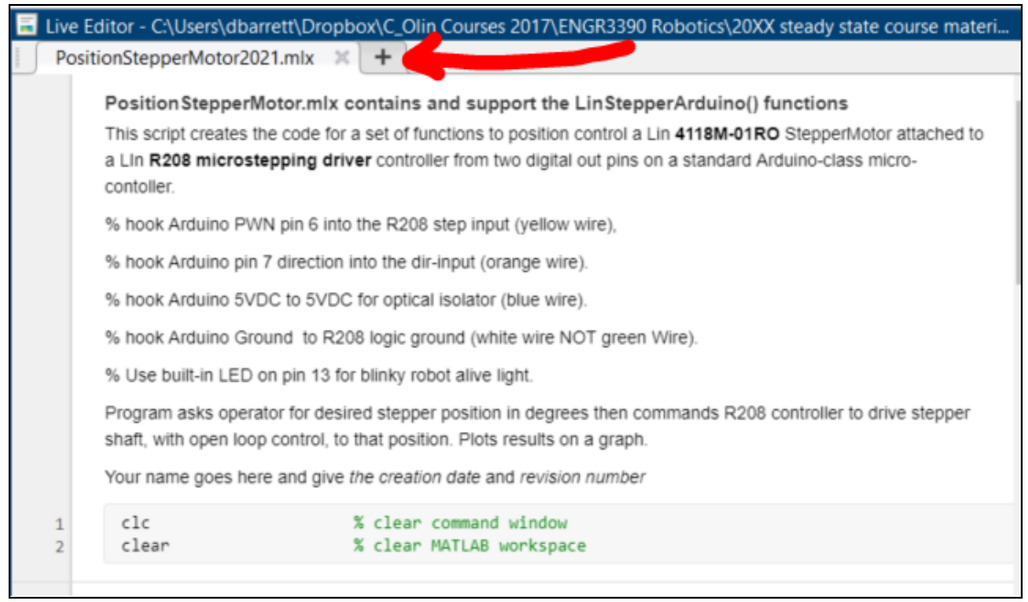
The optical switch
As an optional stretch task, you can hook up the provided optical interruption sensor to provide a fixed home location for your motor. A QVE00112 optical switch has been added to the lab breadboard to support this, see DC servo section for details.



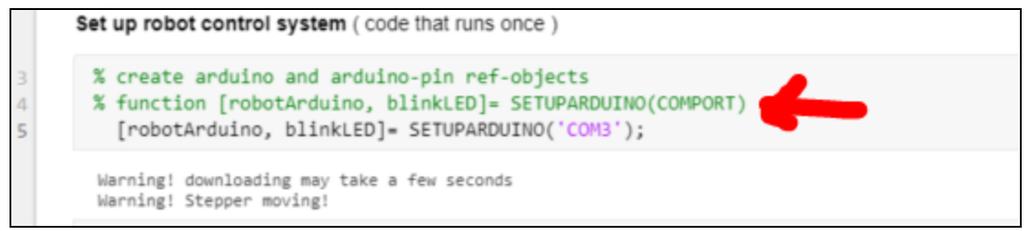
The rotating fish has been given a short tail extension, which when passing through the optical switch, trips it, and this information can be used as an absolute positional feedback mechanism to a digital input pin on your Arduino. Its technical specs can be found here:

https://drive.google.com/open?id=1w6cUEUsaXumfEMAyTmBsgexzAoAyX3Qm

Stepper Motor Test Code
Please make a copy of your Arduino robot template, rename it PositionStepperMotor.mlx and modify the initial comments to be as show:



Next edit the first part of the code that runs once to include a new function to do all of the configuration of the Arduino:



Go down to the general function section and modify the **SETUPARDUINO** function 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 ar
bulk of the work should be done by functions

```
function [robotArduino, 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');
   robotArduino = arduino(COMPORT,'Uno','Libraries', 'I2C');
   disp('Warning! downloading may take a few seconds ');

% configure pin 13 as a digitial-out LED
   blinkLED = 'D13';
   configurePin(robotArduino,blinkLED,'DigitalOutput');
% configure pin 6 as a digitial-out for stepper-motor step pulse
  stepPin = 'D6';
  configurePin(robotArduino,stepPin,'DigitalOutput');
% configure pin 7 as a digitial-out for stepper-motor direction
  dirPin = 'D7';
  configurePin(robotArduino,dirPin,'DigitalOutput');

% Turn on board LED on and off 3X to signal program has started
  Blink(robotArduino,blinkLED,3);
  disp('Warning! Stepper moving! ');
  pause(2.0);
end

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);
```

The Blink function can stay as it was. We will reuse it over and over in this lab.

Next we will waggle dance the stepper to confirm that it is hooked up correctly and operational before writing code to command it to a position.

```
% Move Stepper-Motor through 90 degrees (I'm alive waggle dance)
% ACTmoveStepMotor(arduinoName,numberOfSteps,direction, speed)
% doing an actuator check on startup, helps you avoid hardware problems
% when developing control code
  pause(1.0);
  beep;
  % ACTmoveStepMotor(arduinoName, numberOfSteps, direction, speed)
  ACTmoveStepMotor(robotArduino,50,0,0.01)
  Blink(robotArduino,blinkLED,1);
  beep;
  ACTmoveStepMotor(robotArduino,50,1,0.01)
  Blink(robotArduino,blinkLED,1);
  beep;
```

Please go to ACT function zone and add the **ACTmoveStepMotor** function as show:

Act Functions (store all Act related local functions here)

```
function [] = ACTmoveStepMotor(arduinoName, numberOfSteps, direction, speed)
  % moveStepMotor will rotate stepper motor by numberOfSteps in direction,
  % with a pause of speed (in seconds) between steps
  % arduinoName is the name of your Arduino
  % numberOfSteps is (0-200) direction is (0-clockwise)(1-counterclockwise)
  % no output is returned
  % dbarrett 2/7/20

  % configure pin 6 as a digitial-out for stepper-motor step pulse
  stepPin = 'D6';
  % configure pin 7 as a digitial-out for stepper-motor direction
  dirPin  = 'D7';

  % set direction by writing 1 or 0 to dir pin on Arduino
     writeDigitalPin(arduinoName, dirPin, direction);
  % run stepper for number of steps required at speed requested
     for mIndex = 1:numberOfSteps
     writeDigitalPin(arduinoName, stepPin, 0);
     pause(speed);
     writeDigitalPin(arduinoName, stepPin, 1);
     pause(speed);
     end
  end       % end of function
```

This function controls the stepper motor actuator by commanding the number of steps the motor takes over the 'D6' step wire to the R208 controller and setting the direction of those steps over the 'D7' direction wire. Stepper motors are fully digital and are quite easy to control from an Arduino or Raspberry Pi with just two control wires.

The remainder of the code that runs once,stays unchanged. It asks for how many experimental tests you wish to perform and then creates a matrix to store your experimentally derived data in.

```
20  % Configure test loop to collect n data points.
21  nTests = input(['Enter number of servo test positions, ' ...
22      'followed by enter: ']);
23
24  % create a variable to hold experimental position data
25    positionData = zeros(nTests,2);
26
27  r = rateControl(0.25);                % create a 0.25 hz loop rate
28  reset(r);                             % reset loop time to zero
```

The main control loop just requires a few small edits:

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

```
29  controlFlag = 1;                % create a loop control
30  oldAngle = 0;                   % create  variable to hold last position of motor (degrees)
31  while (controlFlag < nTests+1)  % loop till ntests data captured
32
33      commandAngle = SENSE();                          % collect deired position from operator
34      [steps, direction]= THINK(oldAngle, commandAngle);  % compute what robot should do next
35      % command servo motor to desired position
36      % ACTmoveStepMotor(arduinoName, numberOfSteps, direction, speed)
37      ACTmoveStepMotor(robotArduino, steps, direction, 0.1);
38      oldAngle = commandAngle;                         % update servo shaft position
39
40      % store experimental commanded versus actual data
41      positionData(controlFlag,1)= input('Enter actual position degrees: ');
42      positionData(controlFlag,2)= commandAngle;
43      Blink(robotArduino,blinkLED,1);
44      waitfor(r);                 % wait for loop cycle to complete
45      controlFlag = controlFlag+1; % increment loop
46  end
```

First we will add a motor shaft angle variable called **oldAngle** to store the last position the stepper shaft was in.

The SENSE function will remain unchanged and asks for user command input:

Sense Functions (store all Sense related local functions here)

```
104  function commandAngle = SENSE()
105    commandAngle = input('Enter desired servo angle in degrees:  ');
106  end
```

The THINK function calculator the difference between the steppers exiting position **oldAngle** and the desired new position **commandAngle** and calculates the number of steps (200 per revolution) and the direction (clockwise or counterclockwise) to get there:

Think Functions (store all Think related local functions here)

```
107  function [steps, direction]= THINK(oldAngle, commandAngle)
108     % THINK takes in the exisiting angular location of the stepper, its new
109     % commanded position, both in degreees,and then calculates the number of steps
110     % and the direction to get there
111     differenceAngle = oldAngle - commandAngle;
112     steps = abs(differenceAngle*0.55555);        % convert 200steps/360deg = 0.5555 s/d
113     if (commandAngle >= oldAngle)
114         direction = 1;                          % run stepper counterclockwise
115     else
116         direction = 0;                          % run stepper clockwise
117     end
118  end
```

Then the new **ACTmoveStepMotor** function drives the motor to the new commanded position.Once there it updates the position of the motor by loading the **commandAngle** into **oldAngle** from where it repeats the whole process for the number of tests you imputed. We originally have the step speed set really slow so you can see and hear each step, feel free to crank it up when you want to go faster.

Running the program, you should get:

```
Command Window
  Warning! Stepper motor live!
  Warning! Stepper moving!
  Enter number of test positions:  5
  Enter desired stepper angle in degrees:  0
  Enter actual stepper angle in degs 0
  Enter desired stepper angle in degrees:  30
  Enter actual stepper angle in degs 30
  Enter desired stepper angle in degrees:  60
  Enter actual stepper angle in degs 60
  Enter desired stepper angle in degrees:  90
fx Enter actual stepper angle in degs 90
```

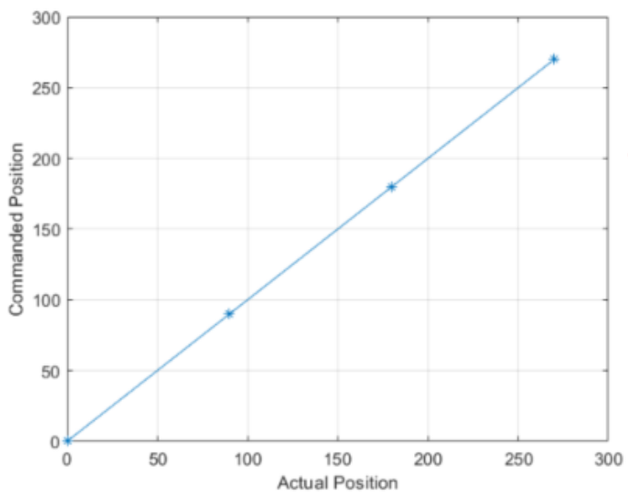
And your final plot like this:

Mission data processing

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

the mission. Here we will plot the desired versus actual motor positions.

```
47  % Plot and store commanded position data vs. actual position
48    plot(positionData(:,1), positionData(:,2), '-*')
49    xlabel('Actual Position')
50    ylabel ('Commanded Position')
51    grid
```




```
52    save positionServo.dat positionData -ascii
```

After the experimental run you just need to reset the stepper for next run and clean up your MATLAB workspace:

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.

```
53  % Stop program and clean up the connection to Arduino
54  % when no longer needed
55    [steps, direction]= THINK(oldAngle, 0);  % command stepper back to zero degrees
56      % command servo motor to desired position
57      % ACTmoveStepMotor(arduinoName, numberOfSteps, direction, speed)
58      ACTmoveStepMotor(robotArduino, steps, direction, 0.1);
59    Blink(robotArduino,blinkLED,3);
60    beep;
61    clc
62    disp('Arduino stepper program has ended');
```

```
Arduino stepper program has ended
```

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

Using the above code and calibration curves as a start, write a demo program that:

1) Show the motor cleanly going to 0-90-180-270-360-270-180-90-0.
2) Add in an optical interrupter switch to always start at the same zero position.
3) At a fast enough fixed speed you will lose steps at that first big jump up to flight speed, but if you ramp up and down on a trapezoidal profile, you can go much much faster without losing steps. Write a trapezoidal speed profile and see how fast you can stepper to go. See Instructor or Ninjas for help.

Stretch Goal Bonus points: add limit switch stuff

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

Two week demo: Please write this code, debug it, and be prepared to demo commanding your stepper motor to 4 locations, having it move to those locations and plotting out a clean performance curve showing the motor is under precise linear control.

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

Velocity Control: Dual DC drive wheels with RoboClaw control

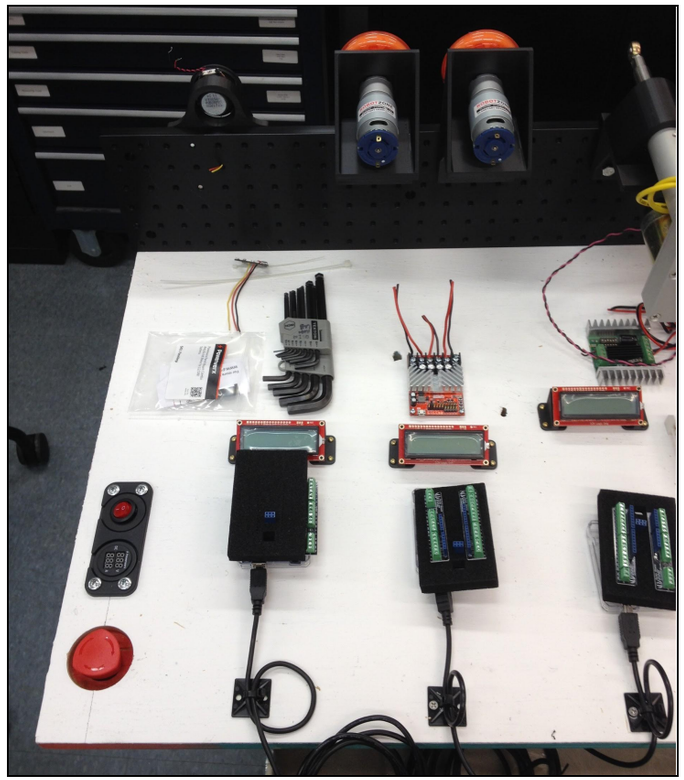
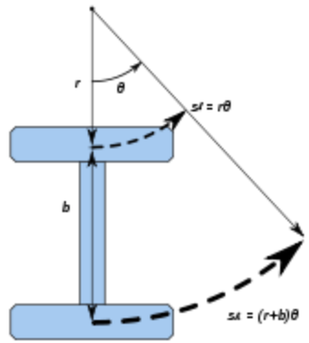


Figure: Dual DC velocity drive motors

In many robots there are two paired drive motors that provide propulsion. The propulsion motors are typically open loop, velocity controlled, DC-brushed gear motors attached to wheels or propellers and act as a pair to control robot motion. If both motors run at the same speed in the same direction, the robot moves forward. If both motors move in the opposite direction at the same speed, it will go in reverse. If both motors go at different speeds in either direction, the robot will travel in an arc whose radius is governed by the speed differential. If both motors turn at the same speed but in opposite directions, the robot will often just rotate in space.

For more detailed background, please see:

https://en.wikipedia.org/wiki/Differential_wheeled_robot



On the ACT test bench, there are two 12VDC Brushed DC gearmotors with representative wheels attached to them. The motors come from Servocity.com and are designated by serial number #638280:

https://www.servocity.com/313-rpm-hd-premium-planetary-gear-motor

Voltage (Nominal)	12V
Voltage Range (Recommended)	6V - 12V
Speed (No Load)*	313 rpm
Current (No Load)*	0.52A
Current (Stall)*	20A
Torque (Stall)*	416.6 oz-in (30 kgf-cm)
Gear Ratio	27:1

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

In order to work these motors, first it is important to choose how to communicate with them using the pre-mounted RoboClaw motor controller. This controller has plenty of options for controlling DC motors that range from RC to RC-Analogue to Serial controlled, but some are much more challenging than others. Starting with RC controls and treating the motors like continuous servos is the easiest way to go, but not the only one. The default mode without programming is the RC mode.

Parts

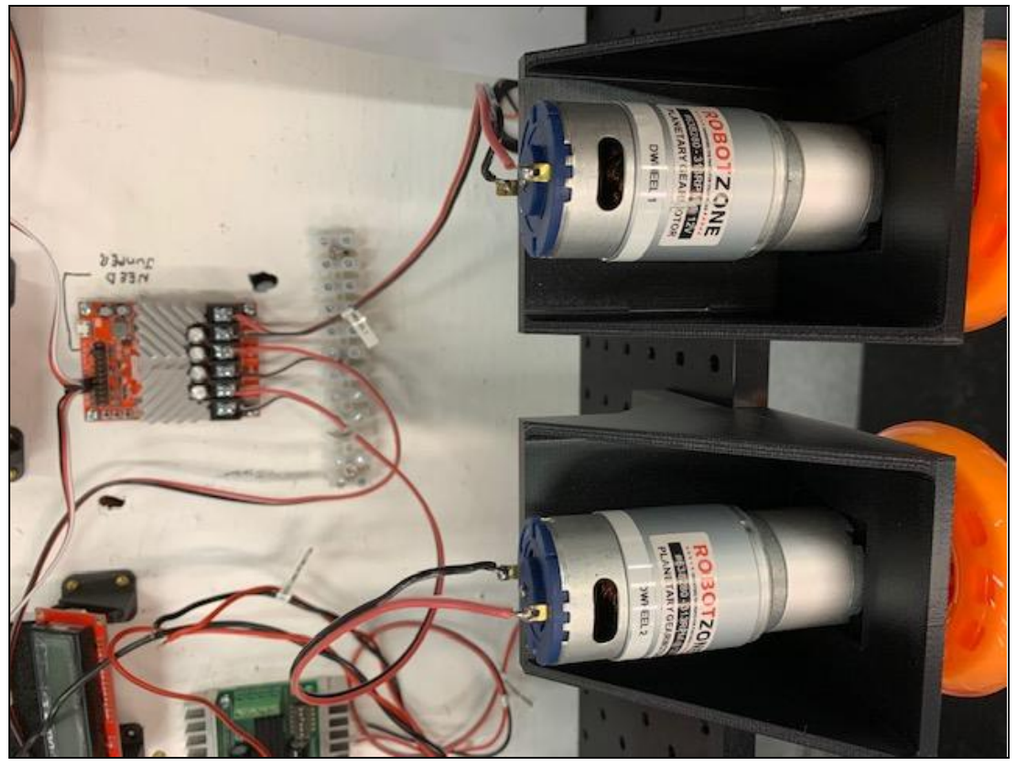
- RoboClaw 2x30A Motor Controller - basically the muscle of the whole operation. Arduino's are too small to output the kind of power needed to run these motors, but the motor controller bypasses this problem. Look here for documentation, especially on how to set modes:
  https://www.basicmicro.com/RoboClaw-2x15A-Motor-Controller_p_10.html
  http://downloads.ionmc.com/docs/roboclaw_user_manual.pdf
- 2 DC planetary gear motors RobotZone#638280 datasheet in canvas
- Arduino or Raspberry Pi - microcontroller that acts as the robot control for the whole system.

Tips

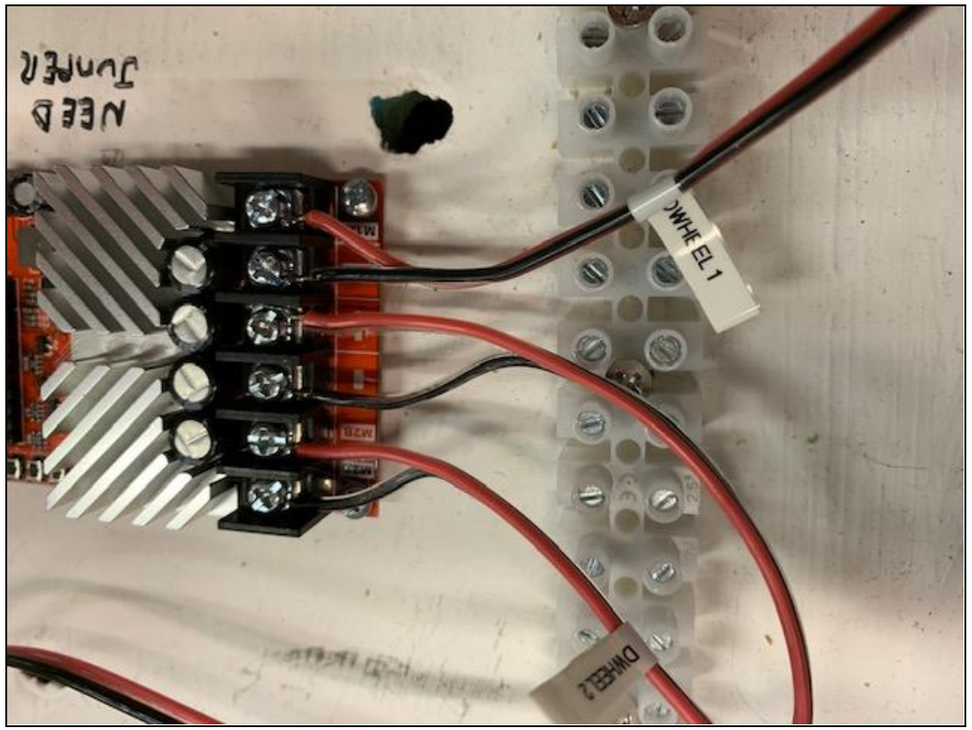
- The RoboClaw operates on 6-34V. This means that the logic only runs in this range.When you use PWM to hook up to the Arduino (5V), you should use the signal wire, and the ground wire but you should *not* connect the hot 5VDC wire. The motor controller will give you an error for a low logic battery if you do so. Also make sure to put the jumper between LB-MB to tell your motor controller that the logic battery should be linked to the main battery!
- Make sure to explicitly declare a stop motor function in your MATLAB control code. Electrical components sometimes do fun and mysterious things at their zero points and in the case of these motors, there can be creep that happens at what should be their stopped setting.

Finally, having set goals and drawing out a detailed system/electrical diagram before moving into the actual tinkering will be extremely useful! It's tempting to try to skip ahead to programming, but a perfect program will not work correctly if your wires are in the wrong places. **First please power off RoboClaw and arduino!**

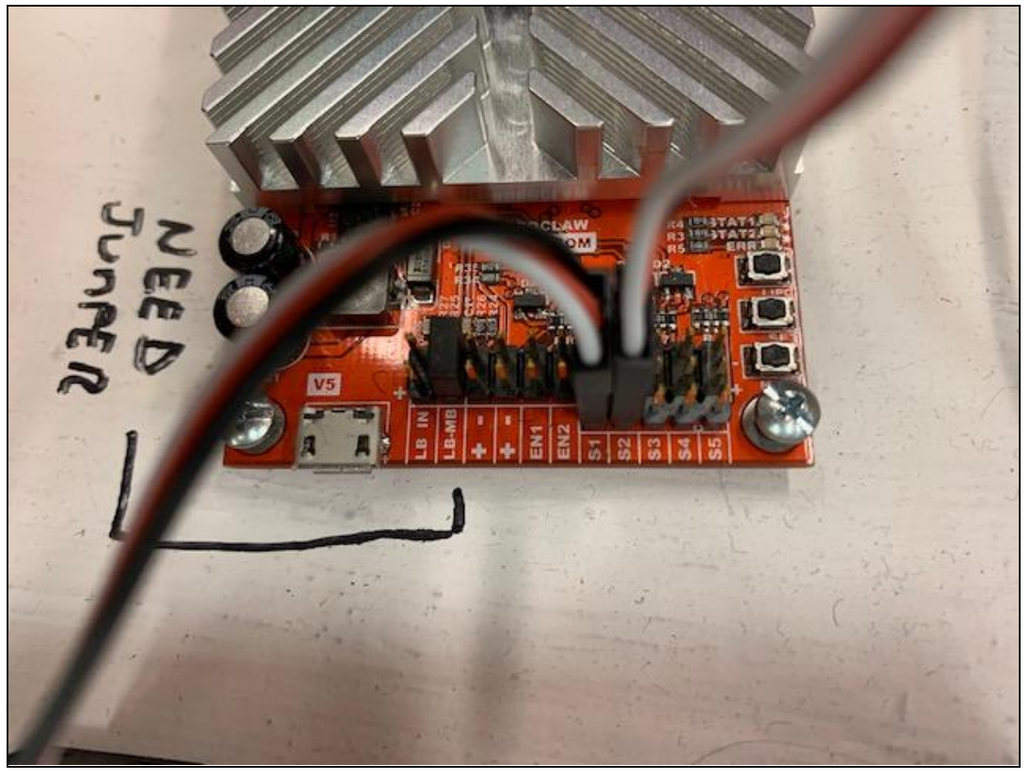
Then to wire the motors first connect each motor and the main motor power (from the motor fuse panel to the terminal strip provided as shown in the photograph below:



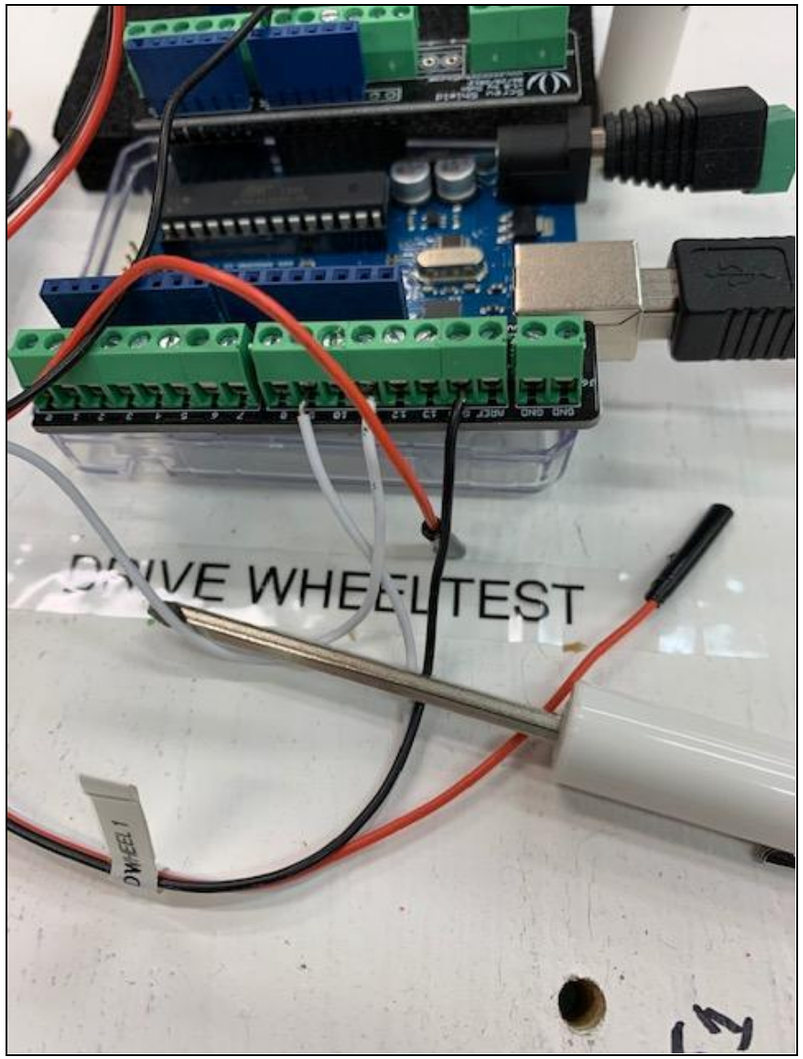
Seen at a bit larger magnification:



Next wire the provided red-white-black servo signal cables as shown below. We will only use the white wire (to carry the PWM signal) and the black wire (to carry reference ground). The red wires (5VDC power) are taped off as the RoboClaw is powered directly from the battery. Note: **Please DON'T attach red wires to your Arduino, it will cause a DC power bus failure.**

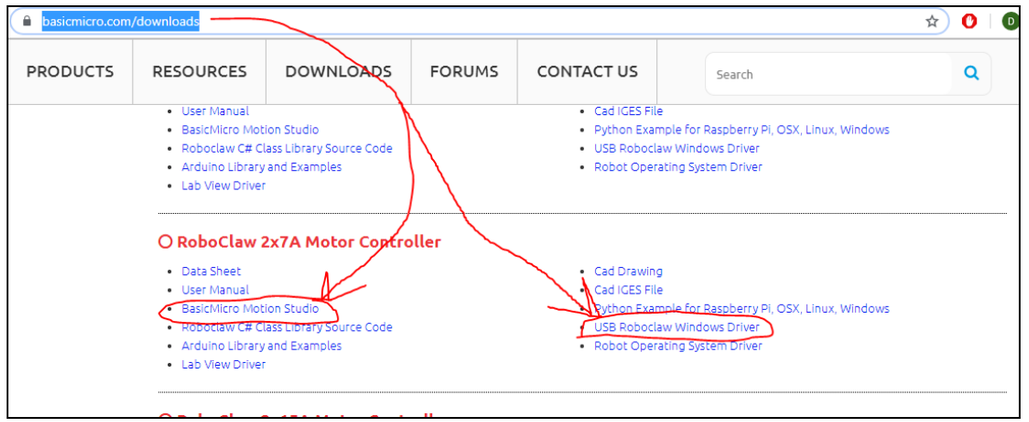


Next complete your wiring by attaching the black (ground) wire in each servo cable to an open ground pin on your Arduino and then attach Motor 1 to PWM pin 9 and Motor 2 to PWM pin 11. See picture below:



Step 1. Download RoboClaw Motion Control Studio

Please go to https://www.basicmicro.com/downloads and download the BasicMicro Motion Studio application. This will let you set up the PD control for and check out the encoder on your finger servos.



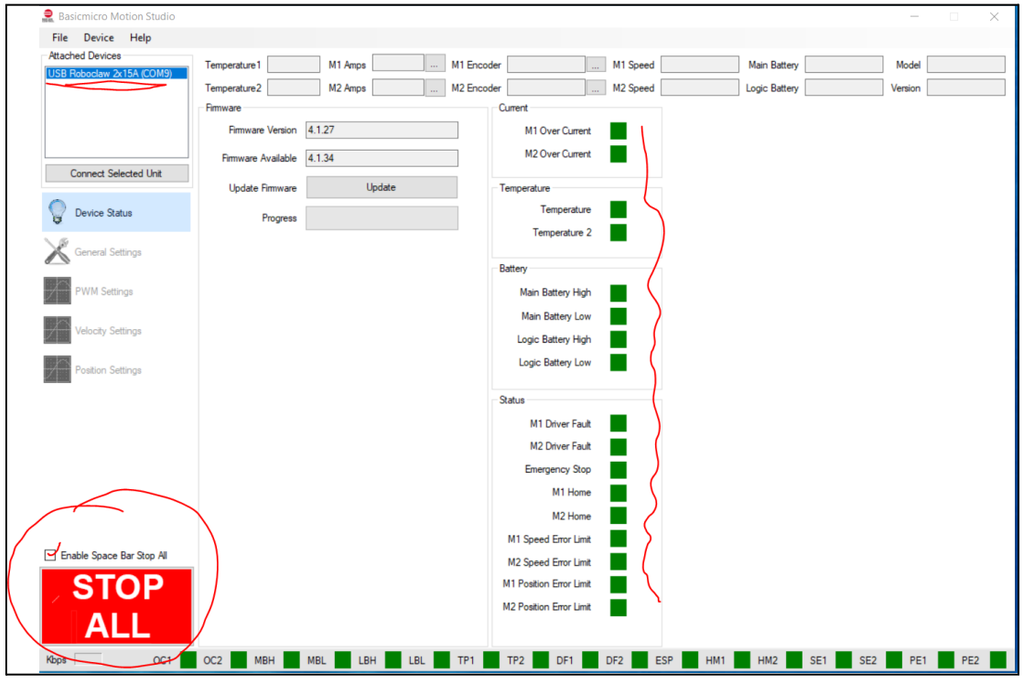
Step2. Download the USB Roboclaw Windows Driver

Please download the USB windows driver from the same location, your Laptop will need this driver to communicate with the mico-controller on the Roboclaw. After downloading, install driver. See course instructors for help if needed.

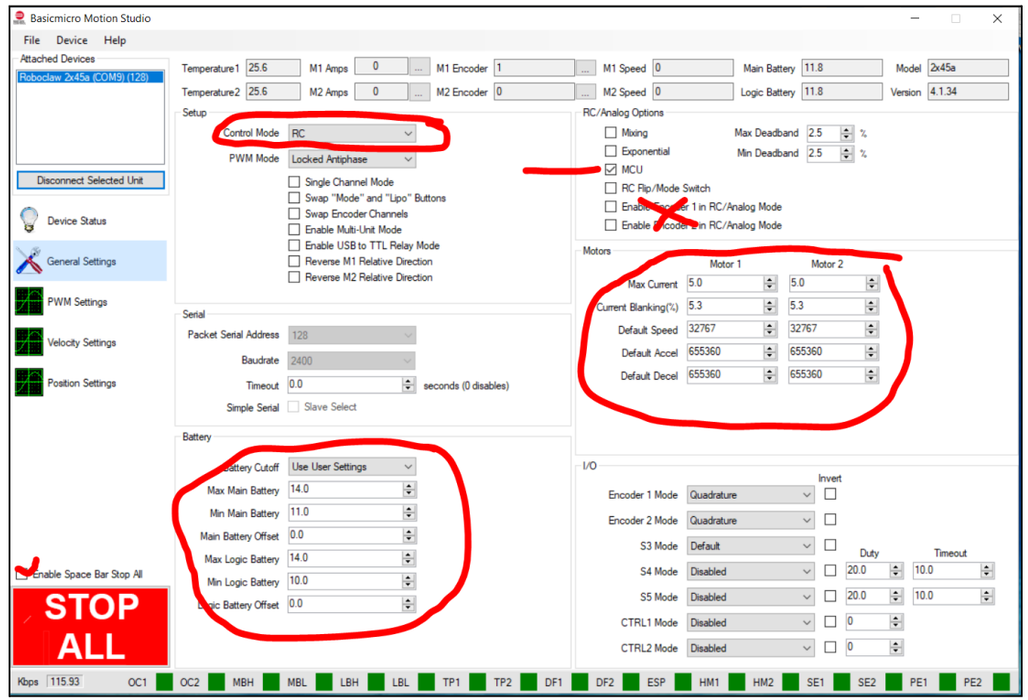
Step3. Setup and Tune RoboClaw for velocity control

The BasicMicro Motion Studio has extensive on-line documentation support. You can find it here:
https://resources.basicmicro.com/layout-and-common-operations-in-motion-studio/

Open up the **Motion Control** Application, plug the provided mini-USB cable into you laptop, select your Roboclaw and click on **Connect Selected Unit** to get:



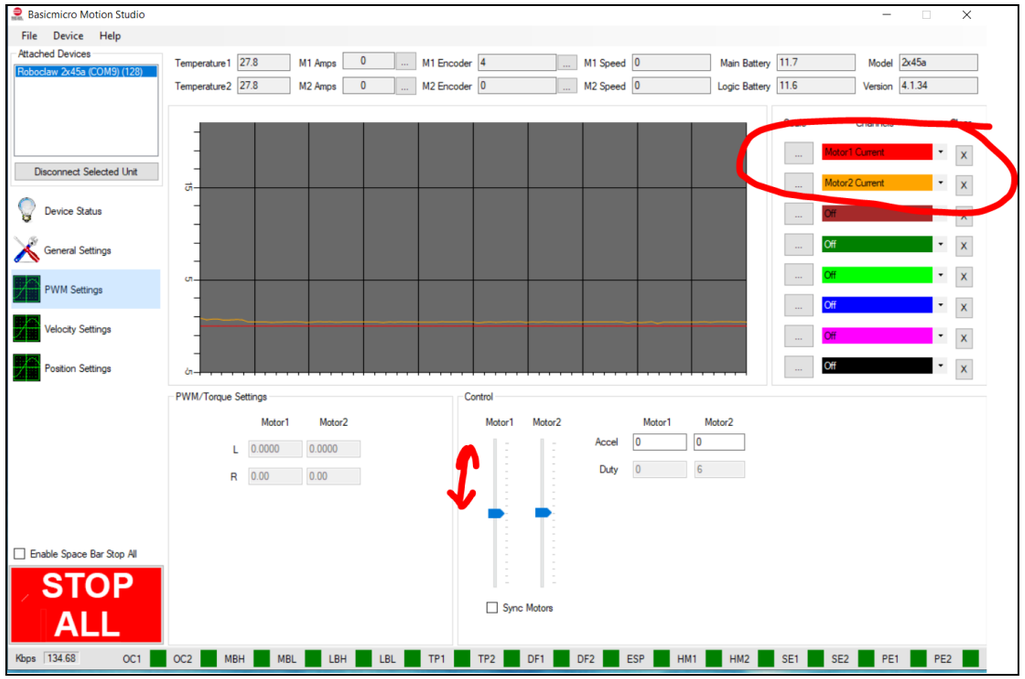
Set the **Enable Space Bar Stop All** check box. This will be your **EmergencyStop**.Then click on the left hand **General Settings** and set the **Control Mode** to **RC,** Set the **Battery voltages, Motor Speeds** and **Max Current** as shown(if you want to really know how to do this type of work independently and what all these control settings mean, please take a look at their giant hundred page **User Manual** that you can get to online under the **Help** pulldown menu of the **Basic Motion Studio Application**).



Please select **File>Save Settings** and save this configuration somewhere on your laptop, then select **Device>Write Settings** to download them to the RoboClaw. This test stand's motors don't have encoders, so there is no need to set them up.

Next, you will want to make sure motors are properly hooked up and turn in the right direction when commanded, so go to the **PWM Settings** tab and gently move the sliders up and down to move your motors.

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C



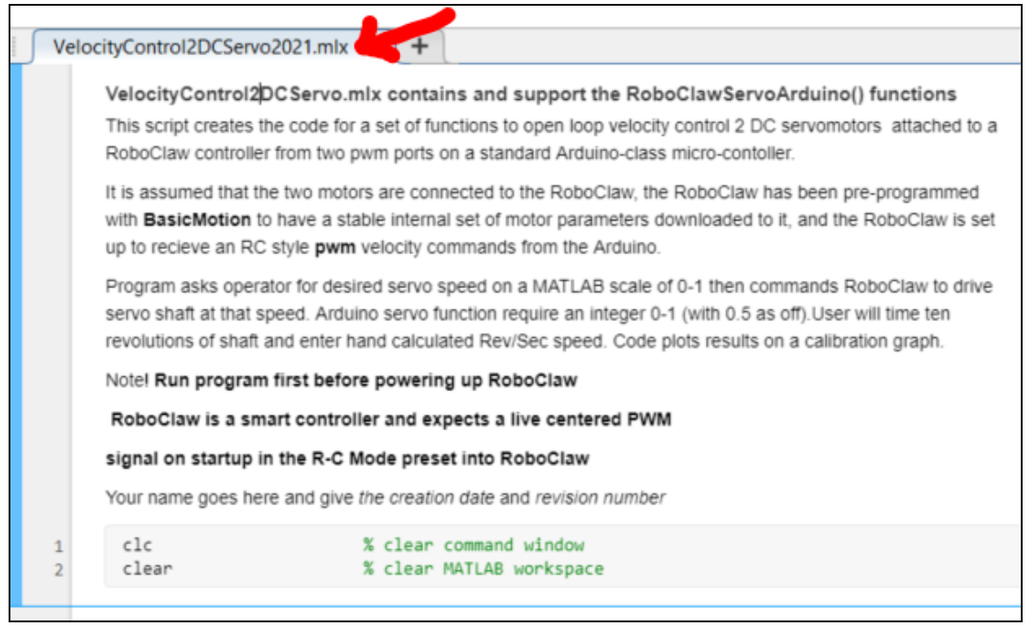
Start in on writing your code

Once the motors and controllers are wired correctly, it is possible to start writing your MATLAB code for the motors and getting a better sense as to how to control them.

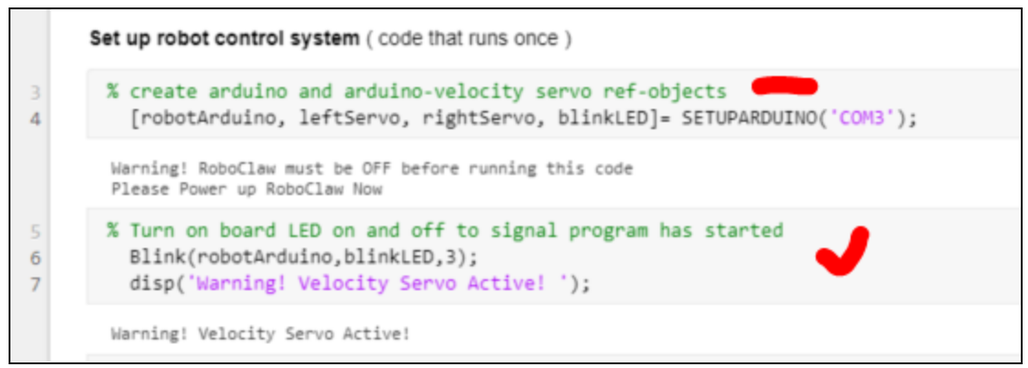
For this Two DCServo Control program:

Please read through this section, then open up your robot control template, make a copy by using **Save>Save As,** then save it as a MATLAB script called **VelocityControl2DCTwoServos.mlx** in your group MATLABDrive folder and then edit it section by section, single step testing each section as you go until it matches the code shown over the next few pages. Its a medium length program, but please do it well as you will reuse 90% of it for your Raspberry Pi code to follow.

Please update the first section of template to be as shown:



Next move the **Run Code Once** section and make the following edits:



The **Blink** function will stay the same, but you will need to redo the

SETUPARDUINO function to set up two PWM capable pins on Arduino to drive the two RoboClaw velocity servo motors, please edit 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, leftServo, rightServo, 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');
   disp('Warning! RoboClaw must be OFF before running this code ');

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

% create two servo objects driving PWM pin 9 and pin 11
% MinPulseDuration: 1120 (microseconds)  center 1520 (microseconds)
% MaxPulseDuration: 1920 (microseconds)
  leftServo = servo(robotArduino, 'D9', 'MinPulseDuration', 10*10^-6, ...
      'MaxPulseDuration', 1925*10^-6);
  rightServo = servo(robotArduino, 'D11', 'MinPulseDuration', 10*10^-6, ...
      'MaxPulseDuration', 1925*10^-6);

% RoboClaw in R-C Mode expects to start up with joystick centered
% In MATLAB function servo position is 0-1 so 0.5 (1520ms) is centered
  writePosition(leftServo, 0.5); % always start servo-command at 0.5
  writePosition(leftServo, 0.5); % always start servo-command at 0.5
  pause(5.0);                    % wait for Arduino to send stable pwm
  disp('Please Power up RoboClaw Now ');
  pause(2.0);
end
```

This code creates the Arduino object, creates two pwm servo outputs on pin 9 and pin 11, and then centers the output of both so that motors will not jump when you turn on RoboClaw servo amplifier.

Returning to the code that runs once section, please update the code to let you capture experimental data from both motors as shown:

```
Warning! Velocity Servo Active!

% Configure test loop to collect n data points.

nTests = input(['Enter number of servo test velocities, ' ...
    'followed by enter: ']);

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

r = rateControl(0.25);              % create a 0.25 hz loop rate
reset(r);                           % reset loop time to zero
```

The main robot control loop is short and direct, please modify your template to look like this:

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

    commandSpeed = SENSE();     % collect deired position from operator
    THINK();                    % compute what robot should do next
    % command both servo motors to desired velocity
    % function ACTRoboClawServoArduino(leftServo, rightServo, leftVelocity, rightVelocity)
    ACTRoboClawServoArduino(leftServo, rightServo,commandSpeed, commandSpeed);

    % store experimental commanded versus actual data
    leftVelocityData(controlFlag,1)= input('Enter actual left velocity (Rev/Sec: ');
    leftVelocityData(controlFlag,2)= commandSpeed;
    rightVelocityData(controlFlag,1)= input('Enter actual right velocity (Rev/Sec: ');
    rightVelocityData(controlFlag,2)= commandSpeed;
    Blink(robotArduino,blinkLED,1);
    waitfor(r);                 % wait for loop cycle to complete
    controlFlag = controlFlag+1; % increment loop
end
```

We will extend the Sense-Think-Act functions, one-by-one below:

Move down to the function part of your code and modify the **SENSE** and **THINK** function to look like this:

```
Sense Functions (store all Sense related local functions here)

function commandVelocity = SENSE()
  commandVelocity = input('Enter desired velocity scale(0-1):  ');
end

Think Functions (store all Think related local functions here)

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

Act Functions (store all Act related local functions here)
```

The **SENSE** function just asks for operator input as to what speed is desired. The **THINK** function is just a placeholder that doesn't do anything. While you are developing code, placeholder functions are a good way of preserving code flow and structure, without needing to code that part just yet.Moving on to the **ACT** function:

```
Act Functions (store all Act related local functions here)

function ACTRoboClawServoArduino(leftServo, rightServo, leftVelocity, rightVelocity)
% is an example of a function to control a RoboClaw DC motor
% controller in open-loop, velocity servo mode.
% Need to run SETUPARDUINO function first to create arduino and
% velocity servo objects.
% hook Arduino PWN pin 9 into Roboclaw digital input S1, hook Arduino
% Ground to RoboClawSI Ground.
% Input is (desired velocity on MATLAB scale (0-1)
% Output is motion of servo motor shaft
% dbarrett 1-13-20  RevC

    % scale desired angle to 0-1 for writeposition Arduino command
    % to Deg/sec once you have created calibration curve
    % scaleComAngle = commandAngle/(Deg/sec);

    writePosition(leftServo, leftVelocity);
    writePosition(rightServo, rightVelocity);

end
```

This is the actual function that you can harvest for use in a later robot toolbox. It has a lot of text to make it useful for when you do. The actual function is pretty simple, it takes in the names of which servos to move and the desired velocities of each (velocity in MATLAB 0-1 scale) and then drives them at that velocity. Normally after you have carefully calibrated the actual wheel speed, via the data you collect with this code, with what speeds the 0-1 command produces, you'd go back and alter this function to take in wheel speed in velocity units like meters per second or inches per second. In all robot code it is good to always stay in engineering units that can be measured and stay away from magic numbers like speed from 0 to 1.

Having created your functions, go back and look at 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

    commandSpeed = SENSE();     % collect deired position from operator
    THINK();                    % compute what robot should do next
    % command both servo motors to desired velocity
    % function ACTRoboClawServoArduino(leftServo, rightServo, leftVelocity, rightVelocity)
    ACTRoboClawServoArduino(leftServo, rightServo,commandSpeed, commandSpeed);

    % store experimental commanded versus actual data
    leftVelocityData(controlFlag,1)= input('Enter actual left velocity (Rev/Sec: ');
    leftVelocityData(controlFlag,2)= commandSpeed;
    rightVelocityData(controlFlag,1)= input('Enter actual right velocity (Rev/Sec: ');
    rightVelocityData(controlFlag,2)= commandSpeed;
    Blink(robotArduino,blinkLED,1);
    waitfor(r);                 % wait for loop cycle to complete
    controlFlag = controlFlag+1; % increment loop
end
```

SENSE asks the operator what speed to run the first experiment at. **THINK** does nothing dangerous. **ACT** runs both motors at that speed. ThThis whole process repeats for **nTest** iterations.

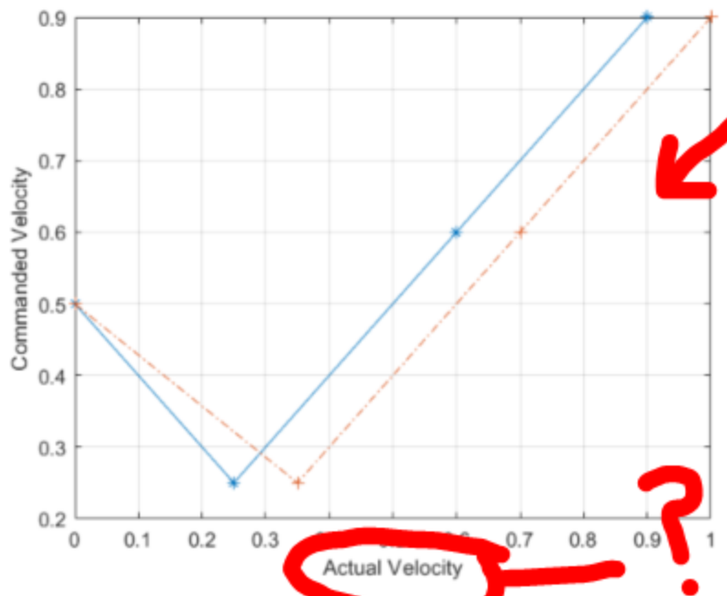
Once your pair-programing team has run their nTests, the following code will plot out a nice calibration graph for you. You will want to change the plot axes to match the real engineering units you took your data in.

VelocityControl2DCServo2021.mlx

Mission data processing

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

```
% Plot and store commanded velocity data vs. actual velocity
  plot(leftVelocityData(:,1), leftVelocityData(:,2), '-*')
  hold on;
  plot(rightVelocityData(:,1), rightVelocityData(:,2), '-.+')
  hold off;
  xlabel('Actual Velocity')
  ylabel ('Commanded Velocity')
  grid
```




```
% write experimental data to a file for later use
save leftServo.dat leftVelocityData -ascii
save rightServo.dat leftVelocityData -ascii
```

When you run this program it will ask you how many tests to do and then let you input desired speeds from 0 (full CW) to 0.5 (stopped) to 1.0 (full CCW) on each motor. As you do, time how long it takes each motor to do 10 revolutions and then enter the appropriate actual speed in rev/min.

```
Command Window
  Warning! RoboClaw must be OFF-unpowered before running this code
  Please Power up RoboClaw Now
  Warning! Position Servo Active!
  Enter number of test positions:  3
  Enter desired left motor speed 0-1  0.5
  Enter desired right motor speed 0-1 0.5
  Enter left motor speed (rev/sec) 0
fx Enter right motor speed (rev/sec) 0
```

At the end the test program will pop up a nice graph of commanded speed versus actual speed for both motors. See graph to the left.

The final code bit shuts the RoboClaw down gracefully and exits the program.

Clean shut down

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

```
% Stop program and clean up the connection to Arduino
% when no longer needed
  writePosition(leftServo, 0.5);      % always end servo at 0.5
  writePosition(rightServo, 0.5);      % always end servo at 0.5
  clc
  disp('Arduino program has ended');
```

```
Arduino program has ended
```

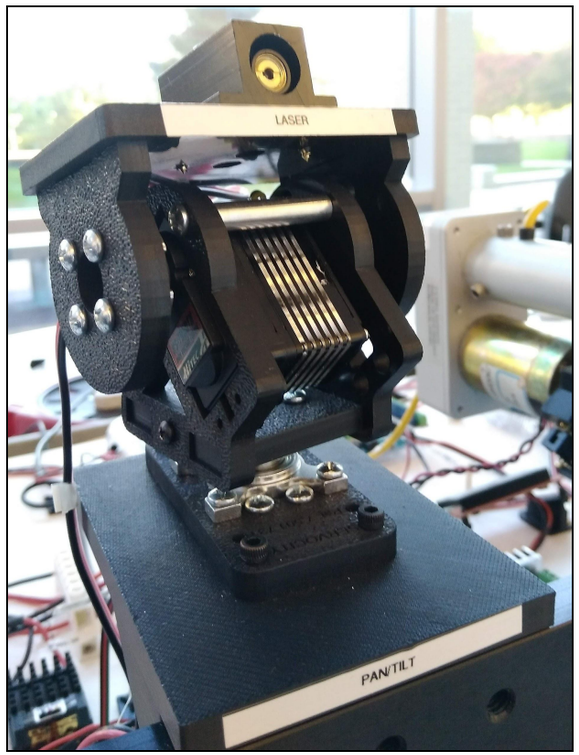
```
  clear robotArduino
  beep              % play sytem sound to let user know program is ended
```

Two week demo: Please write this code, debug it, and be prepared to demo commanding your DC velocity servo motors to 4 locations (full forward, full reverse, spin left and spin right), having it drive at those speeds and plotting out a clean performance curve showing the motor is under precise linear velocity control.

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

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

Position Control: Pan/ Tilt Laser Turret



INTRO

Welcome to RC Servo Pan-Tilt test fixture! Here we have a small two axis laser turret with two RC model servos and a laser. The first thing you're going to want to do is get some understanding of how this platform works (by platform, we mean the pan-tilt test apparatus). Here's a quick rundown of the parts involved

Parts

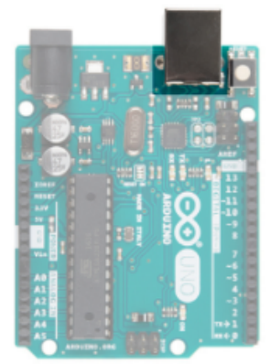
- AdaFruit 5V Laser - a laser with 5VDC power and ground
- SparkFun Motor Shield - this makes your life easier for servo control
- Power Relay - makes sure our laser doesn't turn on until we want it to
- Two Servo Motors - they are a open loop rotation position sourse
- Link Port Dc to DC converter - uses a transformer to turn 12V into 5V
- Arduino - a small robot microcontroller

Documentation on all of the above can be found in the Canvas Act Ia folder.

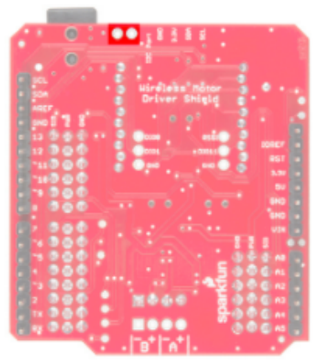
IMPORTANT:

You should put a piece of electrical tape on top of the USB connection since the metal can potentially short the motor driver board

> ϟ **Warning!** For those using the Arduino Uno, the USB female type B connector can short power where the **VS** pins are located. Make sure to add some electrical tape on top of the Arduino Uno's USB connector or bottom of the shield where the **VS** pins are exposed.



Add Electrical Tape where the Arduino Uno's USB Connector is Exposed on the Top



*Add Electrical Tape where the External **VS** Power Pins are Exposed on the Bottom*

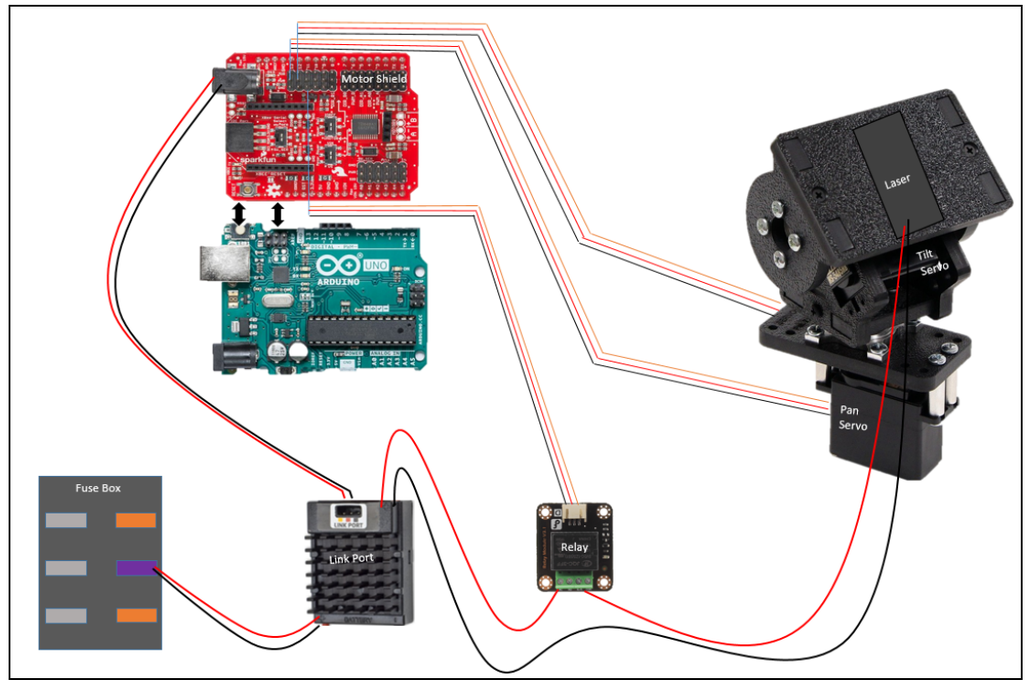
PLEASE READ THE MOTOR SHIELD QUICKSTART GUIDE! (Or, if you don't, keep in mind that it is really necessary to put a piece of electrical tape over the metal charging port of your arduino or you risk shorting everything).

https://learn.sparkfun.com/tutorials/wireless-motor-driver-shield-hookup-guide?_ga=2.253636815.793498985.1581258687-37032538.1477070748

You may not have to create the electrical system from scratch again, but nevertheless you should build a strong understanding of what's going on. Here's some guidance in the form of quick tips!

1. Everything you are working with wants 5V of power. The official documentation for our servos says we need more, but they work fine with 5V. This is what we need our link port Dc to Dc converter for. Our test bench setup only has 12VDC e-stoppable power.
2. Our laser should not be powered unless we tell our Arduino we want it powered. That's what our relay is for. Our relay has an input "COM" (common) where it takes an input power. It has two outputs, "NO" and "NC". "NO" is "normally open", meaning that power *won't* be output to it unless we flip our relay switch. "NC" is "normally closed", meaning that power *will* be output to it unless we flip our switch. Read the documentation and do some googling around for a more in-depth explanation on NO and NC.

What this will wind up looking like in practice: the pan servo's inputs (pwm pin 'D9') will be lined up with the dark wires matching up (grounds matching up) on the first row of pins on the right side and the tilt servos (pwm pin'D11') on the next row of pins matched up in the same way. Put the relay pins on the third row lined up in the same way. Make sure that the red and black wire for the link port are attached to the middle left side of the E-stop connector (when you're facing the side of the table that has the actual sensors on it) and that there is a connector inserted into the slot next to that connection.



The test setup will be mostly wired in place so that you can get up to speed quickly. You should make sure you understand the basic wiring and be able to describe it for your individual lab report.

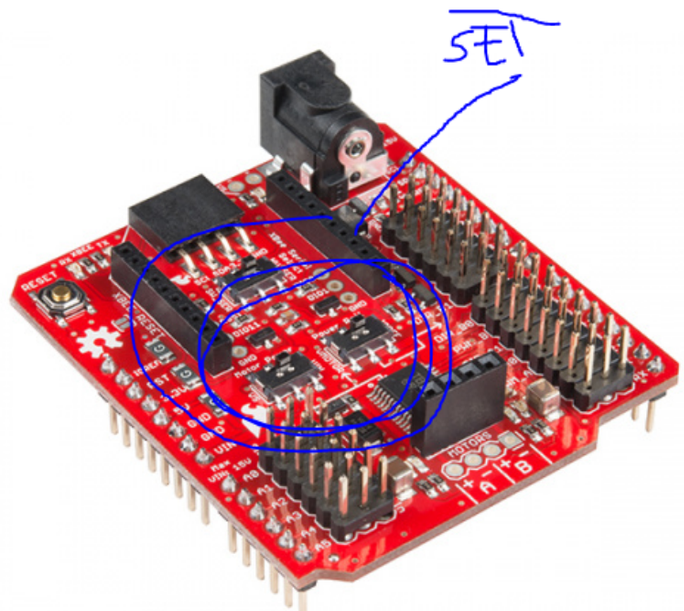
After plugging in your RC servo motor, please have an instructor or Ninja check your wiring before powering your test setup up for the first time. Also please don't stare directly into the laser, it's pretty eye-safe when used as directed, but not for staring.

Use the SparkFun DCPower Jack to power pan and tilt motors

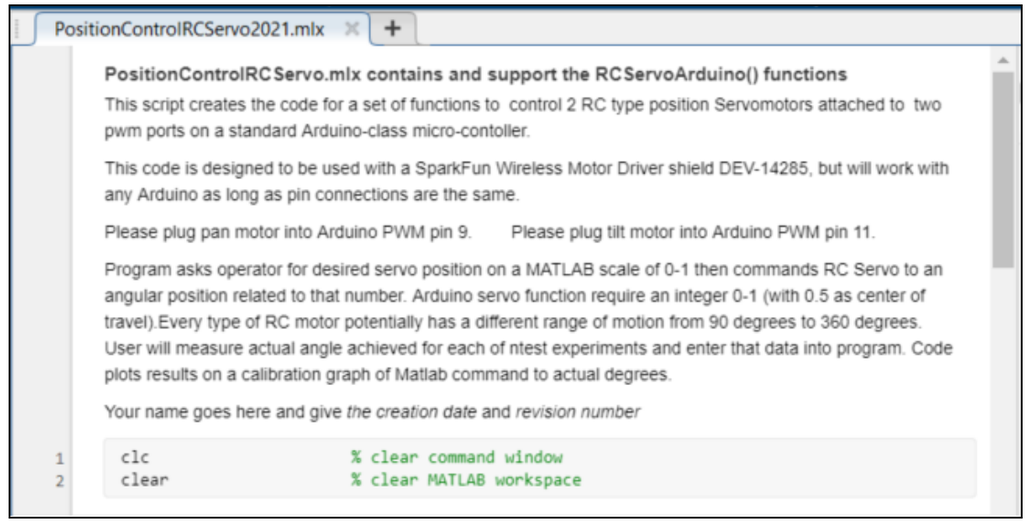
Often, driving motors from the Arduino's power supply (even VIN) will cause the voltage to dip and possibly reset your Arduino. To help with this issue, the power jack on the top-right of the Shield will accept a 5.5 x 2.1mm power plug from a variety of wall adapters and battery packs.

Note that you will need to **set the Motor Power switch to VS** to power the motors from the power jack. If you also set the **Power Rail switch to VMOTOR**, then the power jack will be connected to the PWR rail on the digital pins (e.g. to power servos). Both switches are in the center of the board as shown below.

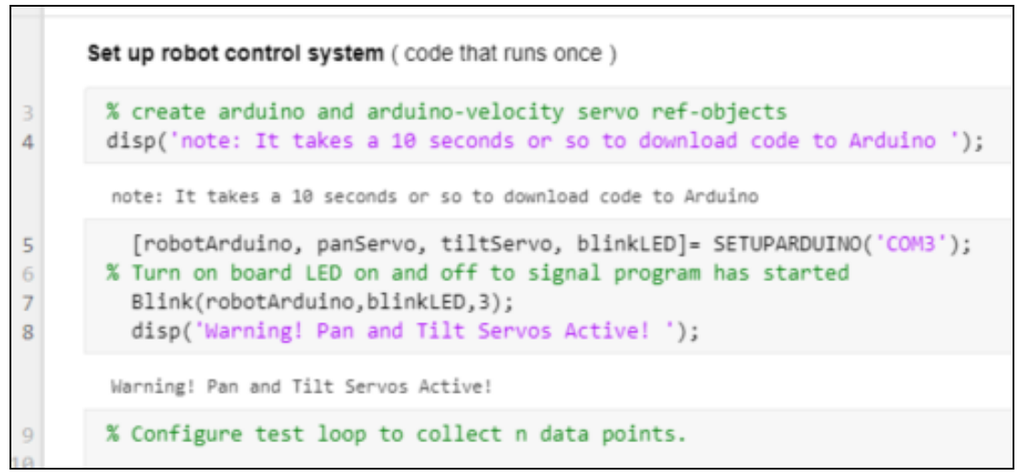
Additionally, this power jack will not power the Arduino. It is intended to provide a power supply to your motors separate from your Arduino.



Please start from your robot control template, save a new copy as a MATLAB file called PositionControlRCServo.mlx and modify the first section to look like:



You will next modify the **code that runs once** section as shown:



This code will call two functions, one to set up your Arduino, the second to make your Arduino LED blink. Please move down to the general function part of your code and modify the two functions to look like this:

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 ar the bulk of the work should be done by functions

```
function [robotArduino, panServo, tiltServo, 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');

% create two servo objects driving PWM pin 9 and pin 11
% MinPulseDuration: 1120 (microseconds)  center 1520 (microseconds)
% MaxPulseDuration: 1920 (microseconds)
  panServo = servo(robotArduino, 'D9', 'MinPulseDuration', 10*10^-6, ...
      'MaxPulseDuration', 1925*10^-6);
  tiltServo = servo(robotArduino, 'D11', 'MinPulseDuration', 10*10^-6, ...
      'MaxPulseDuration', 1925*10^-6);

% R-C Mode expects to start up with joystick centered
% In MATLAB function servo position is 0-1 so 0.5 (1520ms) is centered
  writePosition(panServo, 0.5); % always start servo-command at 0.5
  writePosition(tiltServo, 0.5); % always start servo-command at 0.5
  pause(5.0);                        % wait for Arduino to send stable pwm
end
```

This function creates an Arduino object, designates pin 13 as blinky LED, sets up pin 9 and pin 11 as PWM rC servo type outputs and then sends a centered command angle to both servomotors. Centering your servos is always a good practice when turning on any Robot system.

The Blinky light function **Blink**:

```
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
```

Just makes your LED blink a fixed number of times when commanded to do so.

The remainder of the code that runs once, asks how many tests you'd like to do and then sets up two variables to store your test data in. The reason behind this is Matlab will control your servos on a normalized scale of 0-1 and for clarity and future use, you would like to command them in actual degrees, with 0 degrees being the center position.

You will run a set of nTEsts, commanding your servos to go to 0, 0.2,0.4,0.5 etc. and with each test measure and record how far they move (with a protractor). This data will be automatically plotted on a calibration graph for you. Most RC servos have a fairly small range of motion, somewhere from 90 to 130 degrees. You need to find out what floating point number from 0-1 commands say 35 degrees.

With a solid set of calibration curves, you can rewrite the given RC servo function to take an input in degrees. This will be most useful in your final project and in all future systems you build with RC motors.

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

Moving on to the section of your code that runs in a continuous loop, please modify your template code's Sense-Think-Act control structure to look like this:

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

```
20  controlFlag = 1;                % create a loop control
21  while (controlFlag < nTests+1)  % loop till ntests data captured
22
23      commandAngle = SENSE();     % collect deired position from operator
24      panAngle = commandAngle;    % set panAngle to operator input
25      tiltAngle = commandAngle;   % set tilt Angle to operator input
26      THINK();                    % compute what robot should do next
27      % command both servo motors to desired position
28      % function ACTRCServoArduino(panServo, tiltServo, panAngle, tiltAngle)
29      ACTRCServoArduino(panServo, tiltServo, panAngle, tiltAngle) ;
30
31      % store experimental commanded versus actual data
32      panServoData(controlFlag,1)= input('Enter actual pan angle (degrees): ');
33      panServoData(controlFlag,2)= commandAngle;
34      tiltServoData(controlFlag,1)= input('Enter actual tilt angle (degrees): ');
35      tiltServoData(controlFlag,2)= commandAngle;
36      Blink(robotArduino,blinkLED,1);
37      waitfor(r);                 % wait for loop cycle to complete
38      controlFlag = controlFlag+1; % increment loop
39  end
```


This code runs the loop for **nTests**, storing the commanded and experimentally captured data in each iteration of the loop. Lets look at the functions used in each step.

Sense Functions (store all Sense related local functions here)

```
101   function commandAngle = SENSE()
102   commandAngle = input('Enter desired angle scale(0-1):  ');
103  end
```

Think Functions (store all Think related local functions here)

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

The **SENSE** function pauses the process and asks the operator to command the pan and tilt servos to a desired position form 0-1.

The **THINK** function is a null placeholder that doesn't do anything beyond fill the spot in the control loop.

The **ACT** function takes the command angle and drives both pan and tilt servo motors to those positions, please modify yours as shown below:

Act Functions (store all Act related local functions here)

```
107  function ACTRCServoArduino(panServo, tiltServo, panAngle, tiltAngle)
108  % is an example of a function to control a RoboClaw DC motor
109  % controller in open-loop, velocity servo mode.
110  % Need to run SETUPARDUINO function first to create arduino and
111  % velocity servo objects.
112  % hook Arduino PWN pin 9 into Roboclaw digital input S1, hook Arduino
113  % Ground to RoboClawSI Ground.
114  % Input is (desired velocity on MATLAB scale (0-1)
115  % Output is motion of servo motor shaft
116  % dbarrett 1-13-20  RevC
117
118      % scale desired angle to 0-1 for writeposition Arduino command
119      % to Deg/sec once you have created calibration curve
120      % scaleComAngle = commandAngle/(Deg/sec);
121
122      writePosition(panServo, panAngle);
123      writePosition(tiltServo, tiltAngle);
124
125  end
```

The **ACT** function is written in as general a fashion as possible so you can reuse it in other programs downstream. You would need to add a bit more code to make it more generally usable and have the input command pan and tilt angles be in degrees, and use internal calibration code to convert those to a normalized 0-1 range for the write Position function to work with.

ENGR3390: Fundamentals of Robotics Tutorial (ACT - Motors, lights and sound) 2023C

Returning to the control loop:

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

```
20  controlFlag = 1;                % create a loop control
21  while (controlFlag < nTests+1)  % loop till ntests data captured
22
23      commandAngle = SENSE();     % collect deired position from operator
24      panAngle = commandAngle;    % set panAngle to operator input
25      tiltAngle = commandAngle;   % set tilt Angle to operator input
26      THINK();                    % compute what robot should do next
27      % command both servo motors to desired position
28      % function ACTRCServoArduino(panServo, tiltServo, panAngle, tiltAngle)
29      ACTRCServoArduino(panServo, tiltServo, panAngle, tiltAngle) ;
30
31      % store experimental commanded versus actual data
32      panServoData(controlFlag,1)= input('Enter actual pan angle (degrees): ');
33      panServoData(controlFlag,2)= commandAngle;
34      tiltServoData(controlFlag,1)= input('Enter actual tilt angle (degrees): ');
35      tiltServoData(controlFlag,2)= commandAngle;
36      Blink(robotArduino,blinkLED,1);
37      waitfor(r);                 % wait for loop cycle to complete
38      controlFlag = controlFlag+1; % increment loop
39  end
```

The remaining code in it records and stores the manually observed angular position data for use in the next section of codes calibration plot. Moving on to that section, please modify as shown:

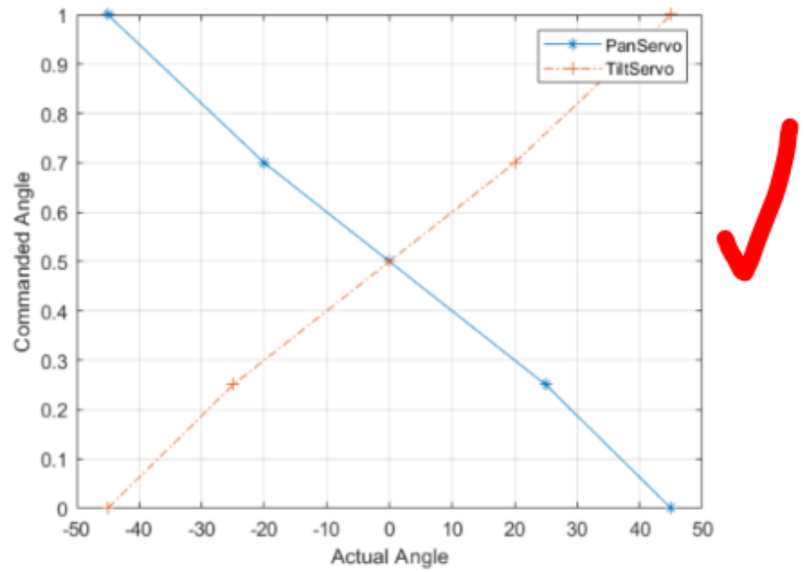
Mission data processing

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

```
40  % Plot and store commanded velocity data vs. actual velocity
41    plot(panServoData(:,1), panServoData(:,2), '-*')
42    hold on;
43    plot(tiltServoData(:,1), tiltServoData(:,2), '-.+')
44    hold off;
45    xlabel('Actual Angle')
46    ylabel ('Commanded Angle')
47    legend('PanServo','TiltServo')
48    grid
```

Which will generate a nice inline servo input versus output calibration plot:

```
47    legend('PanServo','TiltServo')
48    grid
```




```
49    % write experimental data to a file for later use
50    save panServoData.dat panServoData -ascii
51    save tiltServoData.dat tiltServoData -ascii
```

And store your data in a file for future use.

When you run this test program, the pan and tilt servo-motors will center once to make sure all is hooked up ok. The OCU interface will then ask the operator to enter the number of calibration tests required. It will ask for both commanded and actual servo positions reached:

```
Command Window
  Warning! Position Servo Active!
  Enter number of test positions:  3
  Enter desired pan angle in degrees:  40
  Enter actual pan angle: 40
  Enter desired pan angle in degrees:  50
  Enter actual pan angle: 50
  Enter desired pan angle in degrees:  100
  Enter actual pan angle: 100
  Enter desired tilt angle in degrees:  60
fx Enter actual tilt angle:
```

And give you a clean calibration graph.

Next you will want to make the laser-pan-tilt be able to move in straight horizontal and vertical lines. First add a bit of code to turn on and off the laser. Wire Laser relay into any digital output port on your Arduino.

Next, you will need to do a little mathematical heavy lifting to get the two rotational angles of the pan-tilt mapped into the positions needed to draw (more or less) straight horizontal and vertical lines on the wall opposite the test fixture.

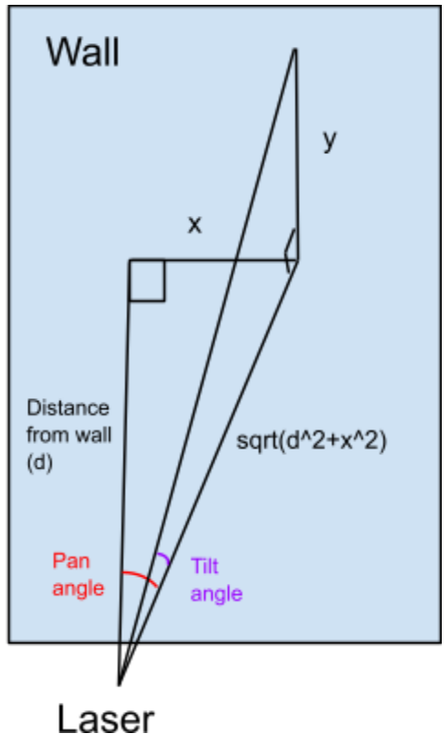
The initial positions of pan and tilt as well as the movement code for both in terms of translating x and y coordinates into angles will require translation from orthogonal to spherical coordinate frames. The ways these are calculated differs due to the different directions and relative positions of the servos.

Finding the pan angle is mostly a matter just of finding the angle of a triangle using distance x along the wall and the distance from the wall itself, then converting the x coordinate to radians.

Finding the tilt angle has to take into account y position as well as the distance that the x has moved (because the pan servo moves before the tilt servo, so by nature the tilt servo has to take the pan's distance into account).

Because this is measured after panning, the triangle the tilt angle is formed by the hypotenuse of the last triangle (representative of the servo's direction after moving x distance) and the distance y up the wall. This number is also converted into radians.

See the following figure for conceptual help here. If you get really stuck, please seek help from a course instructor or Ninjas.



GETTING STARTED

Here's the fun part! Now that you have everything hooked up, you can start programming the turret.

Step 1

Make it such that you can input a pan angle, a tilt angle, and whether to have the laser on or off into the turret, and the turret will point its laser in that direction. You'll have to find a way to account for the pan and servo angles being offset from what you might expect 0 degrees to be. Also, you'll find the MATLAB arduino servo help extremely helpful here.

Step 2

Hooray! We can now command our laser turret! Now we move on to something with a little more math. Right now, we can give our turret two angles for positioning, but we have no idea where our actual laser dot is being projected onto the wall. With a bit of trig, we can make it so that instead of inputting two angles to our turret, we can give it an x and y position for the dot on the wall. Make it so that you can input an x and y position to the turret, and the laser dot will be projected onto a corresponding point on the wall. Don't try and take on all the trig for this at once. Try to first make it so that you can input an x position and have the pan servo change angle appropriately. Then tackle the y position. Also, DRAW sketches of this, it makes the math for this so much easier and more intuitive.

Step 3

Congratulations on making it this far! This is where the math gets *really* interesting. Now that you can make your turret point at specific points, it's time to draw some lines. Make a program where you can input a starting x,y position and an ending x,y position, and the turret draws a straight line between the two points. Start with a fixed non-zero y position, and changing x positions for your line. You might notice that the turret doesn't automatically draw straight lines right away. This is where you might want to double check your math with someone, and look into interpolation.

Step 4

Wow, you're good! By this point, you've built yourself a solid foundation for understanding and manipulating this system. Here's the part where you should get creative and do something fun with the laser! For example, drawing randomly positioned and randomly sized squares on the wall. Go nuts!

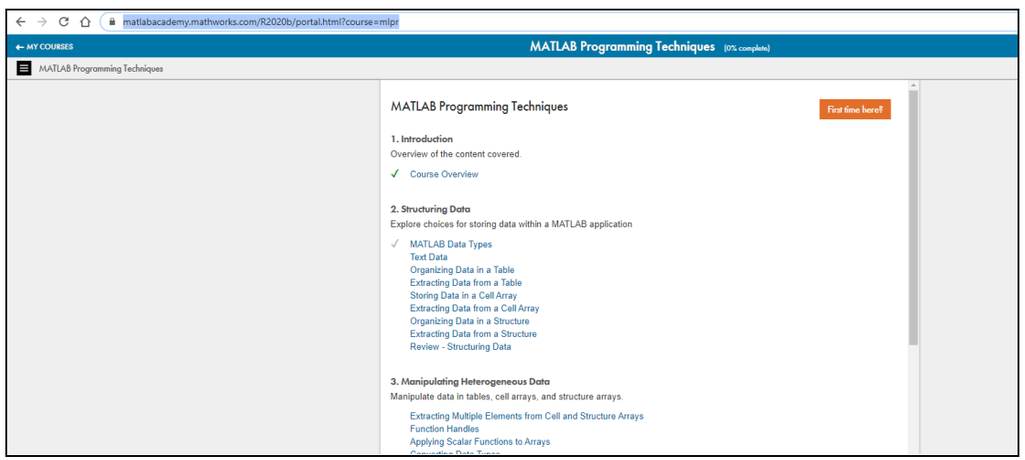
Two week demo: Please write this code, debug it, and be prepared to demo commanding your Pan Tilt Laser assembly to draw out a rectangular box on the far wall of the Robot Lab.

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

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.

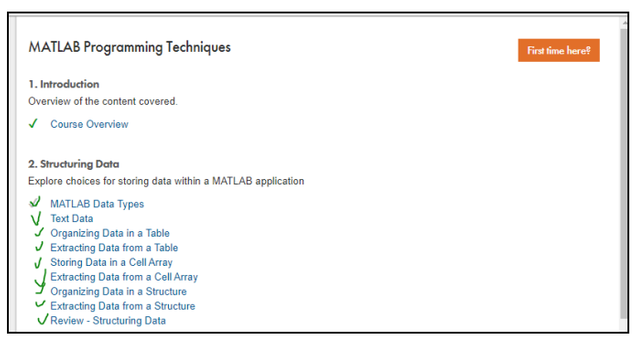
Please see a course instructor or Ninjas for MATLAB help as needed.

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