Design Problem 2

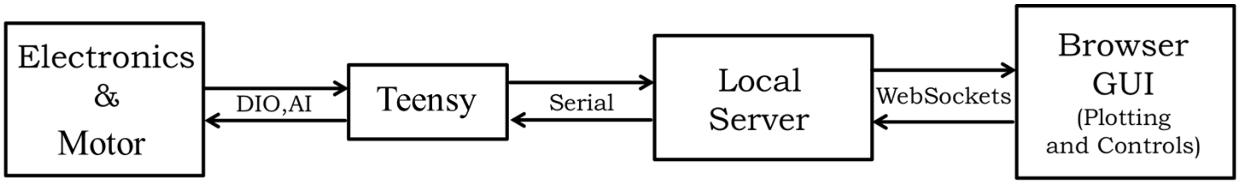
In this design problem you will be asked to design and build several controllers using the Teensy microcontroller.

Problem 0 – Setup (Nothing Must be Submitted For this Problem)

We are first going to design and test a simple controller using the Teensy microcontroller. This problem will walk through the basics of setting up the system for use.

For this design problem and the next, we've designed a set of software that should let you easily vary feedback control parameters in live-running systems controlled by your Teensy. This software is brand new. It is all open-source, and built using open-source tools. At the end of the course, you are invited to use the code in any way you desire. You may find it useful in senior design and in thesis work.

The basic software layout is shown below:



- The Teensy interfaces with the outside world by sending pulse-width modulated (PWM) signals to the motor drive and by reading appropriate sensor signals. Its pwm output signal is derived from sensor and motor readings in a feedback control loop that you will construct. The control loop's behavior is adjusted by values/parameters sent to it from a python-based local server running on your computer using serial communication. In addition, at regular intervals, the Teensy sends its measurements back to the Python server.
- The python-based local server running on your computer is an interface. It communicates with the Arduino via serial link, and with a browser-based GUI via WebSockets.
- The Browser-based GUI plots measured data from the Teensy, passed to it by the local server, and provides simple interface for users to change the Teensy feedback loop parameters, which it passes on to the local server.

The entire package of software is broken down into three pieces:

- **Microcontroller Code:** This is the code uploaded to the Teensy (the .ino file). It sets the Teensy up for listening to and controlling the electronics as well as taking in parameters and reporting values to your computer over serial (the USB cable). It is written in Arduino's C/C++ type language.
- **Server Code:** This is a Python script that talks with the Arduino over serial, does some calculations, and then talks with the GUI code in your browser. It is the "middle-person" of our software stack.

- **GUI Code:** Your GUI (Graphical User Interface) is basically just a web-page that is hosted on your machine. It is written in standard html/css/javascript. It is locally hosted and only you have access to it! It lives at localhost:3000 in your browser url field when the server script is running.

Software Installation

Getting Started

The code distribution we'll use for this software introduction is found in the file 4111_View_v5c.zip on the Canvas page. This is the base file for testing and setting up. While the Python and html/javascript files are the same for all Design Problems, we use different microcontroller code sets for each so you will need to download them separately for each Design Problem.

Download the .zip file, extract it, and look inside. You'll see the following things:

- A called csv_files. Don't worry about this thing for right now.
- The file server.py. This is the Python script you will run on your machine for establishing communiation with your Arduino and browser GUI (Graphical User Interface)
- The file index.html. This is a webpage that acts as your GUI.
- The file req.txt. This is a file that lists libraries needed in addition to normal Python. Don't worry about it right now, but we'll use it during install.
- The directory static. This has the javascript/css libraries your browser GUI uses. We use local copies of these libraries so you can run this software when not connected to the internet. Just leave this as it is.
- Two directories called teensy_test and arduino_test. These contain simple communication check microcontroller code files for both the Arduino Uno and the Teensy 3.2. They are good "sanity check" files.

Arduino IDE Install

Your code will be developed using the Arduino IDE. If you do not already have this, head to one of the following links and follow the instructions:

- Windows

- Mac OS X

Teensyduino Add-On

The Teensy can be programmed using the Arduino IDE. To do so, you need to go to the PJRC software site at this location and follow the instructions for adding their Teensy add-on software. This stuff has been shown to work on all OS's!

Setting Up the Python Server

Windows does not come with Python usually. If you already have it, determine what version you have:

- If it is Python 2, make sure it is at or above Python v2.7.9
- If it is Python 3, make sure it is at or above Python v3.4

One easy way to check on your Python is to open up a PowerShell. This is a Linux-like emulator that comes standard in Windows. To find it, go to the search bar and type Windows PowerShell. DO NOT use Windows PowerShell ISE. I'm not quite sure what "ISE" is, but it does not work.

Once in PowerShell, type

```
python --version
```

Some text should come back telling you if you have Python and what version it is. Red text probably means you don't have it. Our code should work on both Python versions 2.7 and above as well as Python 3.3 and above. If you do not have Python installed, you will need to do that. I'd recommend getting Python 3 for this software, though both families (2.7+ and 3+ will work). Go to the Python Download Page (https://www.python.org/downloads/), find the appropriate install version for your computer (I'd recommend the .exe installer), follow the well-written directions, and you should be good to go. **IMPORTANT!!!: *It is very important that during the install you check the option about adding Python to your Path which happens on the first/second window of the installer executable!!!***

Installing Pip

The next piece which you need is pip, which is a nice Python Package Manager that we use to install the appropriate install files. If you download either the newest version of Python 2 (after v 2.7.9) and any version of Python3 starting with Python 3.4 you should have pip preinstalled. To make sure you have pip, run the following command in the Windows PowerShell (not the Python shell!):

```
pip --version
```

If some stuff comes back giving you a version number, you'll be good to go. If not, I'd recommend you update your Python (using the Python download link above) and in the process get pip by default!

4111-Specific Software Install

Assuming you've gotten everything working up to this point, navigate into the folder "4111_View_v5c" inside the Windows PowerShell. Once there, run the command (you may be prompted for a machine password based on your settings.

```
pip install -r req.txt
```

(Note: if you're using Python3 you may need to do pip3> install -r req.txt instead!)

It may take a minute or so depending on your machine. A whole bunch of text should fly by in a neutral color like green (if red stuff appears, it *may* mean there was an error in install).

If the install was a success, the next thing you need to do is start the server using Python. Type:

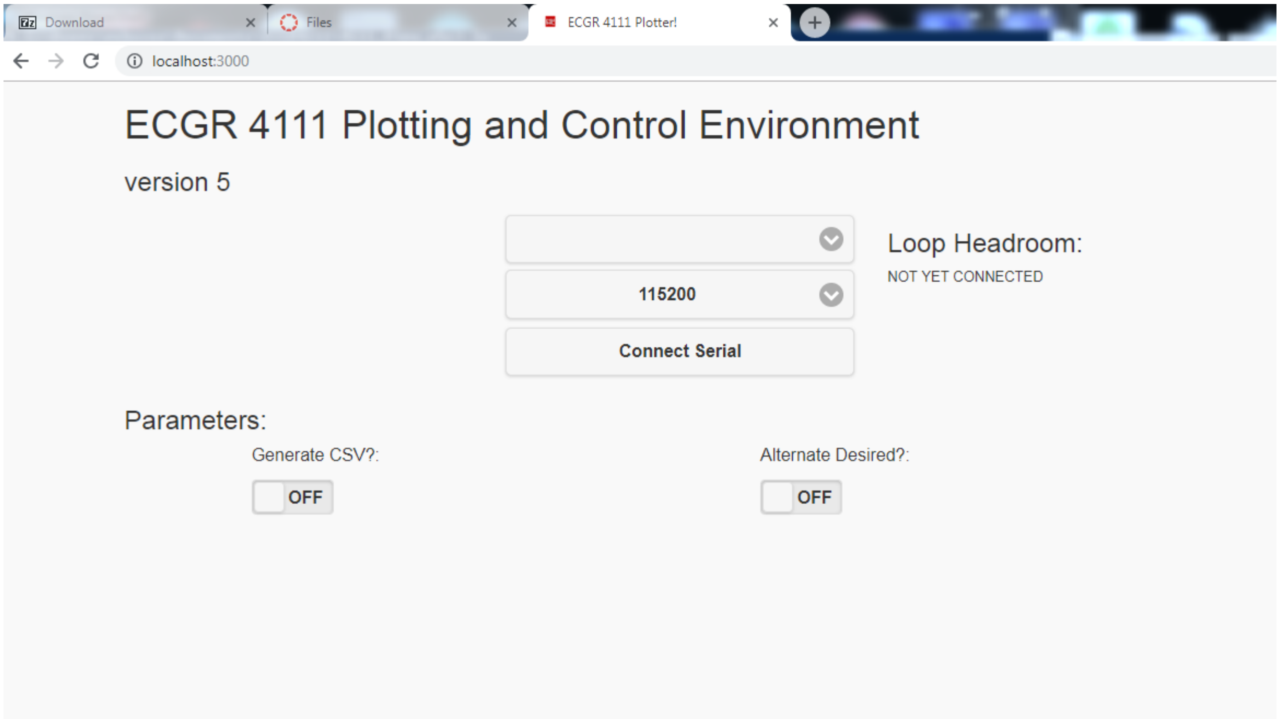
```
python server.py
```

Note: Depending on how you set up your Python, it may be called by the command "python3" rather than just "python". Try both if you have issues. On my several test installs on "clean" Windows machines, when installing Python 3.5, it just established "python" and "pip" as the appropriate commands, but it is possible that the could end up as "python3" and "pip3".

After starting running, some text will appear, but as long as it isn't very long it means you should be good. Head on over to a browser and in the URL field type:

```
localhost:3000
```

You should see something like the image shown below:



ECGR 4111 Plotting and Control Environment

version 5

115200

Loop Headroom:

NOT YET CONNECTED

Parameters:

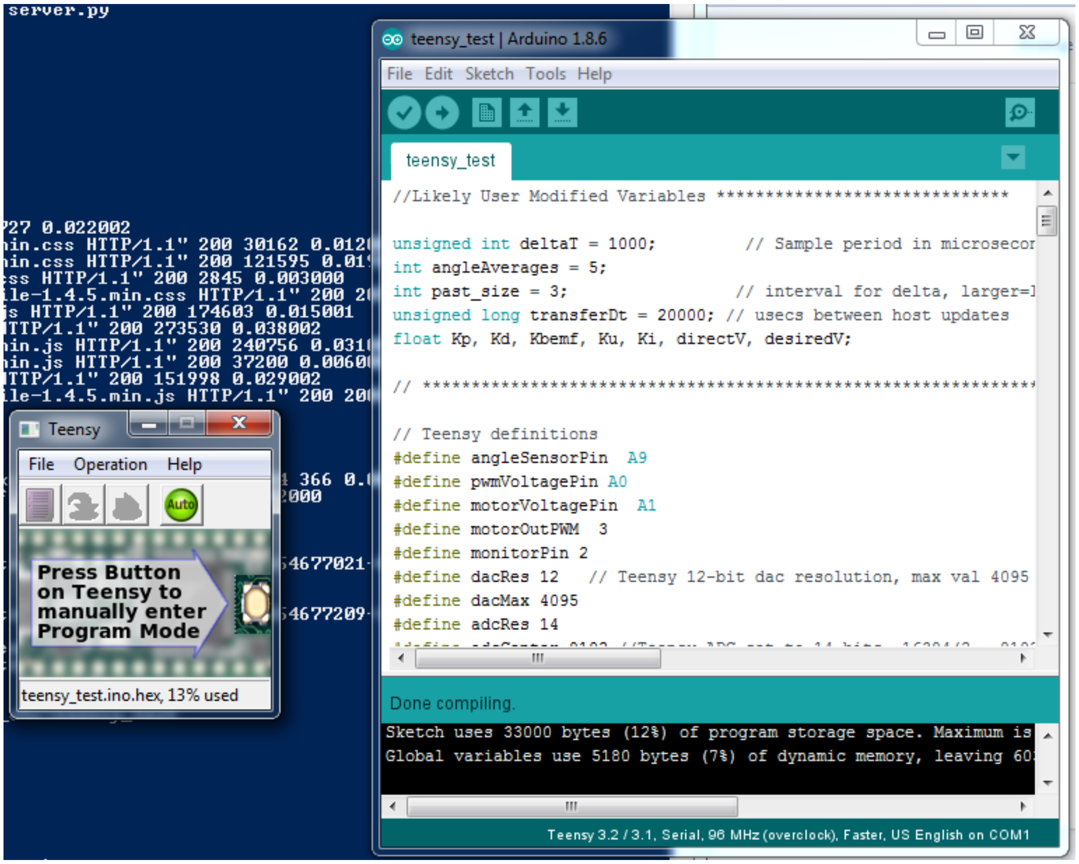
Generate CSV?: OFF

Alternate Desired?: OFF

Testing the Setup

We are now going to test the setup. If you have the plotting and control environment showing, I would do the following:

- Go back to the Windows PowerShell
- Type CTRL-C to kill the Python Server
- Make sure that your Teensy is connected to your computer via USB
- Open the Arduino IDE
- Inside the Arduino IDE, open the file named "teensy_test.ino"
- Once the file is open, click the Checkmark button to compile the code. This should open a new "Teensy" window. I get something like the following:


teensy_test | Arduino 1.8.6
File Edit Sketch Tools Help
teensy_test
Teensy
File Operation Help
Auto
Press Button on Teensy to manually enter Program Mode

```
server.py
```

```
?27 0.022002
hin.css HTTP/1.1" 200 30162 0.012
hin.css HTTP/1.1" 200 121595 0.01
ss HTTP/1.1" 200 2845 0.003000
ile-1.4.5.min.css HTTP/1.1" 200 2
s HTTP/1.1" 200 174603 0.015001
ITTP/1.1" 200 273530 0.038002
hin.js HTTP/1.1" 200 240756 0.031
hin.js HTTP/1.1" 200 37200 0.0060
ITTP/1.1" 200 151998 0.029002
ile-1.4.5.min.js HTTP/1.1" 200 20
```

```
//Likely User Modified Variables ******************************

unsigned int deltaT = 1000;         // Sample period in microsecor
int angleAverages = 5;
int past_size = 3;                  // interval for delta, larger=l
unsigned long transferDt = 20000; // usecs between host updates
float Kp, Kd, Kbemf, Ku, Ki, directV, desiredV;

// *****************************************************************

// Teensy definitions
#define angleSensorPin  A9
#define pwmVoltagePin A0
#define motorVoltagePin  A1
#define motorOutPWM  3
#define monitorPin 2
#define dacRes 12   // Teensy 12-bit dac resolution, max val 4095
#define dacMax 4095
#define adcRes 14
```

teensy_test.ino.hex, 13% used

Done compiling.

```
Sketch uses 33000 bytes (12%) of program storage space. Maximum is
Global variables use 5180 bytes (7%) of dynamic memory, leaving 60
```

Teensy 3.2 / 3.1, Serial, 96 MHz (overclock), Faster, US English on COM1

- Once this is done, click the forward arrow button, which will upload the code to the Teensy.
- Once the code is uploaded, return to the Powershell and start the python server with the code below. Remember, you must be in the folder where this code is located.

```
python server.py
```

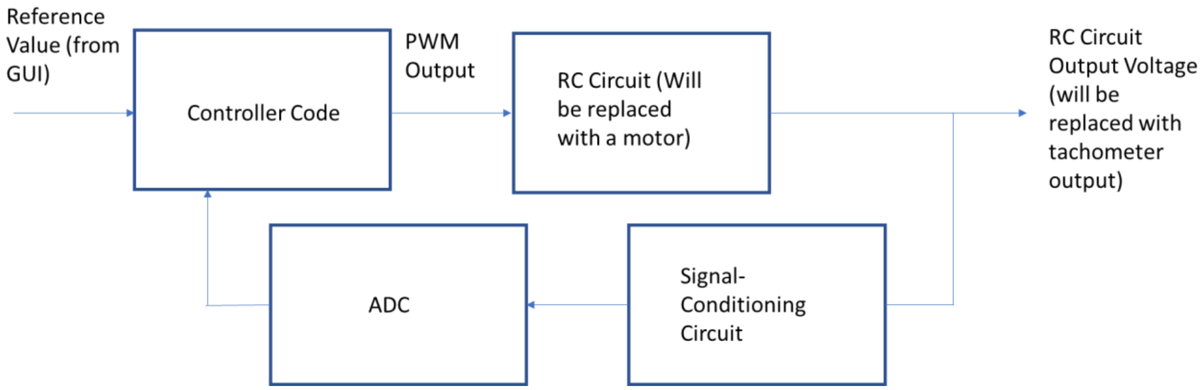
- Once the python server is running (give it a minute once you type the code above), return to your browser and reload localhost:3000. You should now have a fully functional system.

I recommend following this process as you debug:

- Kill the server with CTRL-C
- Modify and then upload your code to the Teensy.
- Restart the server
- Reload localhost:3000

Problem 2

Ultimately, you are going to control the speed of a DC motor in this design problem, but we will start with a similar problem, namely the output of an RC circuit. The image below shows the basic connections:


Reference Value (from GUI)
Controller Code
PWM Output
RC Circuit (Will be replaced with a motor)
RC Circuit Output Voltage (will be replaced with tachometer output)
ADC
Signal-Conditioning Circuit

Bear in mind that this is not the block diagram. It is simply the connection diagram. The PWM output goes into the RC circuit, and the output of the RC circuit is processed by a signal-conditioning circuit that you will build. You will need this circuit with the motor. The output of the signal-conditioning circuit is sampled by the ADC and processed by the microcontroller code.

a) You first must connect your Teensy to the necessary hardware. Place the microcontroller on a breadboard and make the following connections:

- Construct the low-pass filter using a $200k\Omega$ resistor and $0.82\mu F$ capacitor.
- Connect the input of the low-pass filter to pin 4 of the Teensy. Pin 4 has the PWM output signal. See the pinouts at https://www.pjrc.com/teensy/pinout.html
- The signal conditioning circuit in this case should a simple buffer amplifier. Use the LM324 op-amp, which has 4 single op-amps onboard. This seems excessive, but we'll use the other 3 later when we connect to the motor.
  - Connect the output of the low-pass filter to the input of the buffer.
  - Connect the output of the buffer to A9 (analog input 9) on the Teensy. Again, see the pinouts at https://www.pjrc.com/teensy/pinout.html
- Power the LM324 opamp from a $\pm 12V$ source

The appropriate components (LM324 opamp and $0.82\mu F$ capacitor) will be placed in the lab.

Build the circuit and test it in the lab. Verify that it is functioning properly. Please include a complete schematic in your report.

b) Now, we create the basic control code. The starting point is posted on the Canvas page. Here, I describe the key elements of the code:

Important Variables

- There are three key float variables in the code:
  - `desired`: This is a value that you can set from the GUI that is the reference command for your feedback system. This value is an integer between 0 and 255.
  - `direct`: This is a value that you can set from the GUI that will apply a direct command to the motor/RC circuit. This value is an integer between 0 and 255. **Use this value when the feedback is turned off (meaning $K_p = 0$).**
  - `Kp:` This is a proportional controller constant. It can also be set from the GUI.

Code Structure

The code is structured as follows:

- The code first sets up the microcontroller. This includes initializing the PWM, setting up the ADCs, and creating the basic serial-port communication needed to exchange data with the GUI.
- There is a control loop that runs every 1ms. This control loop will be modified by you to do the following:
    - Record a new ADC sample
    - Compare the reference, which is contained in the variable named "`desired`", to the value that was measured by the ADC.
    - Run the compensator code. Initially, this is just a proportional controller.
    - Check the output of the compensator to make sure that it's not too large or too small.
    - Update the duty ratio.

Let's look at some basic elements. The section below includes some key variable definitions that we will use to setup the ADC and PWM. Please note the following:

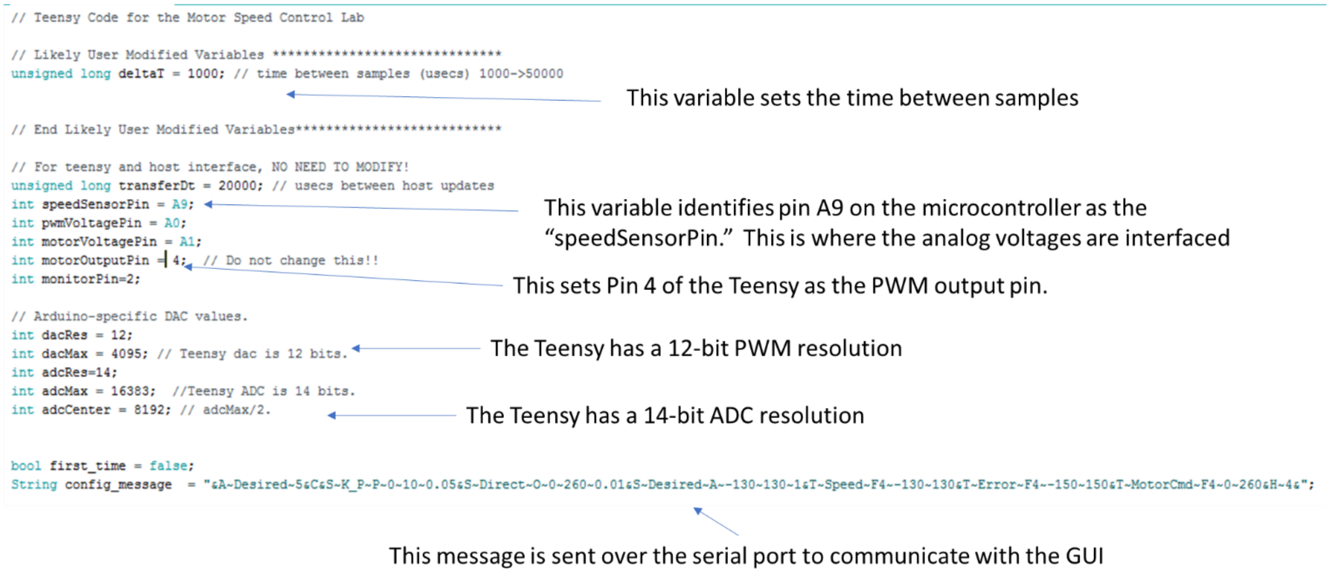
- The Teensy has an ADC with 14-bit resolution and a 3.3V range. This means that an input voltage of 1V would become an integer code like so:

$$round\left(1V\left(\frac{2^{14}}{3.3}\right)\right) = 4{,}965$$

  This integer value is manipulated inside the controller program you write.
- The Teensy has a PWM output with 12-bit resolution. This means that the duty ratio can be adjusted with 12-bit resolution. For example, if you set the duty ratio with the integer code 1024, the duty ratio will become:

$$\left(\frac{1024}{2^{12}}\right) = 0.25$$

- There are several additional elements in this code that we will not use. For example, there are values such as "motorVoltagePin", "monitorPin", etc. We will use these later.



```
// Teensy Code for the Motor Speed Control Lab

// Likely User Modified Variables ******************************
unsigned long deltaT = 1000; // time between samples (usecs) 1000->50000


// End Likely User Modified Variables***************************

// For teensy and host interface, NO NEED TO MODIFY!
unsigned long transferDt = 20000; // usecs between host updates
int speedSensorPin = A9;
int pwmVoltagePin = A0;
int motorVoltagePin = A1;
int motorOutputPin = 4;  // Do not change this!!
int monitorPin=2;

// Arduino-specific DAC values.
int dacRes = 12;
int dacMax = 4095; // Teensy dac is 12 bits.
int adcRes=14;
int adcMax = 16383;  //Teensy ADC is 14 bits.
int adcCenter = 8192; // adcMax/2.


bool first_time = false;
String config_message  = "&A~Desired~5&C&S~K_P~P~0~10~0.05&S~Direct~O~0~260~0.01&S~Desired~A~-130~130~1&T~Speed~F4~-130~130&T~Error~F4~-150~150&T~MotorCmd~F4~0~260&H~4&";
```

This variable sets the time between samples

This variable identifies pin A9 on the microcontroller as the "speedSensorPin." This is where the analog voltages are interfaced

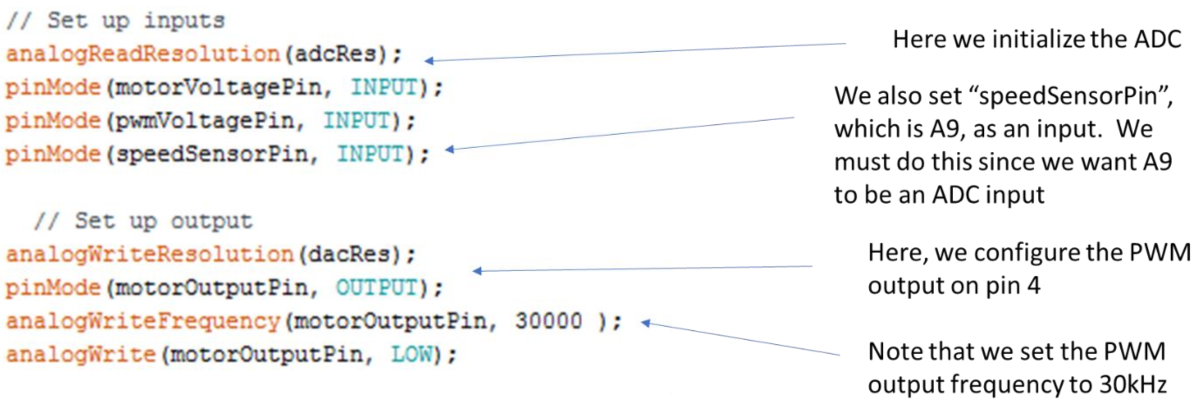
This sets Pin 4 of the Teensy as the PWM output pin.

The Teensy has a 12-bit PWM resolution

The Teensy has a 14-bit ADC resolution

This message is sent over the serial port to communicate with the GUI

Later in the code, we utilize the variables created above. For example:



```
// Set up inputs
analogReadResolution(adcRes);
pinMode(motorVoltagePin, INPUT);
pinMode(pwmVoltagePin, INPUT);
pinMode(speedSensorPin, INPUT);

  // Set up output
analogWriteResolution(dacRes);
pinMode(motorOutputPin, OUTPUT);
analogWriteFrequency(motorOutputPin, 30000 );
analogWrite(motorOutputPin, LOW);
```

Here we initialize the ADC

We also set "speedSensorPin", which is A9, as an input. We must do this since we want A9 to be an ADC input

Here, we configure the PWM output on pin 4

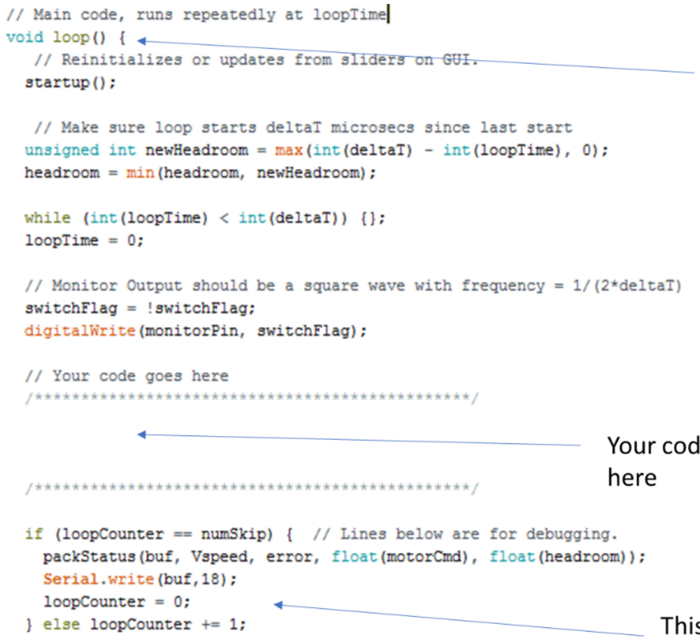
Note that we set the PWM output frequency to 30kHz

You also need to modify the control loop to perform its basic functions, namely:

- Sample the input voltage on A9
- Run the controller code
- Update the duty ratio

Even though the Teensy has a 14-bit ADC, we're only going to use 8 bits of resolution for compatibility with our GUI and server. You will have to think about how to make this conversion. The code may work with all 14 bits, but we've tested it with 8-bit resolution.

The section of the loop you need to modify is shown below



```
// Main code, runs repeatedly at loopTime
void loop() {
   // Reinitializes or updates from sliders on GUI.
  startup();

   // Make sure loop starts deltaT microsecs since last start
  unsigned int newHeadroom = max(int(deltaT) - int(loopTime), 0);
  headroom = min(headroom, newHeadroom);

  while (int(loopTime) < int(deltaT)) {};
  loopTime = 0;

  // Monitor Output should be a square wave with frequency = 1/(2*deltaT)
  switchFlag = !switchFlag;
  digitalWrite(monitorPin, switchFlag);

  // Your code goes here
  /************************************************/


  /************************************************/

  if (loopCounter == numSkip) {  // Lines below are for debugging.
    packStatus(buf, Vspeed, error, float(motorCmd), float(headroom));
    Serial.write(buf,18);
    loopCounter = 0;
  } else loopCounter += 1;
```

This is the main control loop

Your code will be placed here

This code is for serial communication with the GUI. We will not touch it.

Your code should be placed in the section noted above and should include the following steps:

- Step 1: Measure and filter the speed:
  - Write a single statement that stores the output voltage measurement in a variable named `Vspeed`. Use the following approach:
    - Create the variable `Vspeed`. `Vspeed` should be a float. As a hint, the code should look like the following:

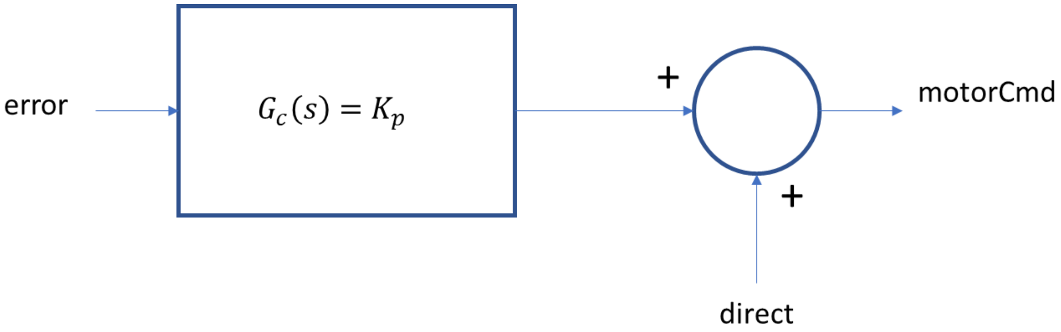
      ```
      float Vspeed = float ( REST OF CODE );
      ```

    - Use the `analogRead` command to sample the value from the pin named `speedSensorPin.` Google "Ardunio analogRead" to see how this function works.
    - To reduce noise, take three successive measurements and average them.
    - The ADC has a resolution of 14 bits. The PWM has a 12-bit resolution. This can create problems. Ideally, both should have the same resolution. In this case, we would like to reduce the resolution to 8 bits. To do this, we must divide the ADC output. By what number should we divide?
    - ***I recommend that you place all of the above operations into one line of code. This can easily be done, and I recommend it. Since our loop operates ever 1ms, we want to make sure that it completes quickly. If we have excessive operations and memory saves, we will slow down the code and it MIGHT affect the operation of the controller.***
- Step 2: Compute the error
  - To do so, create a float variable named `error` and compute its value from the other variables you have.
- Step 3: Apply the controller.

- To do this, create an integer variable named `motorCmd.` This variable will hold the duty ratio. Since the error is a float, you will need to typecast the result of the controller. To do so, your code should look like the following:

```
int motorCmd = int( YOUR CODE GOES HERE).
```

The resulting code should implement the following in the time domain:



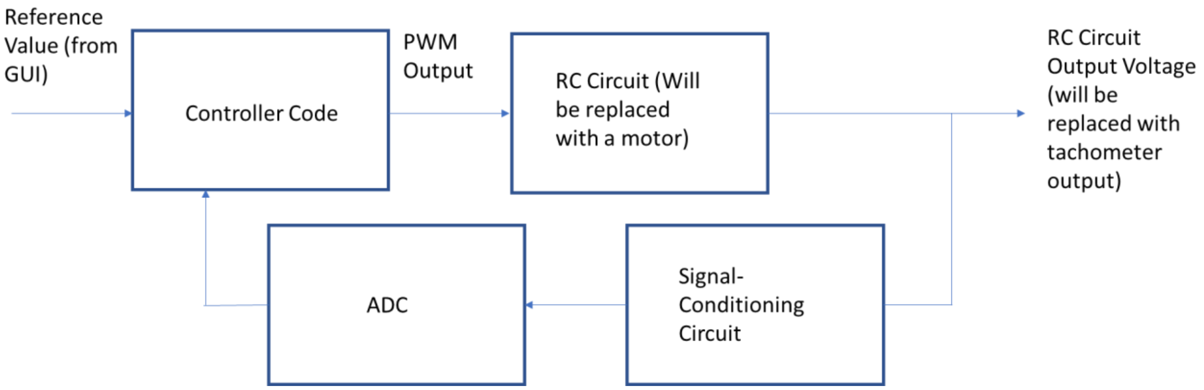
Note that I have asked you to include the variable `direct`, which allows you to drive the motor in an open-loop fashion when $K_p = 0$.

Why do we need to typecast the result?

- Step 4: Verify that the duty ratio is within the appropriate limits.
  - We have one last practical step. If the value of `motorCmd` represents a duty ratio less than 0 or greater than 1 (which can happen), our code will malfunction when we actually use it. We must therefore check to see if our 8-bit result in `motorCmd` is less than 0 or greater than 255:
    - If `motorCmd` is less than 0, we must set it to 0.
    - If `motorCmd` is greater than 0, we must set it to 255.

  I did this using the `min` and `max` commands. You can look these up by Googling the Arduino `min` and `max` commands.
- Step 5: Update the duty ratio.
  - In this case, use the Arduino `analogWrite` command to write `motorCmd` to `motorOutputPin`. Recall that we modified `motorCmd` to be an 8-bit value. Since the Teensy's PWM resolution is 12-bits, you must multiply `motorCmd` by an appropriate value first. What is this value?

c) Before running our code, let's model the system we have built. To do so, recall that the basic system is connected like so:


Reference Value (from GUI)
Controller Code
PWM Output
RC Circuit (Will be replaced with a motor)
RC Circuit Output Voltage (will be replaced with tachometer output)
ADC
Signal-Conditioning Circuit

Create a block diagram for the system. Recall the following:

- The input reference (which is the variable named `desired`) is an 8-bit integer value between 0 and 255.
- The combination of the signal-conditioning circuit and the ADC scales the output voltage to an 8-bit integer. Include the appropriate gain to model this.
- Be sure to model the PWM generator appropriately. In this case, you provide an 8-bit integer to the PWM generator and the output is square wave. We wish the represent the PWM generator using its averaged model. We discussed this previously in class and in Problem Set 4. The averaged model should describe how the 8-bit input to the PWM generator results to the average value of the PWM output waveform. Since the RC circuit is low-pass, the switching frequency and its harmonics should be filtered out. Since the Teensy produces a 0 to 3.3V PWM signal, you should be able to determine this.
- Model all of the other components as you would normally.

d) Now, compile your code. Start the Python server and then start the GUI. If your code is error-free, your browser window should include the following:

- The ability to change the following variables:
  - `desired –` This value can be any integer between 0 and 255.
  - `direct` – This value can be any integer between 0 and 255.
  - `Kp –` This value can be any floating point number between 0 and 20.
- The ability to view the following signals:
  - `error`
  - `motorCmd`
  - The motor speed (This is also an 8-bit value between 0 and 255).

Test your code and verify that it works as it should. I recommend an approach such as the following:

- Start with `Kp = 0.` Play with the value of `direct`. Verify that the speed moves to the value you would expect based on your model. Verify using the oscilloscope.
- Now, set `direct = 0, Kp` = 1, and `desired = 128`. Compare the expected steady-state values of the error and the speed with what you actually have. They should be very close. If there is any difference, check both your model and your code.

e) Manipulate your block diagram so that it is in the unity-gain feedback format. Draw the appropriate block diagram.

f) Use MATLAB to generate the Bode plot of the open-loop transfer function $L(s)$. Using this Bode plot, determine the value of K_p that will set the rise time of the closed-loop system to be about 2.6 times faster than the original, open-loop system. Set the value of K_p to this value. Reset the reference (`desired`) to 0 and allow the speed to reach 0. Now, suddenly change the reference (`desired`) to 128. Once you have verified this is working properly, use the single trigger capability of the oscilloscope to trigger on one step response. Make sure that you have a complete step response that includes:
   a. A small area before the step is applied
   b. A fair amount of steady-state conditions.

   Save the data to a CSV file using the process described on the Canvas page.

g) Generate the predicted closed-loop step response in MATLAB using commands such as the following:

```
t1_b = linspace(0,2,10000);

y1=step(CLOSED_LOOP_TF*DESIRED,t1_b);
```

   Using the process described on the Canvas page, load the csv file from part e. Use the plot command to plot the predicted step response (y1) and the actual step response (from the oscilloscope) on top of each other. They should almost sit on top of each other. If they do not, you are doing something wrong.

h) Let's now examine a key practical problem that many students do not appreciate. Set the gain so that the response will be 10 times faster. Apply a step input of 128. Use the single-trigger function to examine the output. Note that the response actually appears slower and not faster. Why is this happening? Explain by considering how `error` and `motorCmd` will evolve right after the step is applied.

i) Now, set K_p back to 1 and apply a step input of 128. Slowly increase the gain in steps of 1 from 1 to 10. For each value, compute the predicted steady-state error. Compare this to the value of the actual steady-state error observed in the GUI. They values should match fairly closely.

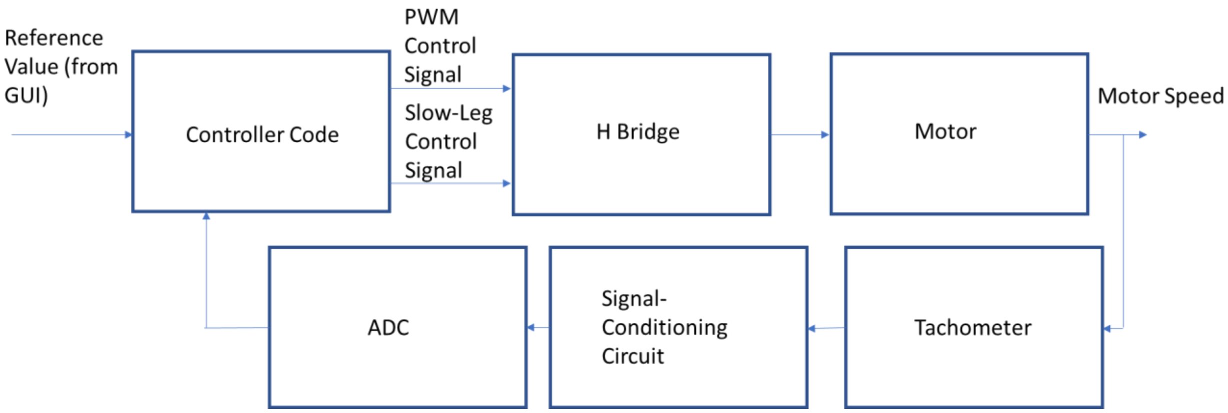
j) As you continue to increase K_p, you should note that the steady-state error continues to drop as expected. Note, however, that the value of `motorCmd` actually begins to fluctuate quite a bit. Why is this happening? You may want to think about the noise transfer function.

Problem 3

Now, we're going to modify our system to do the following:

- Connect to the motor
- Implement proportional-plus-integral control

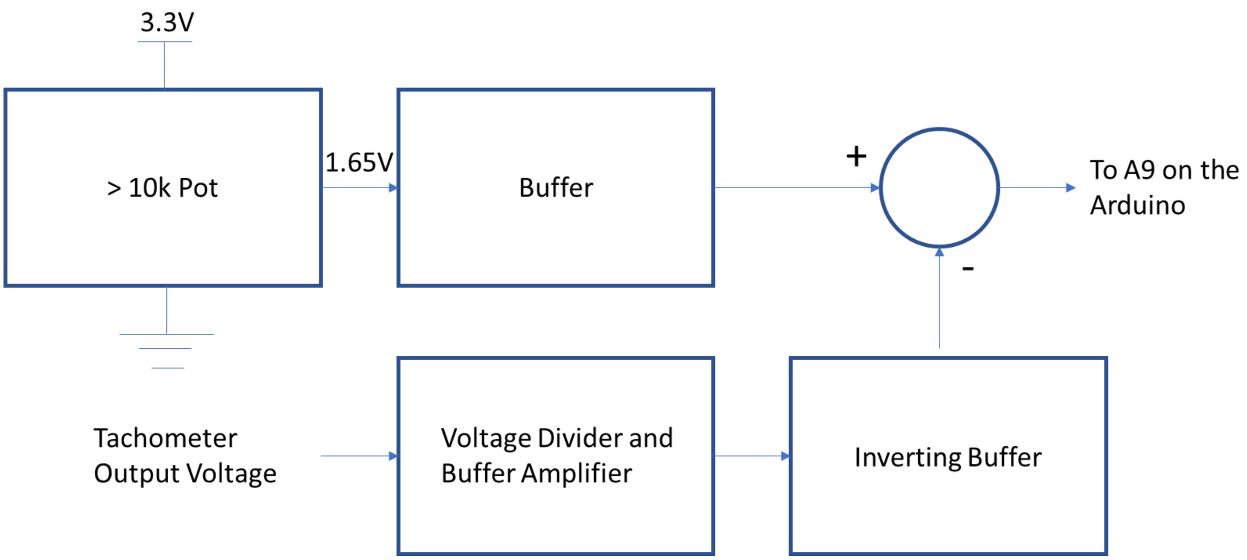
To interface to the motor, our system becomes slightly more complex. Here is the connection diagram now:



Here, we use the unipolar PWM drive method described in class. In this case, one leg of the H Bridge is provided a 30kHz PWM signal and the other is provided a signal that changes which switch is turned on based on the polarity of the `motorCmd` variable. Review the notes on Motor Drives to recall how this works.

a) You now need to modify the signal-conditioning circuit. The key connections are shown in the image below. We need this circuit for the following reasons:

    - If you apply a 100% duty ratio to the motor, the tachometer voltage will exceed 3.3V.
        - For this reason, we need the voltage divider.
        - I used a voltage divider with a divider ratio of about 0.386 using resistors that were each larger than $10k\Omega$
    - Since the motor can go in both the forward and reverse directions, the tachometer voltage can be either positive or negative. This is not acceptable for the Teensy, which has an ADC with an analog input range from 0V to 3.3V.
        - To overcome this problem, we level-shift or offset the tachometer output voltage so that when it reaches its maximum positive value it sends 3.3V to the ADC and when it reaches its maximum negative value it sends 0V to the ADC. This means that 0 speed will correspond to 1.65V. This level shift must ultimately be removed in the code.
        - I suggest you use a multimeter to adjust the voltage at the potentiometer output.
- The signal-conditioning circuit can be constructed using an LM324 opamp, which has 4 single op-amps onboard. Simply modify the circuit you built previously. Note the following:
    - The buffers should have a gain of 1
    - The subtractor should have a gain of 1
    - The inverting buffer should have a gain of -1
- The 3.3V should come from the 3.3V pin of the Teensy
- Power the LM324 opamp from a $\pm 12V$ source



More information on the Teensy pinout can be found here: https://www.pjrc.com/teensy/pinout.html

Also, the appropriate components (LM324 opamp and potentiometer) will be placed in the lab.

Build the circuit and test it in the lab. Verify that it is functioning properly. Please include a complete schematic in your report.

b) We now need to modify the code to work with this system. Download the second Teensy code starter from the Canvas page. You will modify this code. The basic functioning of the code is the same:

- Pin 4 (which we denote as the variable `motorOutputPin`) has the PWM signal for the fast leg of the H Bridge
- Pin 6 (which we denote as the variable `motorOutputPin` has the slow leg signal
- A9 is still the analog input. Now the analog input value will be about 1.65V when the motor is not moving.

We are also going to add an integrator. Because of this, the code has several additional float variables:

- `sum:` This is the running integral value. We update on each pass through the control loop.
- `Ki:` We multiply the integral output by the this value on each pass through the control loop.
- `maxSum:` This is the maximum allowed value of the sum variable. It is set from the GUI.

To get all of this working, you need to make a few changes in your code:

- Use a modified version of your measurement code from Problem 1. This modification should do the following:

    - Remove the offset from each measurement. Note that the variable `adcCenter` can be used for this.
    - In this case, the reference value (`desired`) can take values between -255 and 255. This means that the PWM has an effective 9-bit resolution. Thus, make sure to only reduce the 14-bit measurement output to 9-bits
- Compute the error just like you did in Problem 1.
- Compute the integrator output using the variable `sum` and the newest error value. Be sure to scale the error by the time. Since the look runs every 1ms, you must scale the result appropriately.
- Make sure that the value of `sum` remains less than `maxSum` and more than `-maxSum`. I used the Arduino min and max commands to do this.
- Compute `motorCmd` as a float as before. In this case, `motorCmd` should include the integrator output, the proportional term output and the `direct` command.
- Before updating the duty ratio using `motorCmd`, check to see if it's less than 0. If so, the controller wants to go in the reverse direction. If `motorCmd` is less than 0:
    - Make sure that it can't go below -255.
    - Use the `digitalWrite` command to set the `motorOutputRev` pin high. This turns on the top MOSFET in the slow leg.
    - As we discussed in class, the duty ratio values mean something different when attempting to go in reverse. Study the Motor Drive notes and determine how to appropriately update the duty ratio. Use this result to modify `motorCmd.`
    - Use the `analogWrite` command to update the duty ratio. Once again, make sure to multiply `motorCmd` by an appropriate value. Since the duty ratio in your code is an 8-bit value, scale `motorCmd` back to a 12-bit value before writing it to the pin. Apply the appropriate multiplication.
- If `motorCmd` is greater than 0:
    - Make sure that it can't go above 255
    - Use the `digitalWrite` command to set the `motorOutputRev` pin low. This turns on the bottom MOSFET in the slow leg.
    - Use the `analogWrite` command to update the duty ratio. Once again, make sure to multiply `motorCmd` by an appropriate value. Since the duty ratio in your code is an 8-bit value, scale `motorCmd` back to a 12-bit value before writing it to the pin. Apply the appropriate multiplication.

Once you have written your code, test it using the RC circuit.

When I test the code:

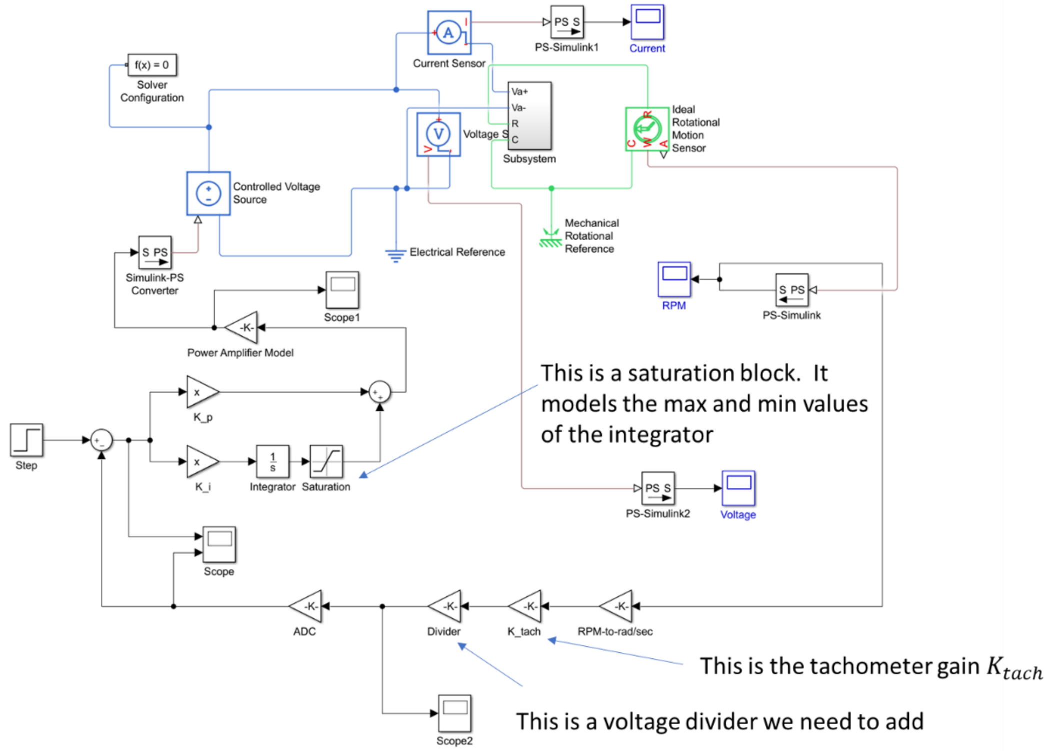
- I first turn off the feedback by setting K_p and K_i to 0. Adjust the potentiometer so that the ADC input is 0 when direct = 0.
- I adjust the value of `direct` to ensure that the window showing Speed matches the expected value.
- I then set `direct` to 0. I set K_p and K_i to 1. I then increase `maxSum` to 5000. Note that `maxSum` is initialized to 0. If you don't change this, the integrator will never integrate. I finally set the reference (`desired)` to 128. Since the integrator is part of the system now, the error should become 0. If it does not, the system does not work.

If all this works, your code should be ready to go.

Problem 4

Now, we're going to build a Simulink model and perform some basic compensator design.

a) Using the DC motor model from Design Problem 1, construct the new model shown below.

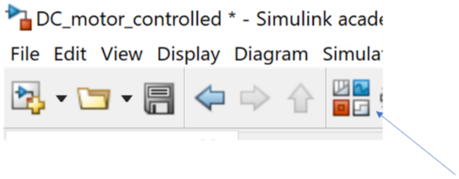


This is a saturation block. It models the max and min values of the integrator

This is the tachometer gain K_{tach}

This is a voltage divider we need to add

To add new components, open the Simulink Library Browser using this icon:


DC_motor_controlled * - Simulink acade
File Edit View Display Diagram Simula

The Simulink Library Browser

All of the items I've added are under the "Simulink" category. You can also copy-and-paste certain items from the existing workspace. A few notes:

- You need to add the voltage divider ratio in the block named "divider." Make sure that it matches your true voltage divider ratio.
- The power amplifier in this case is an H-bridge consisting of four MOSFETs. Since this is a real H-bridge, we expect it to have some losses. When switching MOSFETs on and off rapidly, there are small losses that result during the short turn-on and turn-off of each MOSFET. These are small in MOSFETs, but they can be extremely high in H-bridge chips such as the ones that are commonly used in your robotics and embedded systems

courses. For this reason, I have constructed a motor drive with MOSFETs that significantly reduces these losses. Still, they are not reduced to zero. I generated the data below. The left column is the `direct` input in my Teensy code, and the right column is the corresponding average voltage I measured at the output of the H-bridge.

Direct Input	Voltage
0	0
25	1.3825
50	2.78
75	4.248
100	5.697
125	7.149
150	8.6
175	10.051
200	11.501
225	12.952
250	14.4

Use the `linest` command in Excel to determine the slope of this curve.

- The saturation block here represents the fact that a real integrator can easily grow its output to either $+\infty$ or $-\infty$. In code this is a problem because we could potentially overflow our variables. To avoid this, we will force our code to limit the integrator output to a maximum value. Set the limits in this case to ± 5000.

Verify that your model is working properly by setting K_p and K_i to some appropriate values and simulating.

b) Create a block diagram of the system. The input should be the step, which is the value of the variable `desired`. The output should be the speed in RPM. Modify the block diagram to place it in unity-gain feedback form. For now, assume that the compensator has the form $G_C(s)$.

c) Use MATLAB to generate the bode plot of the uncompensated $L(s)$. You may make the assumption that the electrical pole can be neglected. If so, your phase margin and gain margin should both be infinite. Based on this graph

d) Show that the compensator $G_c(s) = K_p + \frac{K_I}{s}$ can be written as $G_c(s) = K\frac{\tau_z s+1}{s}$. Relate K and τ_z to K_p and K_I.

e) Design a compensator so that you achieve the following performance characteristics:
   - t_p=0.2706s
   - $PO = 13.69\%$

To do so, use the following procedure:

- Determine ζ and ω_n
- Choose a corresponding phase margin. Use the notes to recall how ζ relates to phase margin.
- Chose a corresponding ω_C. Use the notes to recall how speed of response relates to ω_c.
- Determine the gain required from the PI compensator to achieve the desired ω_c.
- Determine the phase required from the PI compensator to achieve the desired ϕ_m. From this, you can determine where to place the zero.

- Choose K_I and K_p

f) Run your simulation and verify that it approximately meets the desired t_p and PO values.

g) You are now ready to connect to the motor. **Since we only have one motor, you will need to sign up for a testing session. Make sure to follow the directions for connecting to the motor and DO NOT TURN ON the MAIN BUS POWER until:**

- **All of the other power has been applied**
- **The code is running.**
- **You have adjusted the potentiometer so that you have 1.65V into the ADC when the motor is not spinning. If you don't have this, the microcontroller will have the incorrect speed.**

Start the system by setting direct and desired to 0. Move K_p and K_i to the values you determined previously. Step desired from 0 to about 120. Before applying the step, reset maxSum to 0 and then back to 5000.

As before, save the data during a step response. In this case, save the ADC input on the oscilloscope. You will analyze this data later. For now, verify that your percent overshoot is approximately correct.

h) With the motor running, start to play around. Set K_I to zero. Set K_p to 0 and desired to 128. Place a "disturbance" on the motor. You can do this by lowering the magnetic brake over top of the rotating disc. What happens to the speed of the motor and error?

Now, turn on the integrator by setting K_i to a value greater than 1. After changing K_i, you should reset the sum back to 0 and then release it. With the integrator functioning, you should see the error is 0. Now, move the brake down over the rotor. What happens to the speed and error now?

i) With the integrator functioning and the brake released, continue to increase K_p. What do you notice about the output of `motorCmd?` Explain this behavior using the noise rejection properties of the system.

j) Why do we keep setting `maxSum` to 0? To understand why let `desired` be 0 and wait a while. Eventually, the motor will start to creep a little bit. Why is this happening?

k) Now, analyze your step response data. The data may be noisy. To remove the noise, I use a median filter. It will look like so:

```
v_x = medfilt1(v_x,11);

v_x  = v_x - mean(v_x(1:7000));
```

Note that I removed the offset at the beginning of the waveform. You may need to change the number of points that you subtract. The median filter is a special non-linear filter that removes impulsive noise such as what occurs from the H-bridge switching.

Your Simulink file saves the RPM values to a vector. I compared the expected and actual ADC inputs like so:

```
plot(t,v_x,RPM1.time,RPM1.signals.values*(2*pi/60)*K_tach*K_divider);
```

These values should sit on top of each other.

I then set direct to 0. I set K_p and K_i to 1. I then increase maxSum to 5000. Note that maxSum is initialized to 0. If you don't change this, the integrator will never integrate. I finally set the reference (desired) to 128. Since the integrator is part of the system now, the error should become 0. If it does not, the system does not work.