# Exercise 2: Localization and Control Tutorial *

## EENG350: Systems Exploration, Engineering, and Design Laboratory

Prachi Sharma and Tyrone Vincent

Department of Electrical Engineering
Colorado School of Mines

Fall 2023

## Contents



# 1 Tutorials

## 1.1 Simulink Introduction

Simulink is a block-diagram oriented system simulation tool. Block diagrams are simply graphical representations of equations, and help us easily see the relationship between variables in a way that is not as clear when the equations are written out. There are several resources that you can use to become familiar with Simulink.

- For an introduction into the basics of Simulink (creating a model and simulating it) follow along to this tutorial: http://ctms.engin.umich.edu/CTMS/index.php?aux=Basics_Simulink. You will want have Matlab available to that you can re-create the model on your own computer.
- To see how to create Simulink block diagram that represents a set of differential equations, use this tutorial: http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=SimulinkModeling

- For more details and some advanced concepts, you can view Simulink video tutorials created by the Mathworks. In particular, we suggest the Simulink On-Ramp: https://matlabacademy.mathworks.com/details/simulink-onramp/simulink as well as the "What Is System Identification?" tutorial series, particularly the first two parts https://www.mathworks.com/support/search.html/videos/system-identification-part-1-what-is-system-identification-1636628273301.html.

Note that you can get access to Matlab and Simulink at most any computer lab on campus, and in addition, you can utilize the virtual lab at http://matlab.mines.edu. It is expected that you are already familiar with MATLAB and creating MATLAB functions. If not, you may also want to look at the Mathworks tutorials for MATLAB https://www.mathworks.com/support/learn-with-matlab-tutorials.html.

### 1.2 Pulse Width Modulation

When controlling a physical device using a microcontroller, like a motor, we often would like to vary an analog signal that is input to the device, such as the voltage applied to the motor. However, digital to analog converters that have a continuously variable voltage output are expensive. Fortunately, most devices that we want to actuate have a frequency response that is similar to a low-pass filter. This means that the most important part of the actuation signal is its local average. This allows digital microprocessors with digital outputs to mimic a continuously variable analog signal using pulse width modulation (PWM). Here are some resources to learn more about PWM

- A general description of PWM signals: https://www.analogictips.com/pulse-width-modulation-pwm/
- A tutorial from Arduino: https://www.arduino.cc/en/Tutorial/PWM.

### 1.3 H-bridge motor driver

You will be using a motor driver board for the Arduino that two H-bridge circuits to apply a battery voltage to two motors in a controlled way.

- The wikipedia description of an h-bridge: https://en.wikipedia.org/wiki/H-bridge.
- The users guide for the motor driver you will be using: https://www.pololu.com/docs/0J55. Download the guide, save it in a place your team can easily access it. Read the guide, skimming the first sections, but reading section 3.c in detail. There is an Arduino library described in 3.d, but **do not use it**. It is not necessary once you understand the pin mappings given in section 3.c, although a few **warnings** are in order about the terminology. The description of pins 9 and 10 as “Motor speed input” and pins 7 and 8 as "Direction" are **incorrect terminology**. Pin 7 and 8 determine which set of switches will be closed during operation (i.e. choosing to close S1 and S4 together, or close S2 an S3 together as descibed in the Wikipedia description.) Thus, a better label for these pins is "Voltage sign". Pins 9 and 10 specify whether the switches are closed or open. When a PWM signal is applied to these pins, it will determine the average voltage applied to the motor. Thus these pins could be better labeled “Motor voltage”.

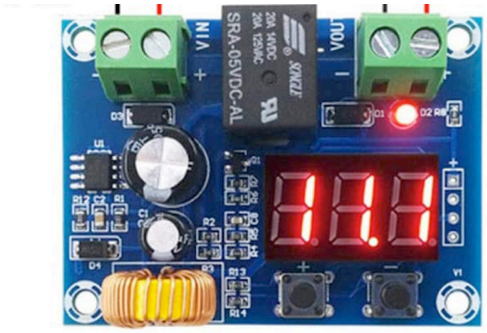
## 2 Hardware Setup

### 2.1 Suggested Construction

For this exercise, we will build a mount that can hold the two motors and wheels upright for testing. Build a T with two longer actobotics shaft pieces, using the flat bracket to connect two longer ends. It will be useful to have the open end of the shaft for the stem of the T facing up, while the crossing bracket has the open end facing out (horizontally). Connect two additional smaller shafts so that the T can sit above the lab bench. Mount two motor brackets on the ends of one arm of the T, clamp down two motors, and mount the wheels on the motor shafts. Put the motor driver shield on the Arduino, and connect the battery to the motor driver via the switch. Connect the motor driver shield onto the Arduino. Hook up a battery (via the switch and fuse connector) to the voltage monitor, and from the voltage monitor,

connect to the motor driver power input. Note that there is a jumper on the motor driver board that allows the battery to also be the power supply for the Arduino, but it is not necessary if you have the Arduino connected via USB.

The voltage monitor looks like this:

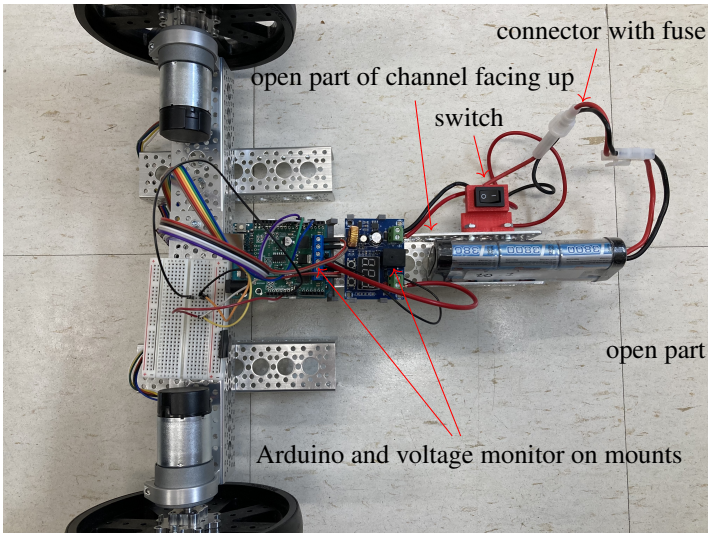


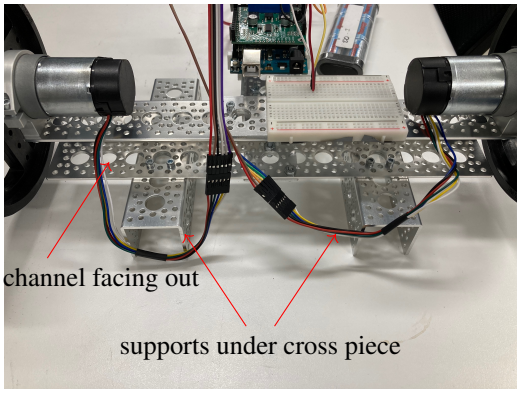

**Instruction**

**"+" button:** Press once to display the protection value, long press to set the protection value
**"-" button:** Press once to display the return difference value, long press to set the return difference value

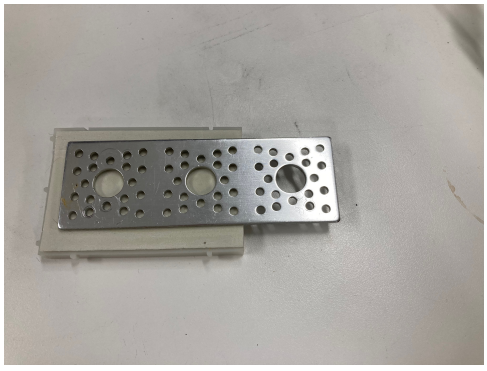
We suggest setting a protection value of 7.4 V, and a return difference value of 0.3 V.

Here are some pictures of the final setup (connections to the motors have also been made and a breadboard mounted in this photo, but you will do that below)




The breadboard is attached to a flat bracket using the adhesive on the back, or double sided tape. The flat bracket can then be screwed on to a channel. **Only connect low current lines using the breadboard. DO NOT make connections to the motor power lines using this, as the motor can draw up to 5.5 A**



You are using Pololu 37D gear motors with a 50 to 1 gear ratio. On the motor product page, there is a list of the functions of each wire of the motor connector:

| Color | Function |
|---|---|
| Red | motor power (connects to one motor terminal) |
| Black | motor power (connects to the other motor terminal) |
| Green | encoder GND |
| Blue | encoder Vcc (3.5 – 20 V) |
| Yellow | encoder A output |
| White | encoder B output |

Hook up the motor power leads to one of the motor driver output channels. Connect the encoder leads to the appropriate pins on the Arduino. Since there are two motors, you won't be able to use a interrupt pin for both encoder output pins, so use pin 2 for one encoder's A output, and pin 3 for the other. For the B output, you need to use a pin that is not also being used by the motor driver. You can read the motor driver's guide (link below) for the full list, but you will be safe if you use pins 5 and 6 for the B outputs.

## 2.2 Use a motor driver to spin a motor under control of the Arduino

You will use `AnalogWrite()` to create a pulse width modulated signal on pins 9 and 10. Since `AnalogWrite` uses timer1 for pins 9 and 10, you should not use timer1 as a timer interrupt for anything else.

1. Use the Arduino to spin the motors. The Arduino is able to supply a pulse-width-modulated signal on pins 9 or 10 using using the `analogWrite` command.

   In the `setup()` portion of your code, you will want to setup pins 4, 7, 8, 9, and 10 for output. Set the enable pin (pin 4) high. In the `main()` portion of the code use the `analogWrite` command on pins 9 or 10 to command the motor driver to supply a pulse width modulated signal of a desired pulse width to one of the wheels. Set the sign of this voltage using pins 7 or 8)

2. Use the oscilloscope to measure the voltage at pin 9 or 10, as well as the voltage across motor. Verify what value of `analogWrite` will correspond to a particular pulse width. Verify that pins 7 and 8 change the voltage sign.

3. Use your code from assignment #1 to read the wheel encoders. Since the encoder has a resolution of 64 counts (measured on the motor shaft,) and the motor has a 50 to 1 gear ratio, you should verify that you count $64 \times 50 = 3200$ counts per revolution. Don't use the motor for this test, turn the wheels by hand. It will be most convenient if the counts increase when the motors are rotated the same direction. Since they are mounted in opposite directions, probably they count in opposite directions. You can change this direction for one of the motors by swapping the A and B encoder output pins where the connect to the Arduino.

4. Apply a constant voltage to the motors and display the wheel rotation in radians with a fixed sample rate. Here are some hints: add these variables definitions at the top of the file:

   ```
   unsigned long desired_Ts_ms = 10; // desired sample time in milliseconds
   unsigned long last_time_ms;
   unsigned long start_time_ms;
   float current_time;
   ```

   Add this code to your setup

   ```
   void setup() {
   ```

```
⋮
⋮  (your other setup code here for encoder and motor pins)
⋮

 Serial.begin(115200); // Set the baud rate fast so that we can display the results
 last_time_ms = millis(); // set up sample time variable
 start_time_ms = last_time_ms;
}
```

Use code like the following in the main loop:

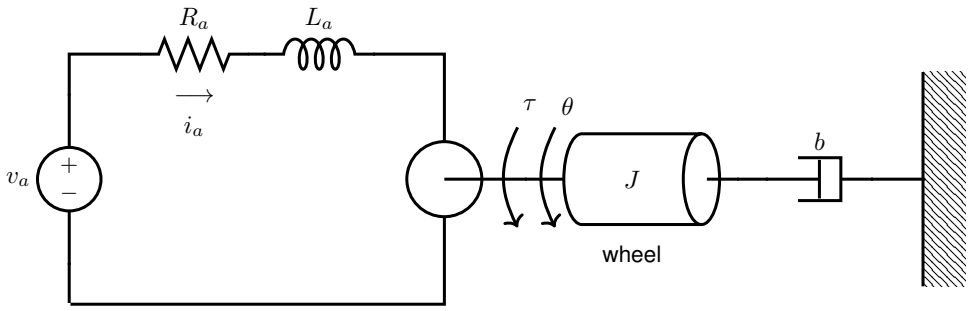
```
void loop() {
 // Variables for calculating position
 long pos1_counts;
 long pos2_counts;
 float pos1_rad;
 float pos2_rad;

⋮
⋮  (your other loop code here to run the motors)
⋮

   pos1_counts=myEnc(1); // your encoder function that returns the current
                         // position for motor 1
   pos2_counts=myEnc(2); // your encoder function that returns the current
                         // position for motor 2
   pos1_rad= 2*pi*(float)pos1_counts/3200;
   pos2_rad= 2*pi*(float)pos2_counts/3200;

   // print out timestamp in seconds
   current_time = (float)(last_time_ms-start_time_ms)/1000;
   Serial.print(current_time);
   Serial.print("\t");
   Serial.print(pos1_rad);
   Serial.print("\t");
   Serial.print(pos2_rad);
   Serial.println("");


    while (millis()<last_time_ms + desired_Ts_ms) {
        //wait until desired time passes to go top of the loop
       }
    last_time_ms = millis();
}
```

You will have to decide on a sign convention. It will be most convenient if the wheels spin so that the counts increase when pins 7 and 8 are set to `HIGH`. If the counts are going in the wrong direction, reverse the motor power connections (the red and black wires on the motor) to the motor driver board.

# 3 Simulation and Control of a Motor

## 3.1 Motor Model

A DC motor driving a wheel with bearing friction has the following ideal component representation. (You can review this in EENG307 Lecture 11, Motors and Hydraulic Actuators.)




where $b$ is the friction constant, and $J$ is the load inertia.

For this load, transfer function from motor voltage $v_a$ to angular velocity $\omega = \dot{\theta}$ is

$$\frac{\omega(s)}{V_a(s)} = \frac{\frac{K_t K_e}{R_a J}}{s + \frac{R_a b + K_t K_e}{R_a J}},$$

where $K_t$ is the motor torque constant, $K_e$ is the motor back emf and $R_a$ is the internal motor resistance. While this looks like a complicated function with many parameters, we see that is of the form

$$\frac{\omega(s)}{V_a(s)} = \frac{K\sigma}{s + \sigma},$$

which is a first order system with DC gain $K$ and time constant $\frac{1}{\sigma}$. In order to simulate the dynamic behavior of this system, we only need to know the two parameters

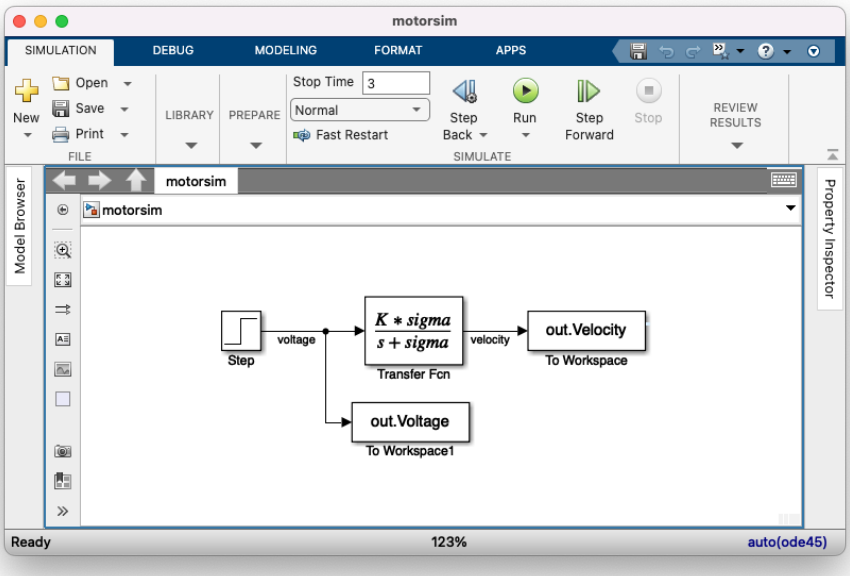
$$K = \frac{K_t K_e}{R_a b + K_t K_e},$$

and

$$\sigma = \frac{R_a b + K_t K_e}{R_a J}.$$

## 3.2 Simulink Motor Model

You can create a simulink block diagram to simulate the motor system as shown below.




There are several important points to note about this block diagram:

- We have set the simulation length to 3 seconds using the data entry box at the upper center of the window next to the words 'Stop Time'.
- We have added a step input with amplitude 1 and step time 1 s as the voltage input ($V_a$).
- The to workspace blocks are used to save the results in the variables `out.Velocity` and `out.Voltage`.
- Two of the signals have been named in the diagram: voltage and velocity (look for text below the lines/arrows). The variable voltage is the voltage $V_a$ and the variable velocity is the angular velocity $\omega$. You can name a signal by clicking on the line/arrow associated with that signal and typing the name.

The following code sets up the parameters, runs the simulation (assuming you have saved your Simulink block diagram as `motorsim.slx`,) and plots the results.

```
%% Runmotorsim.m
% This script runs a simulation of a motor and plots the results
%
% required file: motorsim.slx
%
%% Define motor parameters
K=2.2; % DC gain [rad/Vs]
sigma=5; % time constant reciprocal [1/s]

%% Run a Simulation
%
% open the block diagram so it appears in the documentation when published.
% Make sure the block diagram is closed before running the publish function
%
open_system('motorsim')

%
% run the simulation
%
```

```
out=sim('motorsim');

%% A Plot of the results
%
figure
subplot(2,1,1)
plot(out.Voltage,'linewidth',2)
xlabel('Time (s)')
ylabel('Voltage (V)')
subplot(2,1,2)
plot(out.Velocity,'linewidth',2)
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
```

Note that simulation results are saved in the structured variable `out`, and `out.simout` is accessed using the dot notation. The default for the to workspace variables are structures called Timeseries, that include both the time and signal magnitude. (Type `out.simout`) at the command line to see what is included in the structure.

### 3.3 Finding motor parameters

We would like to be able to simulate the specific motors that you have available. This means finding the parameters $K$ and $\sigma$ that characterize the motors. This can be done experimentally through a step response test. Modify your code from Section 2.2 to print the time, voltage applied to one of the motors, and velocity of the motor in three columns. The (average) voltage of the PWM signal applied to the motor will be your battery voltage times the value of AnalogWrite command, divided by 255. Use `if` statements to set the voltage depending on the `current_time`. Have the initial voltage be 0, then between 1 and 3 seconds, set the voltage to a mid-range value (about 3 V), and then set the voltage back to zero. To calculate the velocity, you will have to add a variable that can hold the motor position at the prior iteration through the loop, and calculate the velocity as the current position in radians minus the prior position divided by the sample time in seconds (this would be `(float)desired_Ts_ms/1000`). To help easily collect the data, you can also use an `if` statement to only display data for the first 3 seconds.

When you run your experiment, the Arduino should display 3 tab-delimited columns in the serial monitor. In Subsection 2.2, you should have wired the motors so that the velocity is positive when pins 7 and 8 high. If this is not the case, go back to that section and correct it before going forward. Uncheck the autoscroll button, and scroll to the top. Select the first 3 seconds of the experiment, and copy it. In MATLAB, type `data=[];` in the command window, and then double click on `data` in the workspace browser. (See link if it is not visible). You can then paste your data into the spreadsheet that opens up. The variable `data` now contains your experiment. To save the variable for use later, `save('stepData.mat','data')` at the command line. When `stepData.mat` is in the current directory, you can load the variable `data` back into the workspace by typing `load('stepData.mat')`.

> **Bug alert:** There is a bug in the new versions of the Arduino IDE that do not allow you select data in the serial monitor properly. There are a few options for getting around this:
>
> - Use a different serial monitor, such as Putty, Screen, or TerraTerm. Before you run this program **make sure** you have shut the Arduino serial monitor.
> - Use Matlab - there is file called `ReadFromArduino.mlx` available on the Tips and Tricks page on Canvas. To use this file, add the line `Serial.println("Ready!");` in your `setup()` function, after your `Serial.begin`. After you display your data for 3 seconds, execute `Serial.println("Finished");`. Before you run this file **make sure** you have shut the Arduino serial monitor. It will save your data in the file `stepData.mat` with the variable name `data`.

Copy your Matlab script from presented in Section 3.2, and save it as a separate file. You can compare your experimental data to your simulated data by adding the code

```
load('stepData.mat')
```

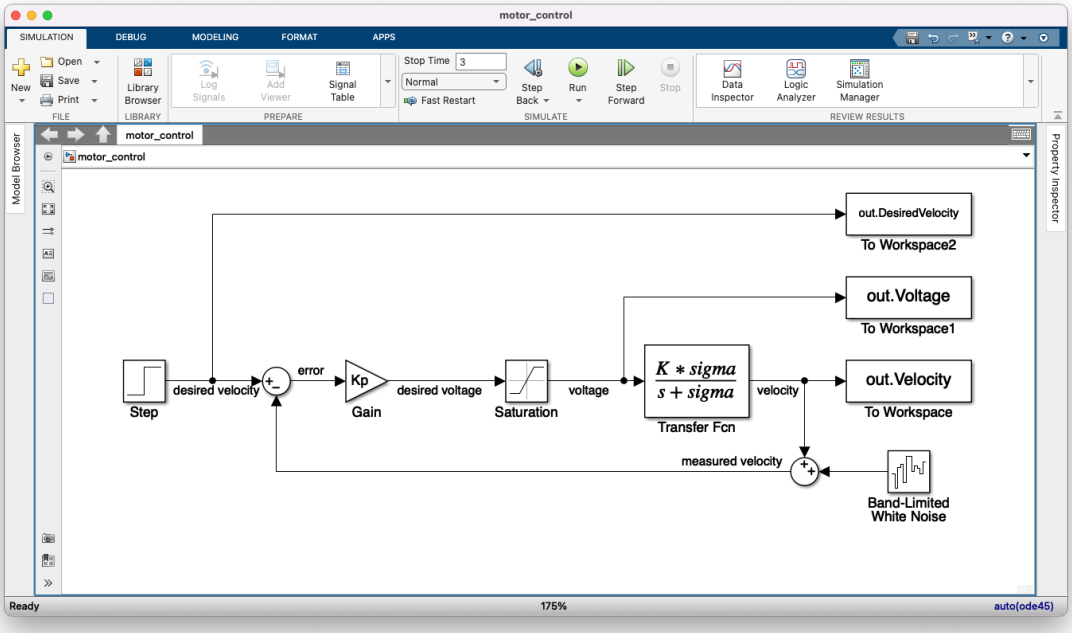
to the top of the script and modifying your plots to be

```
figure
subplot(2,1,1)
plot(out.Voltage,'--','linewidth',2)
hold on
plot(data(:,1),data(:,2),'linewidth',2)
legend('Simulated','Experimental','location','southeast')
hold off
xlabel('Time (s)')
ylabel('Voltage (V)')
subplot(2,1,2)
plot(out.Velocity,'--','linewidth',2)
hold on
plot(data(:,1),data(:,3),'linewidth',2)
hold off
legend('Simulated','Experimental','location','southeast')
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
```

Your goal is to adjust the simulation voltage, step time, $K$, and $\sigma$ so that the plots all line up. The experimental velocity plot will likely look noisy. This is actually due to the quantized nature of the encoders, as they only measure position to within a fixed accuracy. You will want to have the simulated velocity match average of the experimental data.

## 3.4 Velocity Control Design

Once you have found the transfer function parameters $K$ and $\sigma$, you can use Simulink to design and test a controller for the motor. Copy your `motorsim.xls` simulink block diagram file and rename the copy as `motor_control.slx`. Add the components to your simulink block diagram so that is looks like the following:




Here are some things to note about this block diagram:

- We have built a feedback control loop around the motor transfer function. The motor velocity is fed back to a summer (with signs chosen so that the feedback is negative) to calculate the error between the actual velocity and the desired velocity. This error is multiplied by the gain $K_p$, so this is a Proportional feedback controller.
- You have observed that the measured velocity is noisy. To model that, we have added band-limited white noise to the velocity. To get the summer and noise block to face the right way, we selected these blocks and hit `ctrl-i`. The summer list of signs is `++|`. In the band-limited white noise block, we chose Noise Power as .0001 and sample time .01 s.
- The Step block still has magnitude 1 and step time one, but now it is setting the desired velocity in radians per second, rather than the motor voltage.
- The saturation block has been set with an upper bound of 7.5 and a lower bound of -7.5. This models the fact that the largest average voltage that can be applied to the motor is the battery voltage (your battery may be between 7.5 and 8.2 V)

Copy or duplicate your code that ran the `motorsim` block diagram, and rename it. At a line that sets $Kp$ to a desired value. Change the `opensim` and `sim` commands to open and run `motor_control`. Modify your second plot to be

```
subplot(2,1,2)
plot(out.Velocity,'linewidth',2)
hold on
plot(out.DesiredVelocity,'--','linewidth',2)
```

Use this code to choose a value of `Kp` so that the steady state velocity comes within 20% of the desired velocity, but is not affected by the sensor noise too much.

## 3.5 Velocity Control Implementation

The control implementation of a proportional controller is very simple: find the error, and multiply it by the gain! However, we need to take care of a few subtleties concerning the voltage sign and the voltage limits.

We assume the following

- You have set up the floating point variable `desired_speed` to hold the desired wheel speed, and calculate the floating point variable `actual_speed` using the encoders
- `Kp` is defined as a float and set to your controller gain
- `Battery_Voltage` is set to 7.8 (expected battery voltage)
- `error` and `Voltage` are defined as floats
- `PWM` is defined as an unsigned int.

The following code implements the controller for Motor 1:

```
error = desired_speed - actual_speed;
Voltage = Kp*error;

// check the sign of voltage and set the motor driver sign pin as appropriate
if (Voltage>0) {
   digitalWrite(7,HIGH);
} else {
   digitalWrite(7,LOW);
}
// Apply the requested voltage, up to the maximum available
PWM = 255*abs(Voltage)/Battery_Voltage;
analogWrite(9,min(PWM,255));
```

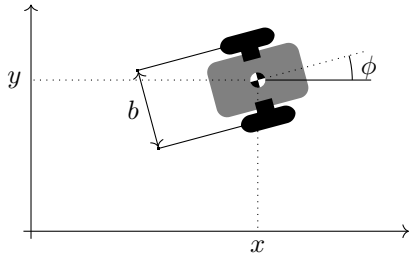
Implement a controller for both motors. For each motors separately, use `Serial.print` statements to display the time, motor voltage, and velocity. Have the desired velocity go from 0 to 1 at 1 second. Copy the data to Matlab, and display the simulated and experimental closed loop step response on the same plots.

> **Note:** Because of quantization error, the velocity measurements will have more variations than the actual velocity, and could look fairly noisy. This is ok as long as the wheel moves relatively smoothly according to your eyes. There are ways of fixing the velocity measurement, but since your final project will ultimately be more interested in position accuracy, we won't waste time on that now.

# 4 Localization

## 4.1 Calculating Position and Orientation

In this section, we discuss taking the wheel velcoity measurements, and using them to estimate the robot's position and orientation on the ground. The setup is as shown in the diagram below. A two wheeled robot with a baseline of $b$ m between the wheels is located at position $x$, $y$ relative to a fixed axis (the origin is perhaps the starting point of the robot). The orientation of the robot is relative to the x axis, and measured by the angle $\phi$. The is the direction the robot will go if the wheels turn equally.




Let $v_\ell$ be the velocity of the left wheel in ms$^{-1}$. This is the velocity with which the point where the wheel touches the ground moves along the ground. Since the encoders measure the angular velocity, you will need to convert to get the velocity of the wheel along the ground. If the wheel has radius $r$ in meters, and $\dot{\theta}$ is the rotational velocity of the left wheel in rad/s, then $v_{ell} = r\dot{\theta}$ ms$^{-1}$. Similarly, let $v_r$ be the velocity of the right wheel. The position and orientation of the vehicle satisfies the following set of differential equations

$$\begin{aligned} \dot{x} &= \cos(\phi)(v_\ell + v_r)/2, \\ \dot{y} &= \sin(\phi)(v_\ell + v_r)/2, \\ \dot{\phi} &= (v_r - v_\ell)/b. \end{aligned} \tag{1}$$

Given measurements of the wheel velocities, you can keep track of the robot position and orientation by solving this set of differential equations. Howver, since the arduino is a discrete time system, we will have to make a discrete time approximation.

If you have a differential equation

$$\dot{z} = f(z),$$

an Euler approximation calculates the current value of $z$, denoted as $z_{\text{new}}$ as

$$z_{\text{new}} = z_{\text{old}} + f(z_{\text{old}})\Delta t,$$

where $\Delta t$ is the time since $z_{\text{old}}$ was calculated. The update equations to find the current position and orientation are then

$$\begin{aligned} x_{\text{new}} &= x_{\text{old}} + \Delta t \cos(\phi_{\text{old}})(v_\ell + v_r)/2, \\ y_{\text{new}} &= y_{\text{old}} + \Delta t \sin(\phi_{\text{old}})(v_\ell + v_r)/2, \\ \phi_{\text{new}} &= \phi_{\text{old}} + \Delta t \frac{1}{b}(v_\ell - v_r). \end{aligned}$$

These equations also show us an alternate implementation. Notice that all the velocities are multiplied by $\Delta t$. But $\Delta t v_\ell$ is none other than the distance moved in $\Delta t$ time. That is, let $d_\ell$ be the distance the left wheel has moved since the robot was turned on. Then $v_\ell = \dot{d}_\ell$, and over elapsed time $\Delta t$, the change in distance is related to the velocity via $\Delta d_\ell = \Delta t v_\ell$. Letting $d_r$ be the distance the right wheel has moved,

$$\begin{aligned} x_{\text{new}} &= x_{\text{old}} + \cos(\phi_{\text{old}})(\Delta d_\ell + \Delta d_r)/2, \\ y_{\text{new}} &= y_{\text{old}} + \sin(\phi_{\text{old}})(\Delta d_\ell + \Delta d_r)/2, \\ \phi_{\text{new}} &= \phi_{\text{old}} + \frac{1}{b}(\Delta d_\ell - \Delta d_r). \end{aligned}$$

In short, instead of waiting $\Delta t$, measuring the change in distance, and converting to velocity, it turns out we can just work with the change in distance. However, either method will work and in fact they are exactly equivalent.

You have some flexibility in selecting units. So far, we have worked in SI units, so $\Delta d_\ell$, $\Delta d_r$, $b$ and $r$ will be in m. If you would like to report your position in feet instead, you can change all of these units to feet ($\phi$ will still be calculated in radians). The important thing is to be consistent - they all have to be the same units!

Using these equations, calculate the position and orientation of your wheels (if they were moving along the ground). In your implementation, pay attention to type casting! A common mistake is to divide two integers - the calculation will be done as an integer, so if the denominator is bigger than the numerator, you will get zero. Use type casting when doing arithmetic with integers that you want to record as a double. To test your implementation, you can rotate the wheels by hand, or use the motors to move them.

## 4.2 Localization Animation

Although you can view the results of your localization code with print statements, an animation can be much more illuminating. In Matlab, animations can be made by simply re-plotting data at a set rate. For example, try the following code:

```
%% Animation of the position and orientation of a vehicle

% define some data to animate
% first column is time, second is x, thrid is y, fourth is phi
data = [0    0  0  0;
       .5    0  0  0;
        1    1  0  0;
        1.5  2  0  .3;
        2    3  1  .5;
        2.5  4  1   0;
        3    5  1   1.57;
        3.5  5  2   1.57;
        4    5  3   1.57;
        4.5  5  4   1.57;
        5    5  5   1.57];

time      = data(:,1);
xposition = data(:,2);
yposition = data(:,3);
phi       = data(:,4);

% Define shape verticies
% first row is x position, second row is y position
r_width =   .5;
r_length =  1;
V = [-r_length/2 -r_length/2    0          r_length/2    0;
     -r_width/2   r_width/2     r_width/2  0            -r_width/2];
```

```
figure
for i=1:length(time),
    % move shape by moving verticies
    % define rotation matrix
    T = [cos(phi(i)) -sin(phi(i));sin(phi(i)) cos(phi(i))];
    % define center position
    pos = [xposition(i);yposition(i)];
    % find position of current vertices: each position is multiplied by the rotation
    %matrix, and added to the current position
    v_c = T*V+pos*ones(1,5);
    % draw shape
    fill(v_c(1,:),v_c(2,:),'y')
    axis([-10 10 -10 10]) % set axis to have specified x and y limits
    % (type 'help axis' for more info)
    % make sure matlab draws the figure now
    drawnow
    % if not last drawing, wait
    if i<length(time),
     pause(time(i+1)-time(i))
    end;
end;
```

You can animate the results of your robot localization by using `serial.Print` to display the Arduino's calculation of time, $x$, $y$, and $\phi$, and then copying the results into the variable `data` (deleting the assignment of data in the code above before running).