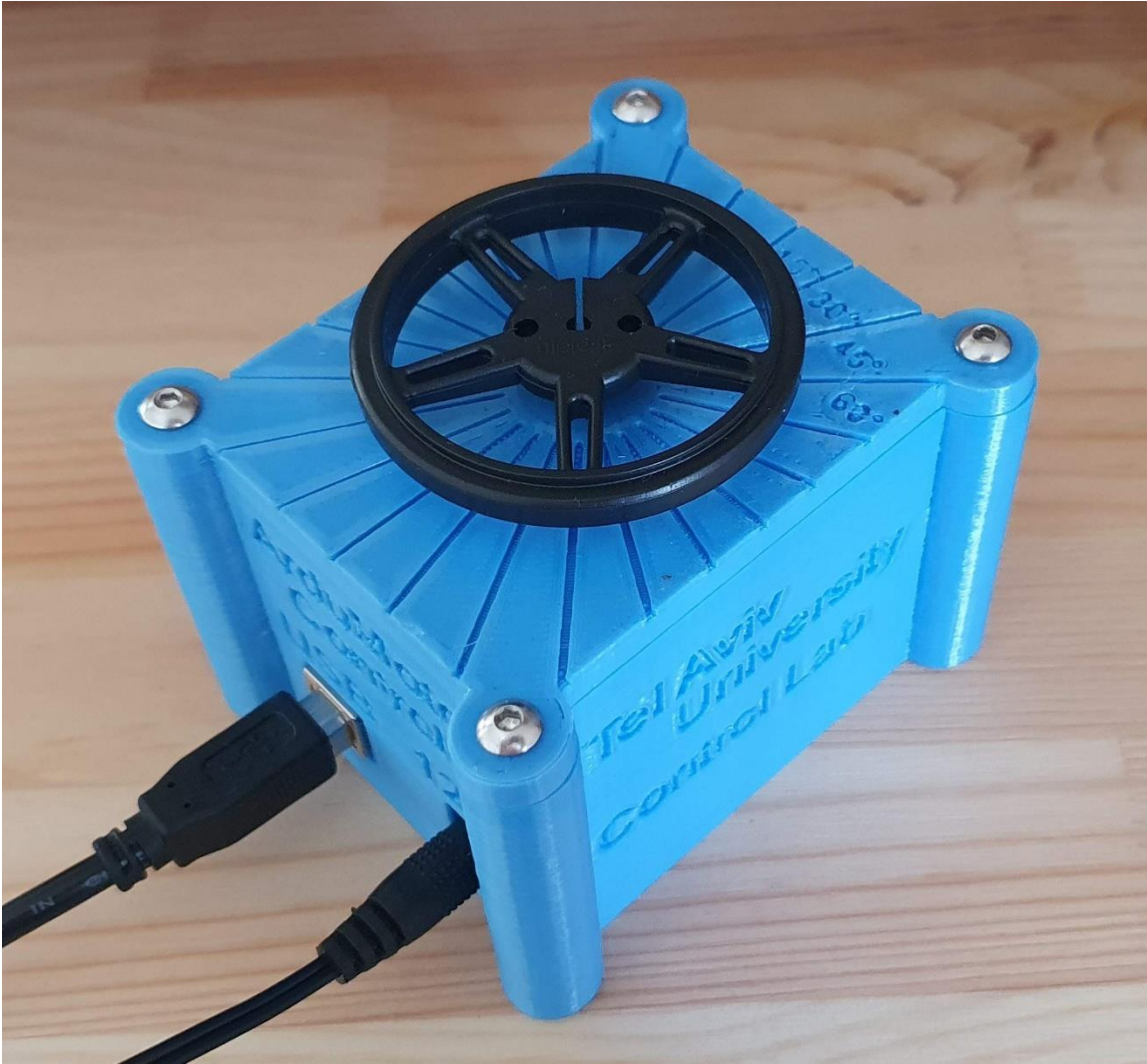


# Discrete Control Lab



## 1. Introduction

Modern control systems are realized using a micro controller unit (MCU) and implemented in code. Since an MCU is a discrete system, the control implementation is also a discrete control system. During this lab, you will get a taste of implementing a simple motor control system using an Arduino development board with the Arduino development environment (IDE).

## 2. System overview

The experiment setup consists of, Arduino Uno development board with a carrier shield for motor driver, a DC motor including a gearbox, encoder and a rotation mass. The control implemented in code using an Arduino development environment in C++. See literature review chapter for thorough description of the experiment concepts.

System components:

- Arduino Uno Micro Controller Unit (MCU):  
<https://store.arduino.cc/arduino-uno-rev3>
- Motor Driver Shield (H-bridge):  
<https://www.sparkfun.com/products/14129>
- DC Motor including Gear box 1:100:  
<https://www.pololu.com/product/3041>
- Magnetic Encoder with 3PPR (Pulses Per Revolution) :  
<https://www.pololu.com/product/3081>
- Rotating mass Flywheel:  
<https://www.pololu.com/product/1420>
- 12V Power supply:  
<https://www.4project.co.il/product/wall-adapter-power-supply-12v-1a>
- PC with an Arduino IDE Connected over USB cable:  
<https://www.arduino.cc/en/Main/Software>  
<https://www.4project.co.il/product/a2b-standard-usb-cable-1.8m>
- 3D Printed Case:  
<https://a360.co/3iXi0aZ>

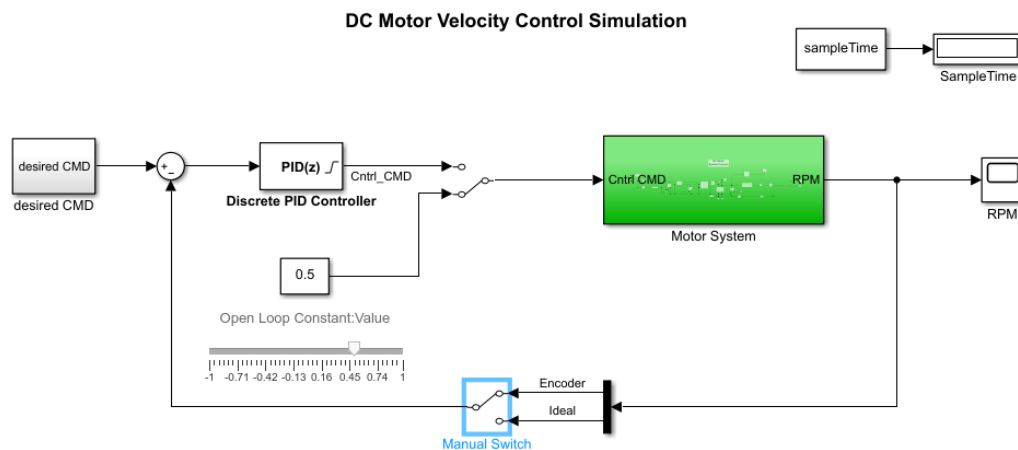


### 3. Discrete Control Part 1 – Simulation

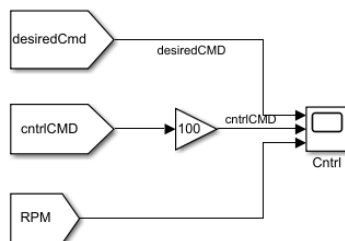
During this section we are going to study the concepts of a DC motor system and learn the implementation of a discrete velocity motor control using a simulation in Simulink. We will get familiar with the concepts of a sampled system and its influence on the control performance.

#### 3.1 Matlab Simulation review, Open loop.

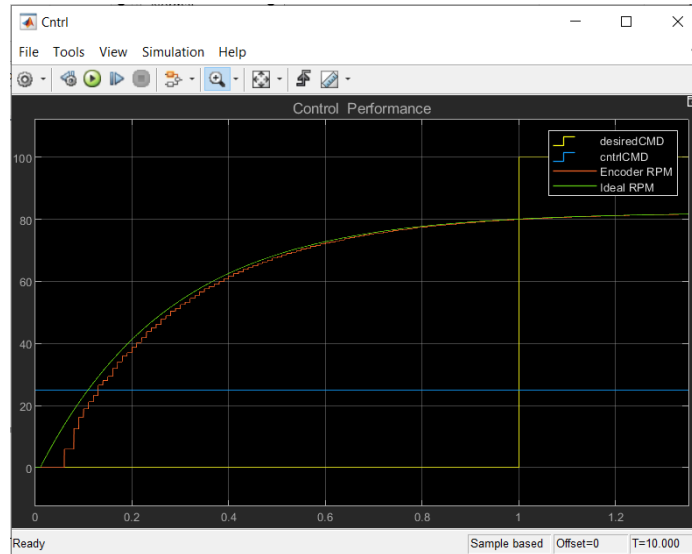
Start by opening Simulink model: **dcMotorControlSim.slx**, the model contains open and close loop implementation for a velocity motor control. The model uses an initialization script **dcMotorControlInit.m**, which sets the system sample time variable. Make sure it is present in the work folder with the simulation model.



#### Control Performance

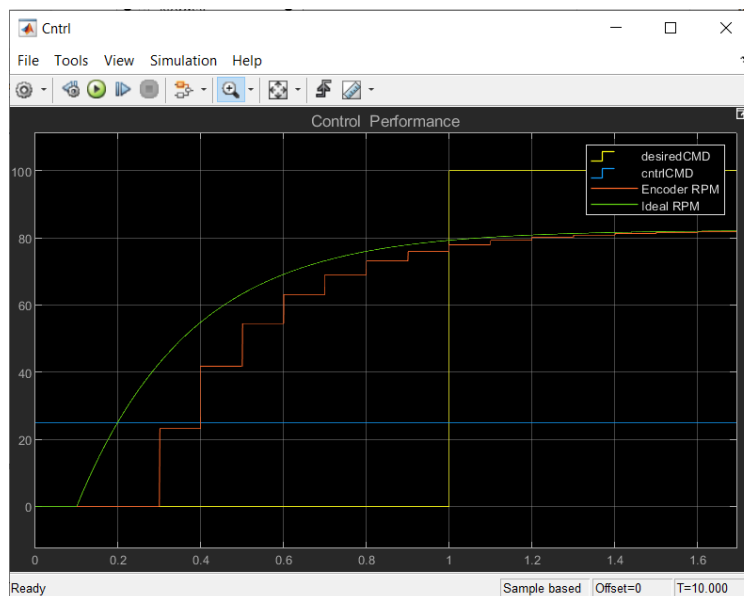


Run the model in open loop with various motor command values. You may change the stop time for the simulation to inf and change the values through the slider while the simulation is running. Note the motor RPM calculation as measured by the Encoder compared to the Actual RPM.



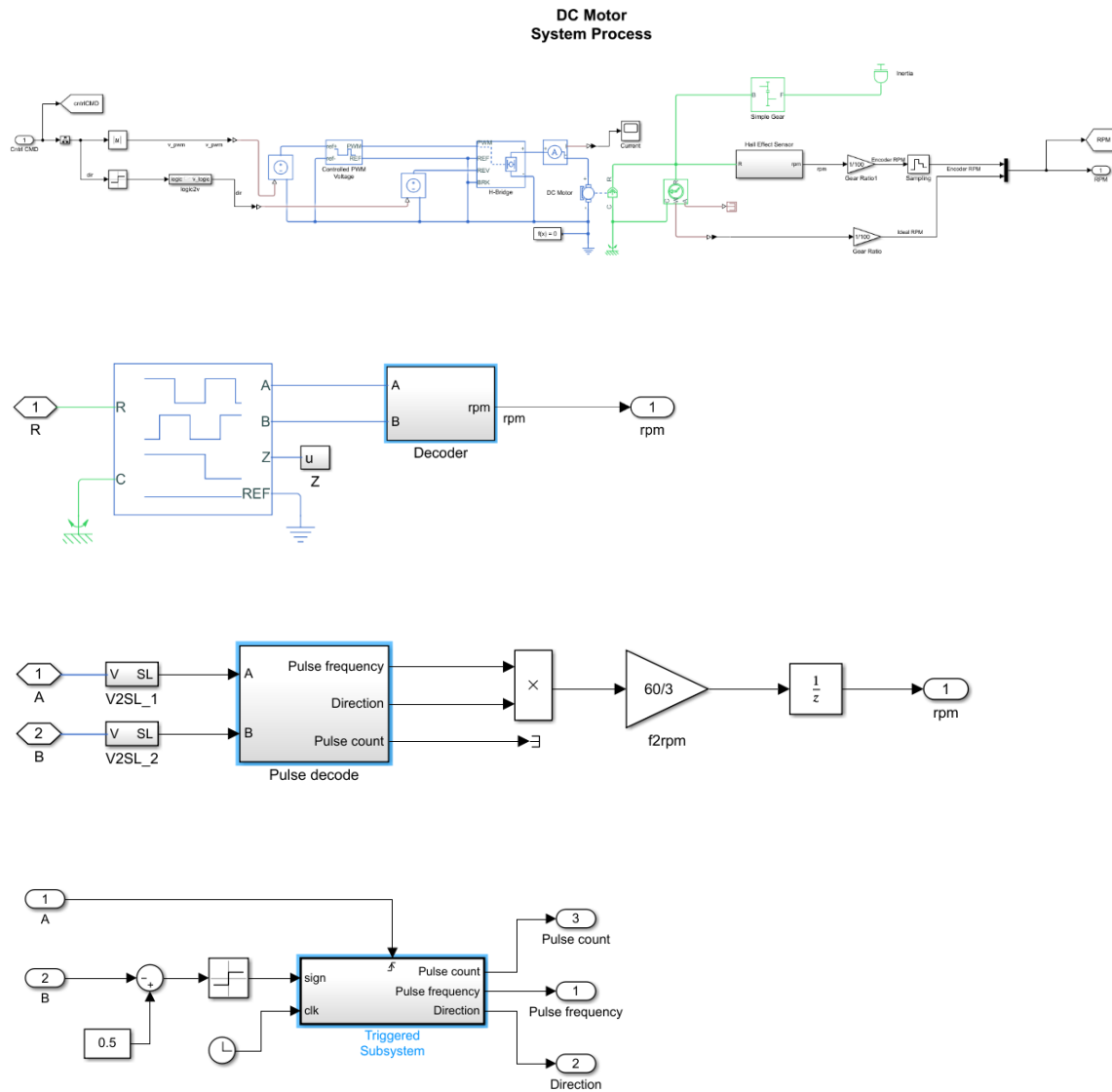
*Open loop with motor command at 0.25 Simulation sampleTime 0.01 sec*

Change the system sampleTime variable and note the influence on the Encoder RPM calculation. You can update the variable through the Command window.



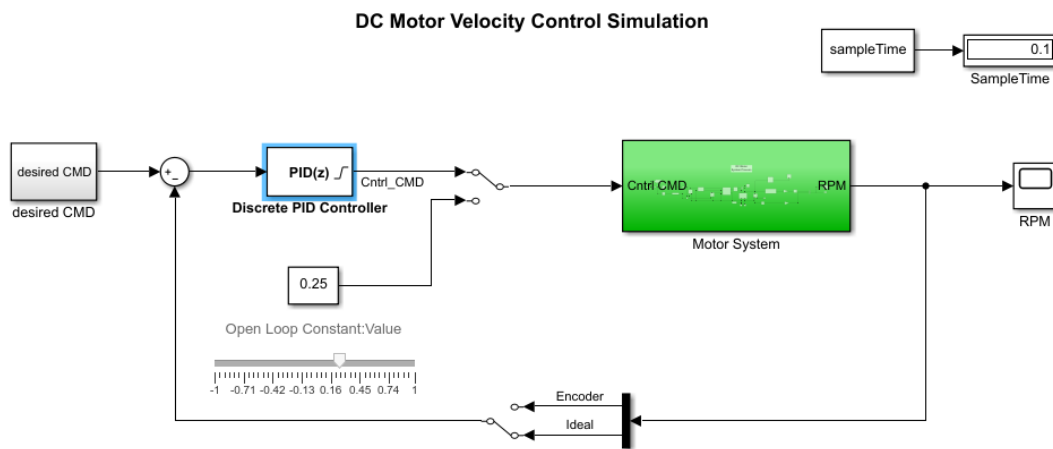
*Open loop with motor command at 0.25 Simulation SampleTime 0.1 sec*

Open the simulation subsystem named Motor System and review how it is implemented and it's subsystems. Explain how the encoder works and how the motor velocity is calculated. Hint it is implemented at dcMotorControlSim/Motor System/Hall Effect Sensor/Decoder/Pulse decode.



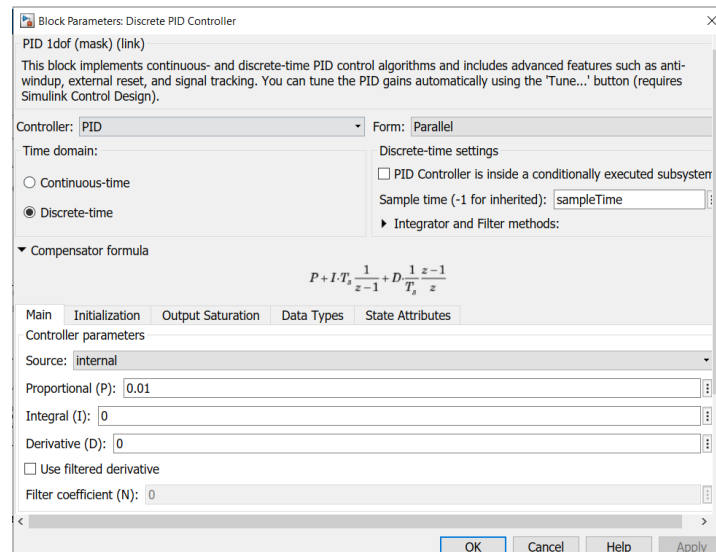
### 3.2 Close loop proportional control

For this lab we will consider the system as a black box - without a known continuous model, and learn how to manually tune the controller to achieve the desired system response. First route the model switches to use the signals from the PID controller output, and for the feedback the Ideal RPM value. As seen in the figure below. Make sure to change back the simulation sample time to 0.01 sec.

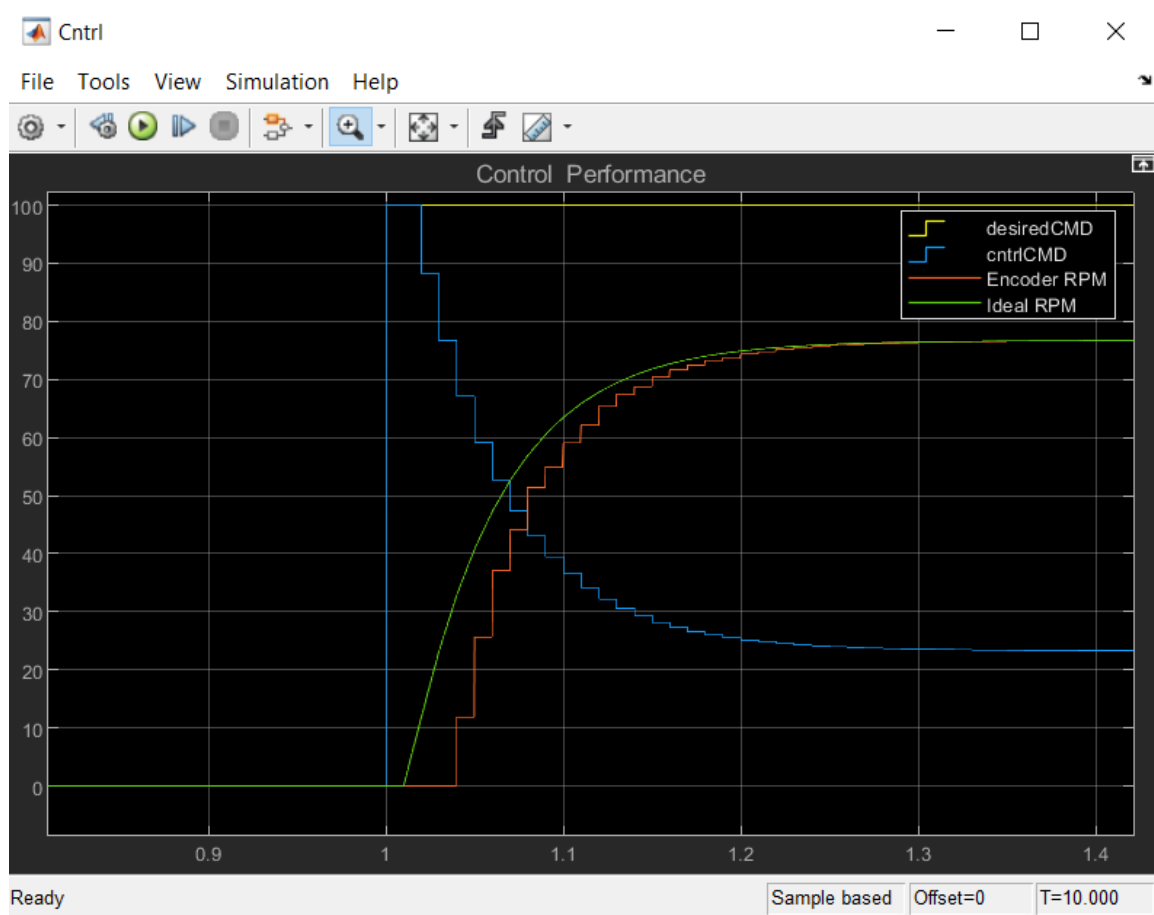


*dcMotorContrlSim.slx, set for PID feedback*

For the simulation a discrete implementation of the PID controller is used. Open the PID block and set it to a proportional gain only.

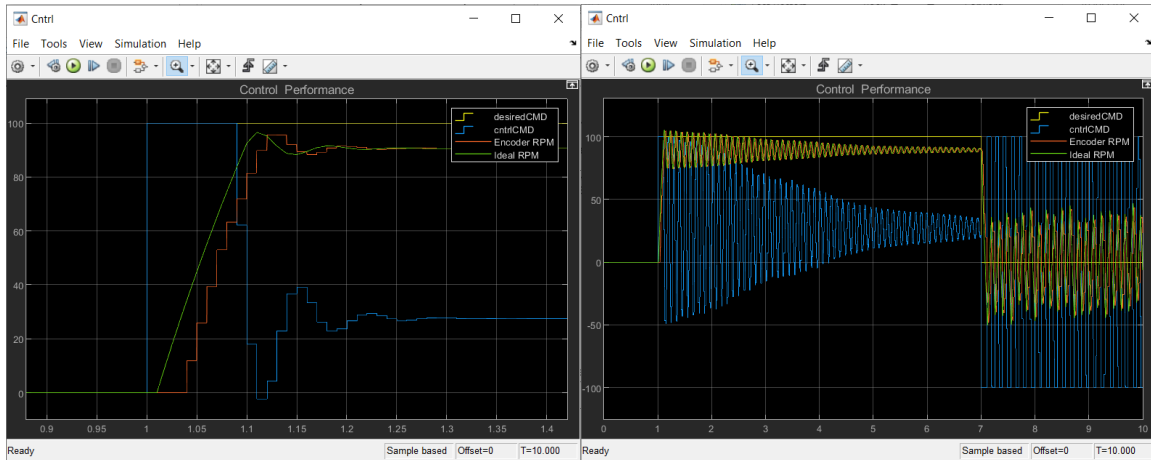


Evaluating the initial value for the proportional gain: at the previous section, we have used various motor commands values, which resulted in various motor velocities. From the model review section, you may have noticed that the PWM command is scaled by a value of 0 to 1. Zero for 0-duty cycle – Motor is off and one for 100% duty cycle – Motor is fully on. For a 0.25 value, we got a response of approximately 80 RPM. Meaning that a rational starting value for a step response of 100RPM will be at the proximity of 0.01 gain. The reasoning is that for a starting error of 100 we will get a starting motor command of 1 which is equivalent to 100% duty cycle. Run the simulation with various gains and review the response.



Close loop response Ideal RPM feedback,  $k_P=0.01$  SampleTime=0.01

Choose a proper response gain which results with a slight overshoot and switch to the Encoder feedback. What happened? The system has lost stability. Perform the experiment with various gains and review the system response you get. Change the sample time and see what happens, run the experiment at different settings (10 gains values and 5 sampleTime values ranging at 0.1-0.001sec) make conclusion for best gains at various sample rate.



*Ideal feedback vs Encoder feedback.  $kP=0.05$ ,  $sampleTime=0.01$*

Because the experiment is based on discrete system implementation and uses an encoder, we no longer may consider it as a linear system. Write down several rules of thumb concluded from the experiments for a proper proportional gain value.

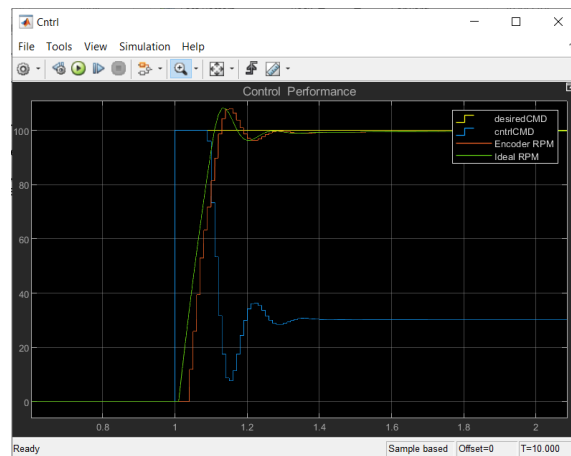
- Consider the rise time as a function of the control loop frequency ( $sampleTime$ ).
- Consider the Encoder resolution influence.



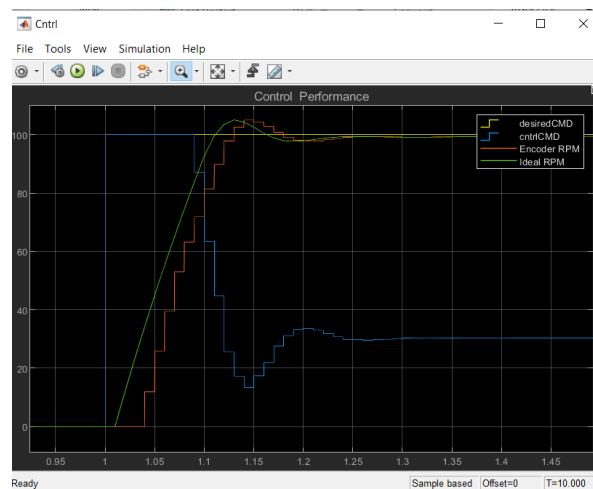
### 3.3 Close loop PID tuning

After setting a proper response with a proportional gain controller we want to improve the response by adding the integral and differential gains. Set the model to encoder feedback and sampleTime variable of 0.01sec

First, we will want to close the steady state error. Start by adding an integral gain at the scale of the proportional gain. Slowly increasing the gain and reviewing the system response. If the system loses stability, you may consider to slightly reducing the proportional gain. Tune the system until you get a rise time of 0.1 sec overshoot of up to 10% and settling time of 0.2 sec.



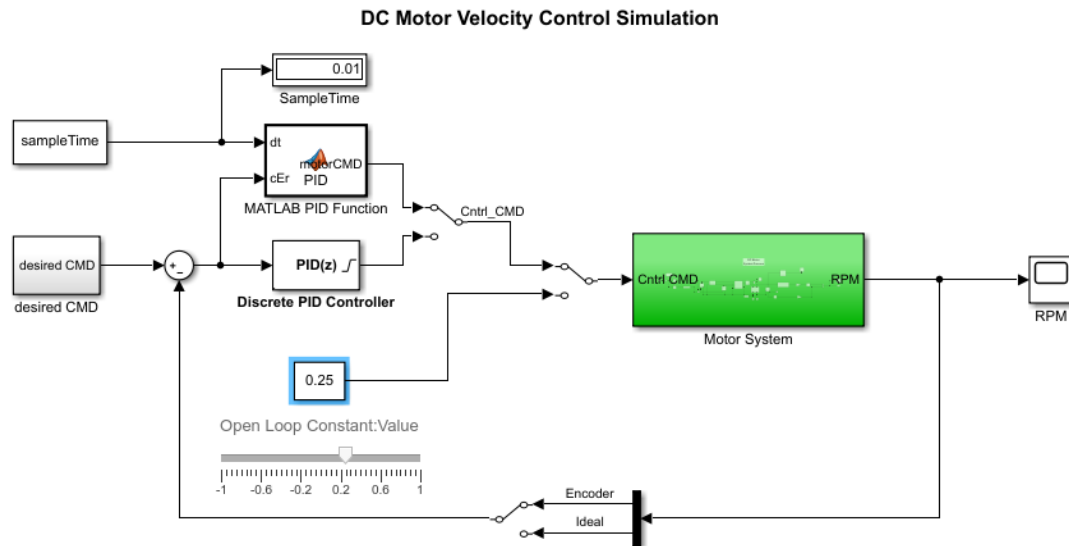
The differential gain isn't mandatory for a velocity control loop of a dc motor but you may consider adding it to slightly reduce the system overshoot. A good starting point will be at the range of  $k_p \cdot \text{sampleTime}$ .



You may have noticed that the controller loses stability at low velocity, which isn't an issue when using an ideal feedback. Write your thoughts for the reasons that may happen. For the actual experiment we will be using a slightly improved method for velocity estimation based on the encoder feedback as such it won't be such a big issue as in the simulation.

### 3.4 PID code implementation

Open Simulink model **dcMotorControlSimPIDcode.slx**. Note that it has a Matlab function for implementing the PID control in code. Route the system Control command to the Matlab PID function. It has implementation of proportional gain. Review the code and complete it for the PID implementation. Compare the response between the Simulink block and the code implementation for the same PID values.



```
function motorCMD = PID(dt,cEr)
% persistent variables persist between function calls. similar to
% static variable declaration in c
persistent lEr cdEr ciEr;
if isempty(lEr) %initialize variables for first run of the code.
    lEr = 0; % previous error initialization
    cdEr = 0; % differantional error initialization
    ciEr = 0; % integral error initialization
end
% control coefficients
kp = 0.025;

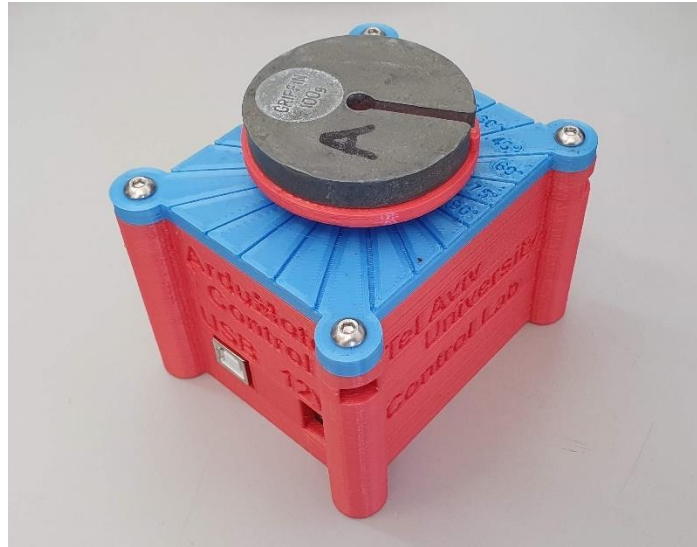
% clip integrator error
if (ciEr > 10)
    ciEr = 10;
elseif (ciEr < -10)
    ciEr = -10;
end

% update control command
motorCMD = kp*cEr;

% clip motor command
if (motorCMD > 1)
    motorCMD = 1;
elseif (motorCMD < -1)
    motorCMD = -1;
end

% update last error variable before next iteration
lEr = cEr;
```

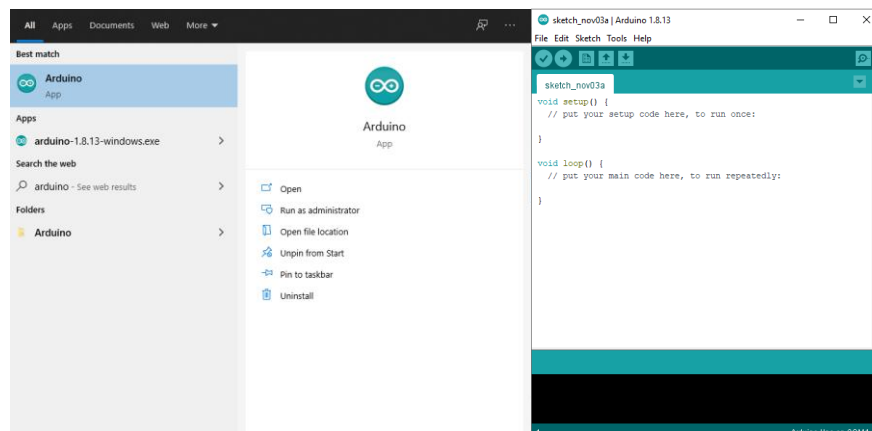
## 4. Discrete Control Part 2 – Hardware



During this section we are going to implement the control laws using the Arduino IDE in C++ and monitor its response using the embedded tool in the IDE. Since most of the modern control systems use a micro controller for motor control this lab will be a good peek into the actual implementations of the learned theories during the classes.

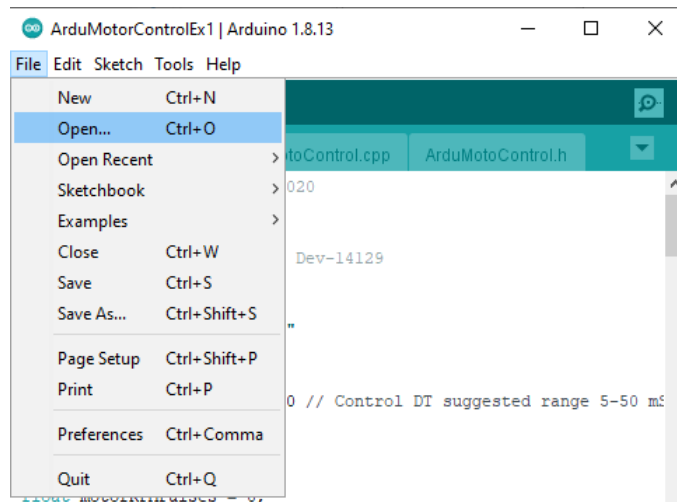
### 4.1 Getting started with Arduino / Arduino IDE

Open Arduino IDE, Either use the desktop icon, or search for the application.



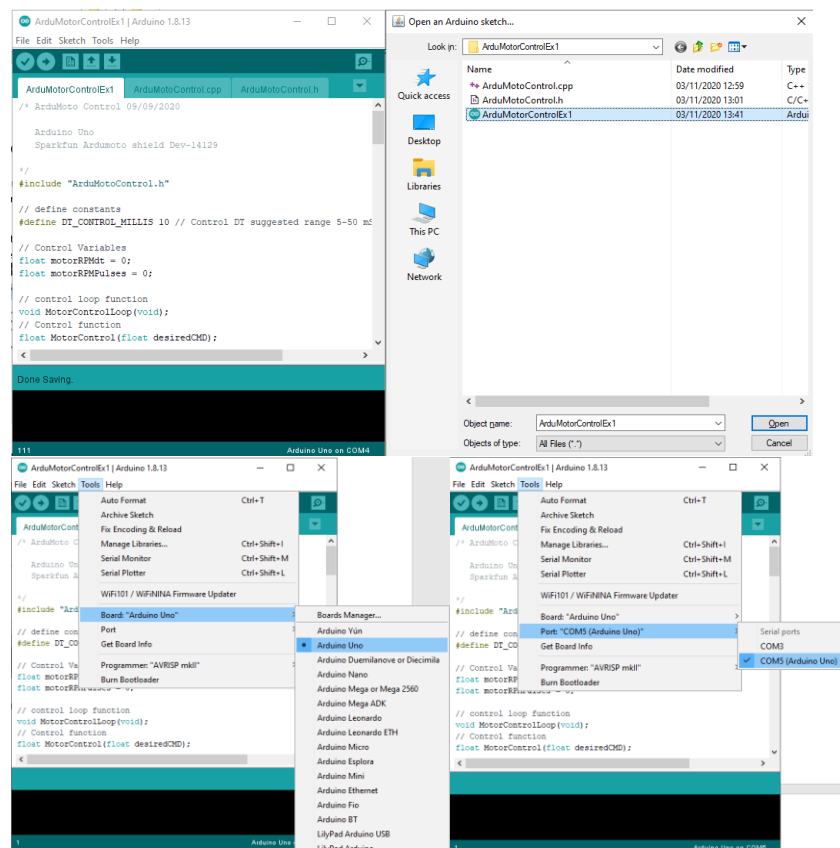
Connect the USB cable to the ArduMotoControl Box, You may notice that the motor starts to spin, that is because it is implementing the last code that was uploaded to the micro controller, Make sure the 12V power plug is also connected. When you finish working with the setup disconnect the power supply and the USB cable.

We will start by opening the first task and uploading the code to the micro controller. Select: File→Open Navigate to **ArduMotorControlEx1.ino** and open it.

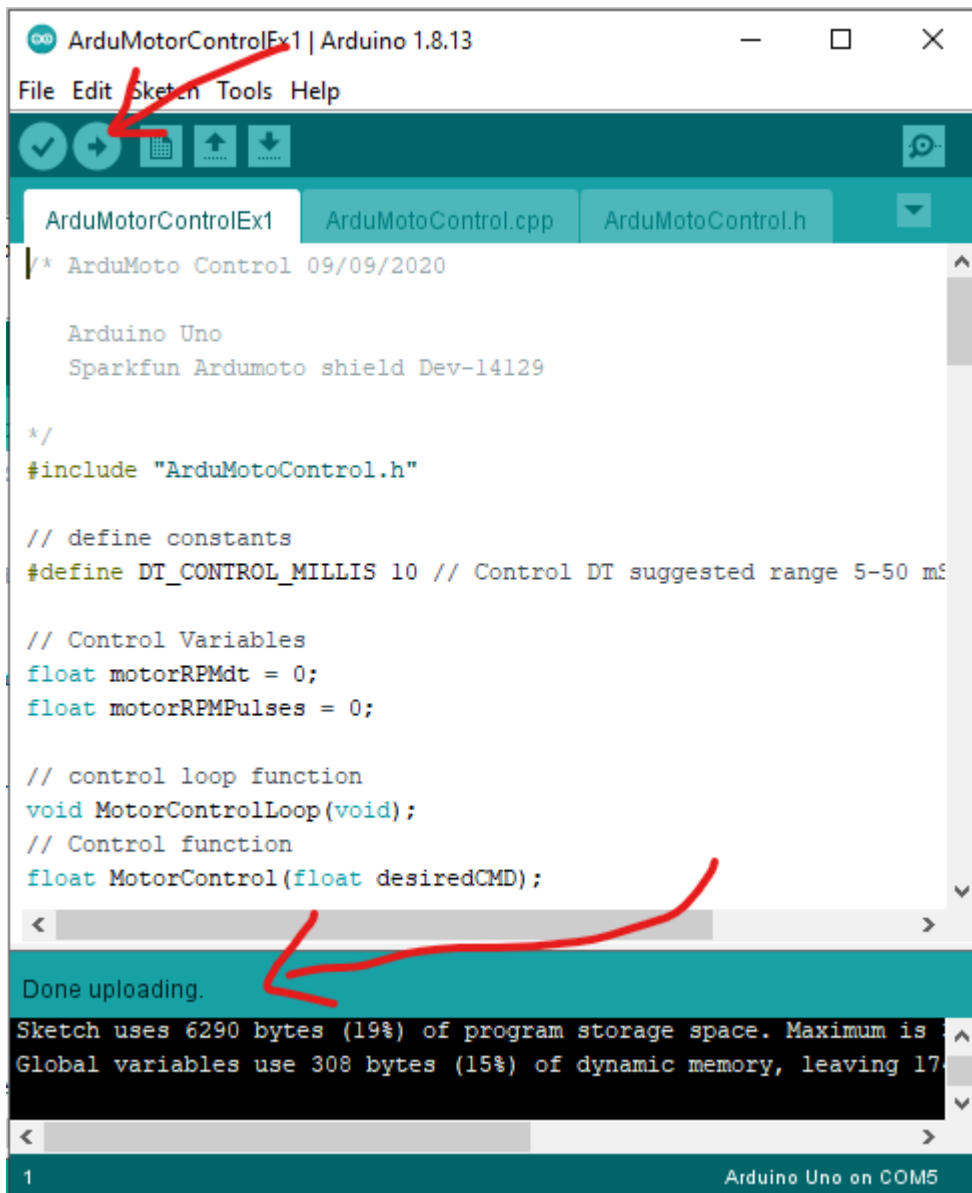


Connect to the Arduino Uno micro controller and make sure the correct port is selected. The COM number may change between computers.

- Select: Tools→Board→Arduino Uno,
- Select: Tools→Port→COMX (Arduino Uno),

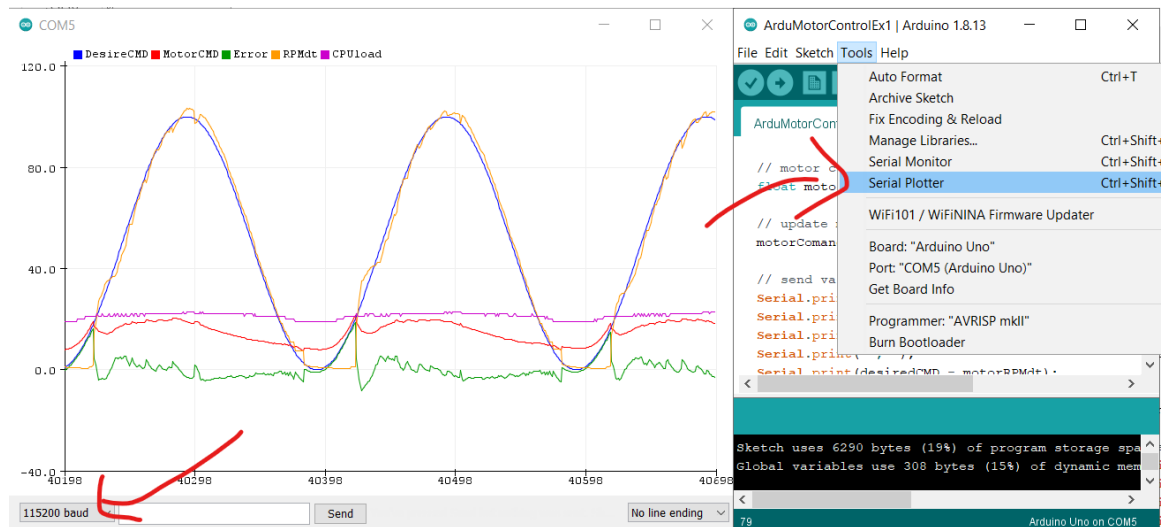


Upload the example to the micro controller by pressing upload button→. If everything is successful than you should see the “Done uploading” message and the motor will start rotating with a Sine Profile.



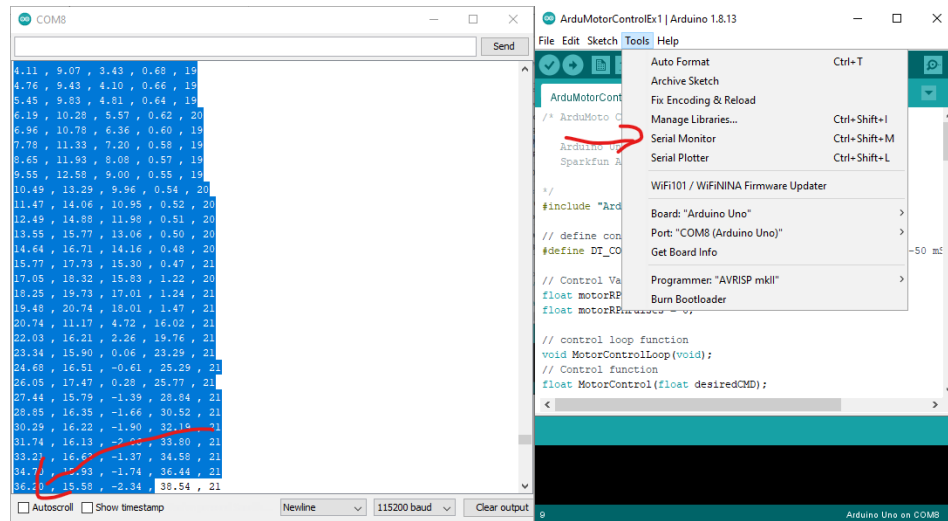
View the System response using the Serial Plotter tool, Make sure the baud rate is set to 115200 otherwise you get gibberish on the screen instead of the desired data. If everything has started correctly you will see the legend of the data on top of the graph, if it is mixed due to incorrect baud rate or some other start up issue you may reopen the Serial Plotter. It will reset the controller and start correctly.

- Tools → Serial Plotter

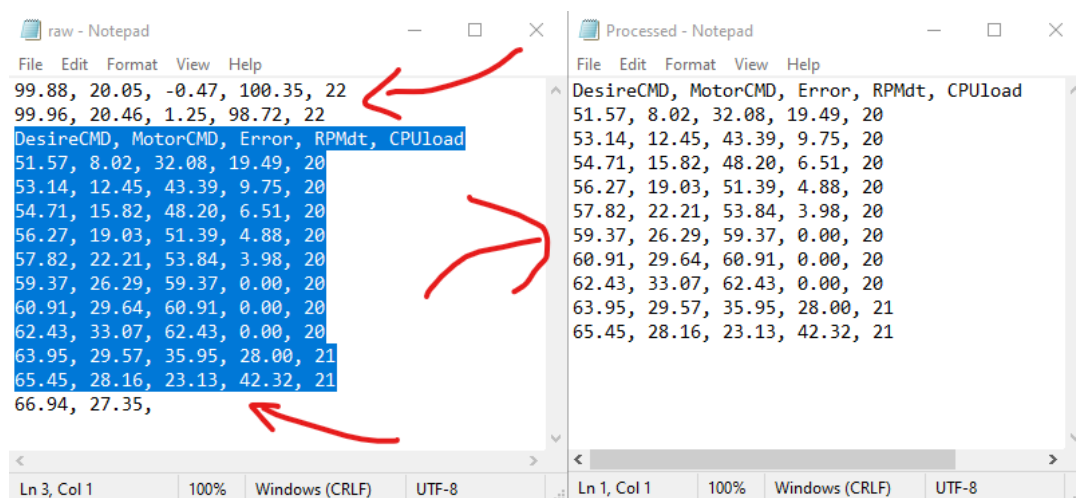


Sometimes during the lab you will be required to record a series of data, for that we will use the Serial Monitor tool. Each time you open the Serial Monitor or the Serial Plotter tool it will reset the micro controller and restart the latest code uploaded to it. When you open the Serial Monitor you will be able to see the streamed values, make sure the baud rate is correct and in order to make a recording deselect the Auto-scroll feature. Select the data set by pressing CTRL+A and CTRL+C, Past the data set into a text file you have created on the pc.

#### - Tools→ Serial Monitor



You may notice that the recorded data has unfinished data at the bottom of the data set and some values prior the variable names, which are of a previous session (prior reopening the Serial Monitoring tool). In order to make it easier to import the data into Matlab we will preprocess the data set by deleting all the values prior the variable names and the last unfinished line.



At this point importing the data into Matlab is easy using the Matlab Import Data tool. Press the Import Data option select the processed file, make sure the Comma separating is selected and import the samples to the workspace. To fast preview the recorded samples use the Matlab stacked plot option. Right click on the data set in Matlab select Plot Catalog and choose the stacked plot option.

The image is a composite of two screenshots from MATLAB R2018b, illustrating the process of importing data and creating a stacked plot.

**Top Screenshot: Import Data Wizard**

- File:** C:\Users\arkadira\Documents\ArduMotorControl\StudentFiles\ArduMotorControl
- Import:** C:\Users\arkadira\Desktop\Processed.txt
- Column delimiters:** Comma (selected)
- Range:** A2:E11
- Names Row:** 1
- Output Type:** Table
- Import Selection:** Import (checked)
- Standard Delimiters:** Comma (checked), Tab, Space, Semicolon
- Suggested Delimiters:** CommaSpace
- Custom Delimiters:** Enter Custom Delimiter

**Bottom Screenshot: Plot Catalog and Workspace**

- Plot Catalog:** Stacked Plot (selected)
- Workspace:** DesireCMD (selected)
- Right-click Context Menu:** Plot Catalog (selected)

**Data Table (from top screenshot):**

DesireCMD	RPMdt	CPUload
1	19.03	20
2	51.57	20
3	53.14	20
4	54.71	20
5	56.27	20
6	57.82	20
7	59.37	20
8	60.91	20
9	62.43	20
10	63.95	21
11	65.45	21



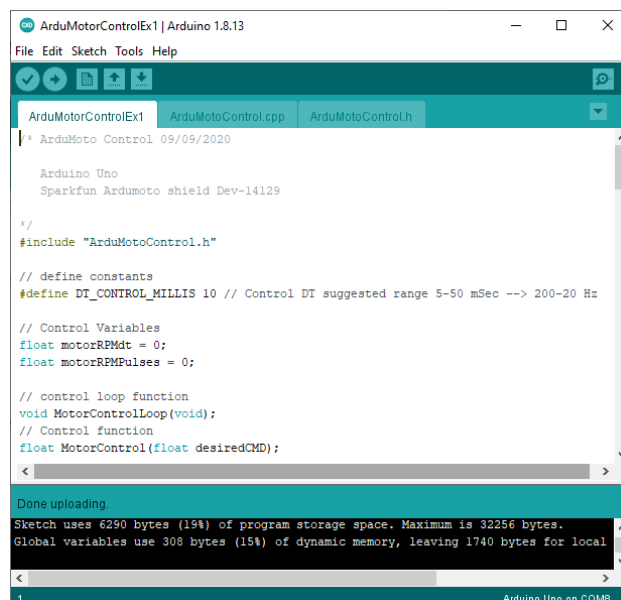
## 4.2 Code Review

The Experiment is implemented in C++ in order to handle the control section we will review the general approach for the code. When opening the ArduMotorControlEx1 you may have noticed the added files ArduMotoControl.cpp and ArduMotoControl.h those are the library files for the experiment and they have the implementation of the functions available at the main file ArduMotorControlEx1 including the motor driver handling and the encoder reading. You may review them and the comments for each function but for the experiment we will focus at the code in the main file.

The file Starts with the #include line for the library following some variables and function declaration.

- DT\_Control\_Millis – Sets the control interval for the experiment similar to the simulation. Take care not to exceed the recommended settings as not to overload the CPU. (Nothing will break but the controller will lose its stability)
- motorRPMdt – Is a global variable for the motor velocity based on the time interval between encoder pulses.
- motorRPMpulses – Is a global variable for the motor velocity based on the number of pulses occurred between the calls of the readRPMEncoder() function
- MotorControlLoop – Is a function which is called every controlled interval set by the DT\_Control\_Millis.
- MotorControl – Is the actual function for the Control implementation called inside the MotorControlLoop function.

Note the sections which start and end with the comment Student Code / Student Function, they indicate the sections which are available for modification so as not to harm the general flow of the rest of the code.



```
ArduMotorControlEx1 | Arduino 1.8.13
File Edit Sketch Tools Help

ArduMotorControlEx1 ArduMotoControl.cpp ArduMotoControl.h

/* ArduMoto Control 09/09/2020

Arduino Uno
Sparkfun ArduMoto shield Dev-14129

*/
#include "ArduMotoControl.h"

// define constants
#define DT_CONTROL_MILLIS 10 // Control DT suggested range 5-50 mSec --> 200-20 Hz

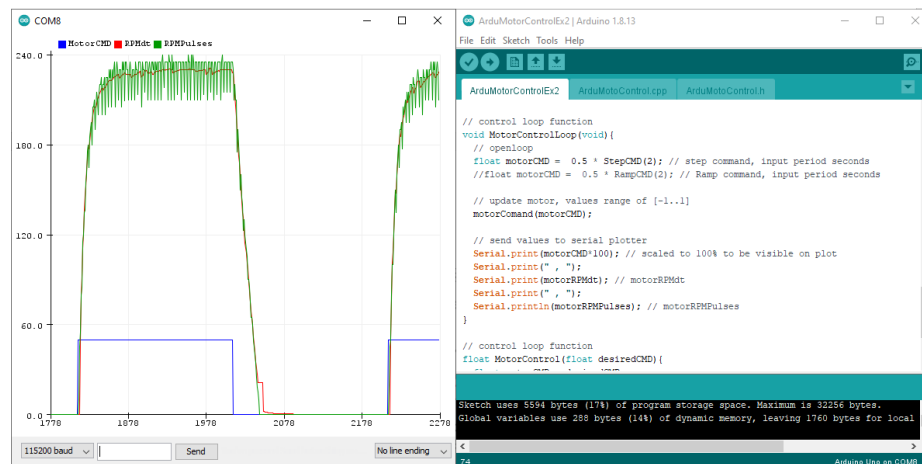
// Control Variables
float motorRPMdt = 0;
float motorRPMpulses = 0;

// control loop function
void MotorControlLoop(void);
// Control function
float MotorControl(float desiredCMD);

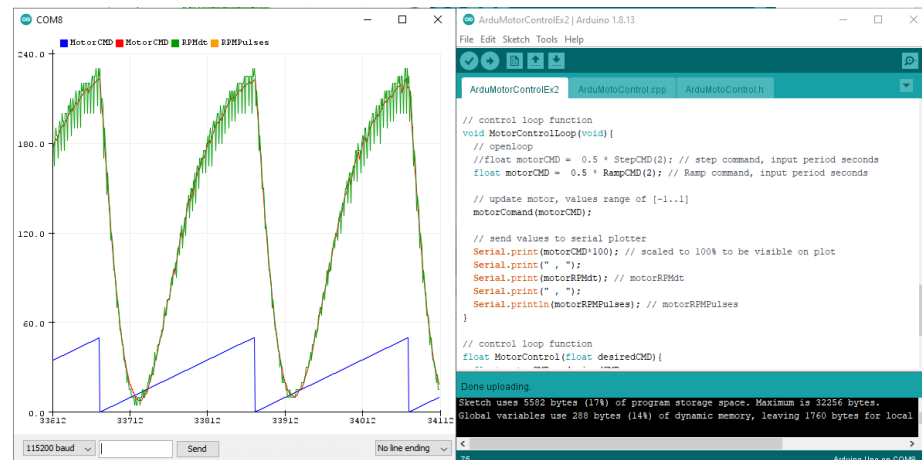
Done uploading.
Sketch uses 6290 bytes (19%) of program storage space. Maximum is 32256 bytes.
Global variables use 308 bytes (15%) of dynamic memory, leaving 1740 bytes for local
1 Arduino Uno on COM8
```

### 4.3 Open loop velocity response & Encoder readings

For this section we will evaluate the system response and understand the limitation of working with a cheap encoders, Start by Opening the **ArduMotorControlEx2.ino** sketch in the Arduino environment. Review the code and find the Open loop command inside the MotorControlLoop() function. We will work with a ramp command and step command function which are implemented in the library, the input to the function is the time period and the output is a command in the range of [0..1]. Verify that the StepCMD() function is uncommented with a gain of 0.5 and the RampCMD is commented and upload the code. Open the Serial Plotter and view the system response. The plot contains the desired command, the velocity measurement using the two approaches, time between encoder pulses, and number of pulses per function call RPMdtt, RMPulses.



Change the command to the rampCMD by uncommenting the RampCMD function and commenting the StepCMD function. Upload the updated code to the micro controller and view the response using the Serial Plotter.



Modify the code by changing the gains and the time period generating various commands and view the responses you get. In addition change the control frequency by modifying the value of DT\_CONTROL\_MILLIS.

- What is the minimal command for the motor to start moving?
- What is the maximum velocity?
- What can you say about the two methods for velocity measurements? Which is better at high speeds and which for lower speeds, how the control frequency influences the results?

### **In the final report:**

Prepare a recording for analysis in Matlab of a ramp response with a period of 5 seconds, and gain of 0.25, at two control periods, 10ms and 100 ms. Generate plots comparing the measured RPM of both methods.

- A plot containing RPMPulses at the two measured control intervals
- A plot containing the difference between RPMPulses and RPMdt as a function of the motor command for both control intervals.

Settings:

```
float motorCMD = 0.25 * RampCMD(5);
```

```
#define DT_CONTROL_MILLIS 100
```

Prepare a recording of a step response at a gain of 0.25 and period of 2 seconds. Plot the response in Matlab and mark the rise time of the system.

Settings:

```
float motorCMD = 0.25 * StepCMD(2);
```

```
#define DT_CONTROL_MILLIS 2
```

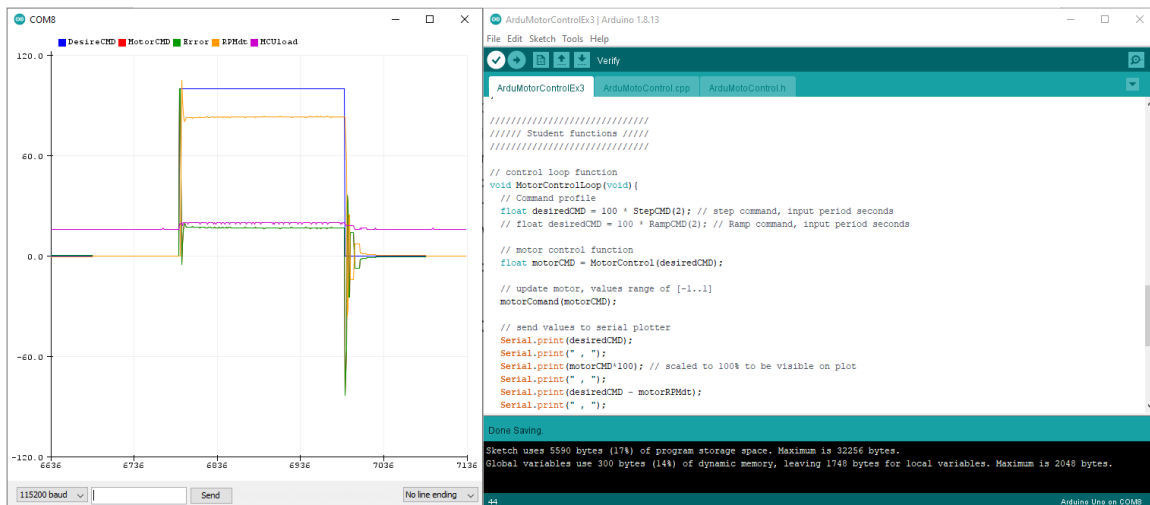
#### 4.4 Close loop proportional velocity control

For this section we will implement a simple proportional control command and tune it's gain. Start by Opening the **ArduMotorControlEx3.ino** sketch in the Arduino environment. Review the code and find the MotorControl() function. Understand the controller implementation and start with a logical value for the proportional gain – Remember the concepts described at Part 1 - Simulation.

Run a step response of 100RPM with the selected gain and view the response using the Serial Plotter tool. Set the control interval value at 10 milliseconds.

```
float desiredCMD = 100 * StepCMD(2);
```

```
#define DT_CONTROL_MILLIS 10
```



Change the DT\_CONTROL\_MILLIS value to 50 and see what happens. Explain the behavior and adjust the gains until you reach a stability. Find the stable gains for DT\_CONTROL\_MILLIS of 10,25,50 milliseconds. Record each of the responses you get with the final coefficients for each of the control interval for later analysis in Matlab.

**In the final report** generate a plot containing the 3 step responses generated previously and determine the rise time for each.

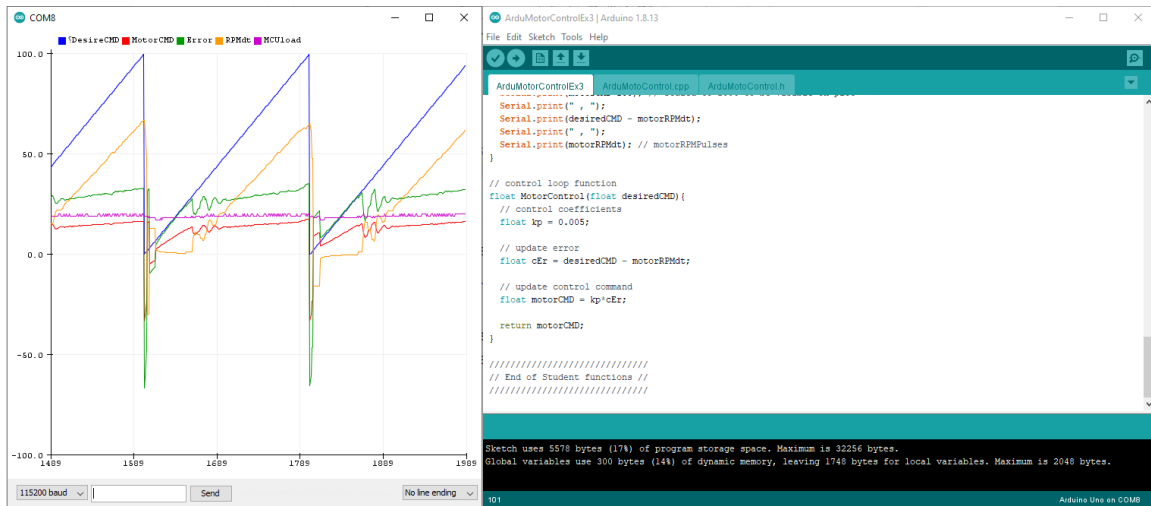
Up until now we have tested the proportional gain for a specific velocity of 100RPM, in order to understand the potential stability for various speeds we will use the rampCMD() function. Run a ramp response at 100RPM with 10 milliseconds control interval. With the stable gain found for this control interval.

```
float desiredCMD = 100 * RampCMD(2);
```

```
#define DT_CONTROL_MILLIS 10
```

From the open loop experiments we have seen that the encoder resolution is far from perfect at the lower velocities. The ramp function helps identify those issue and will allow us to easily tune the gain to meet stability for the desired control range.

Adjust the proportional gain value until a steady response is achieved for the ramp command similar to the plot below. We will be using this value as a starting point for the PID controller next section.



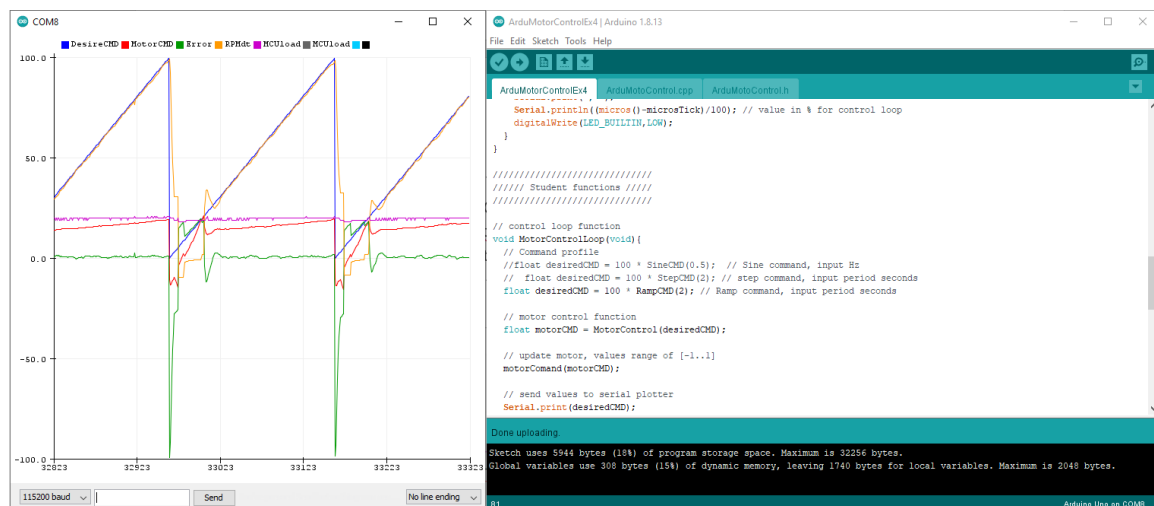
## 4.5 Close loop PID velocity control

Open the **ArduMotorControlEx4.ino** sketch in the Arduino environment. Review the code and find the MotorControl() function. Understand the PID controller implementation and start with a logical value for the integral gain – Remember the concepts described at Part 1 - Simulation. Tune the value until a slight overshoot is achieved for a step response of 100RPM and control interval of 10 milliseconds. Record the response for later analysis in matlab.

**In the final report** calculate the rise time overshoot and settling time of the step response.

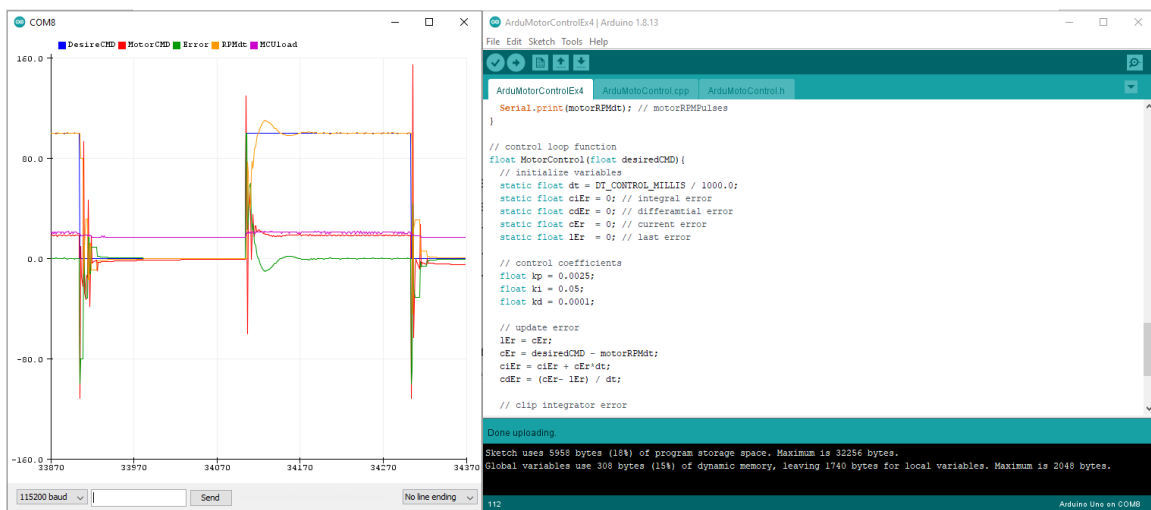


Run the ramp Response with the same values. Record the response and in the final report evaluate the instability point. What happens to the encoder values around this point? Compare this point's motor command to the minimal command needed to start the motor spinning found in the open loop experiment, write your conclusions.



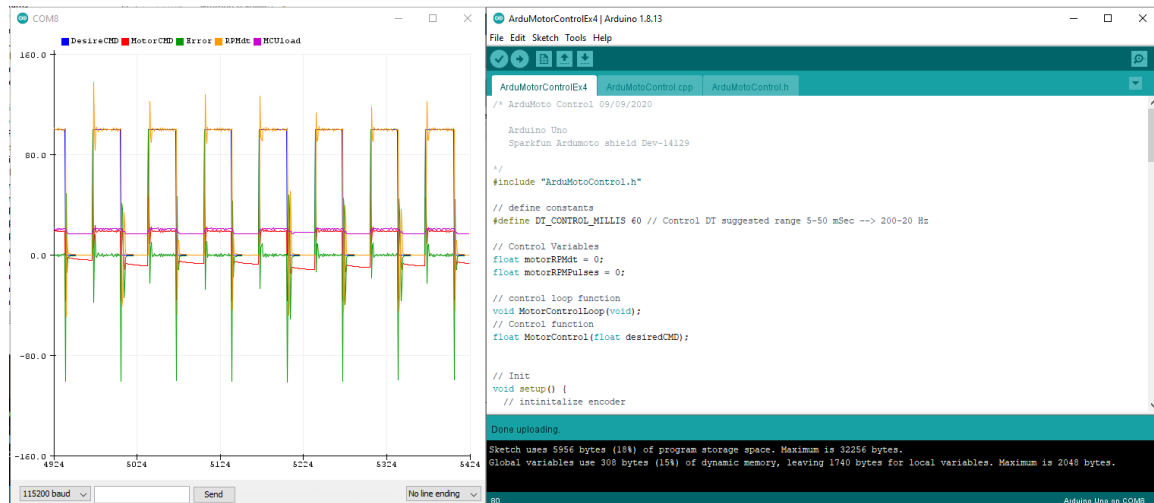
Usually the differential gain isn't required for the velocity control loop of a dc motor, and at the current example it has a very simplified implementation as such adding it will mainly interfere with the motor control implementation. But in order to understand the concept and the simplified implementation we will run a simple example. Start by setting the differential gain to a logical value using the concepts described at Part 1 – Simulation. Increase the value slightly until it's influence will be noticed but without losing stability.

**In the final report** record the results and generate a plot in Matlab marking its influence. Suggest a better implementation of a differential element. Hint – overcoming the error of a numerical differentiation.



## 4.6 Control loop frequency influence on control performance

For this section we will use the PI coefficients found at the previous section with the differential gain set to zero. (PI Controller). Run a series of step responses at 100RPM at various Control intervals `DT_CONTROL_MILLIS`. Start with 10ms and increase the value until you lose stability.



Record the experiment with 3 settings of control interval, 10ms last stable value and a middle value.

**In the final report** generate plots in Matlab marking the rise time and the settling time. Compare the values to the control interval. Write your conclusions regarding suggested control loop frequency.

## 4.7 (Bonus) Comparison between simulation and actual setup

Run the simulation with a PI controller with the gains found for the hardware setup at a 10ms control interval. Generate a Matlab plot comparing the step response of the simulation and the hardware. The simulation is modeled based on the theoretical parameters of the hardware model, suggest a method to improve the simulation model to meet the actual hardware.



## 5. Literature

### 5.1 Micro Controller Unit (MCU)

A microcontroller (sometimes called an MCU or Microcontroller Unit) is a single Integrated Circuit (IC) that is typically used for a specific application and designed to implement certain tasks. Products and devices that must be automatically controlled in certain situations, like appliances, power tools, automobile engine control systems, and computers are great examples, but microcontrollers reach much further than just these applications.

Essentially, a microcontroller gathers input, processes this information, and outputs a certain action based on the information gathered. Microcontrollers usually operate at lower speeds, around the 1MHz to 200 MHz range, and need to be designed to consume less power because they are embedded inside other devices that can have greater power consumptions in other areas.

A microcontroller can be seen as a small computer, and this is because of the essential components inside of it; the Central Processing Unit (CPU), the Random-Access Memory (RAM), the Flash Memory, the Serial Bus Interface, the Input/Output Ports (I/O Ports), and in many cases, the Electrical Erasable Programmable Read-Only Memory (EEPROM).

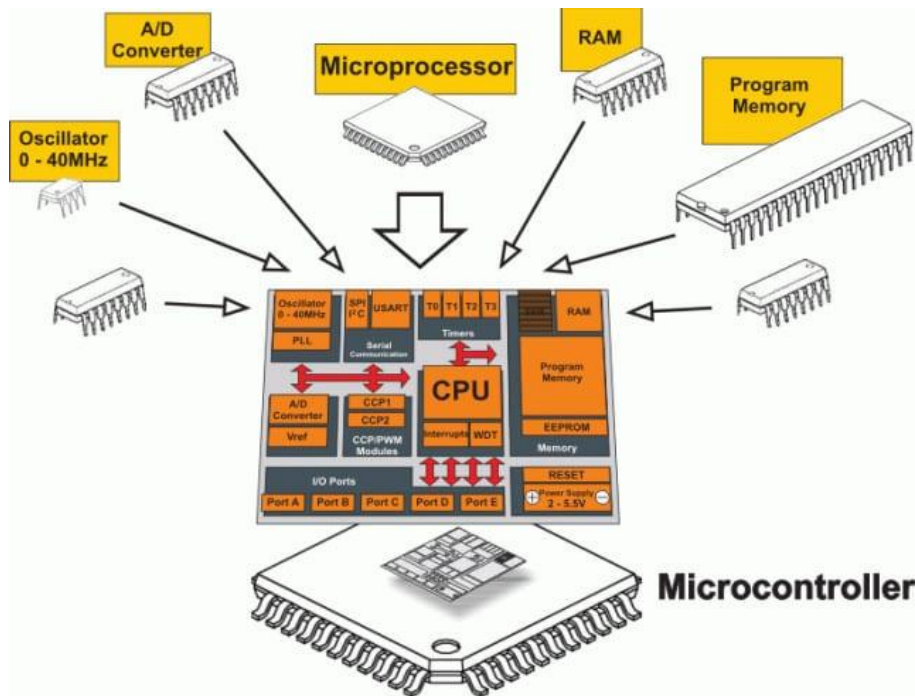
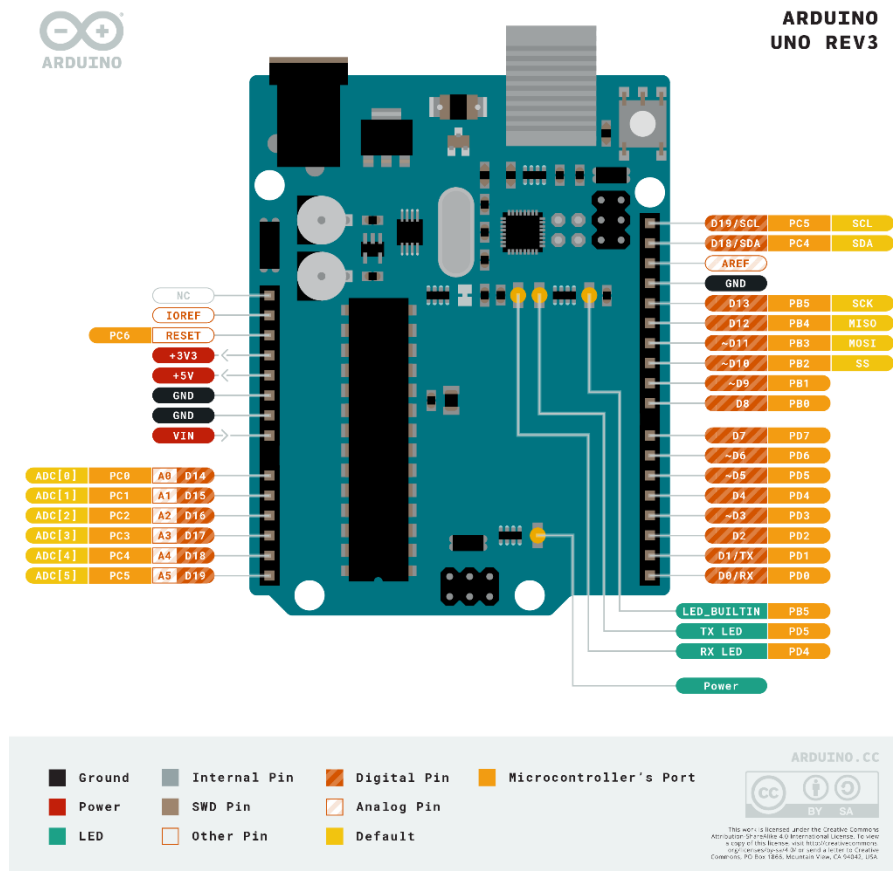


Figure: Parts of a microcontroller. (Source: Max Embedded)

## 5.2 Arduino Development Board and Environment

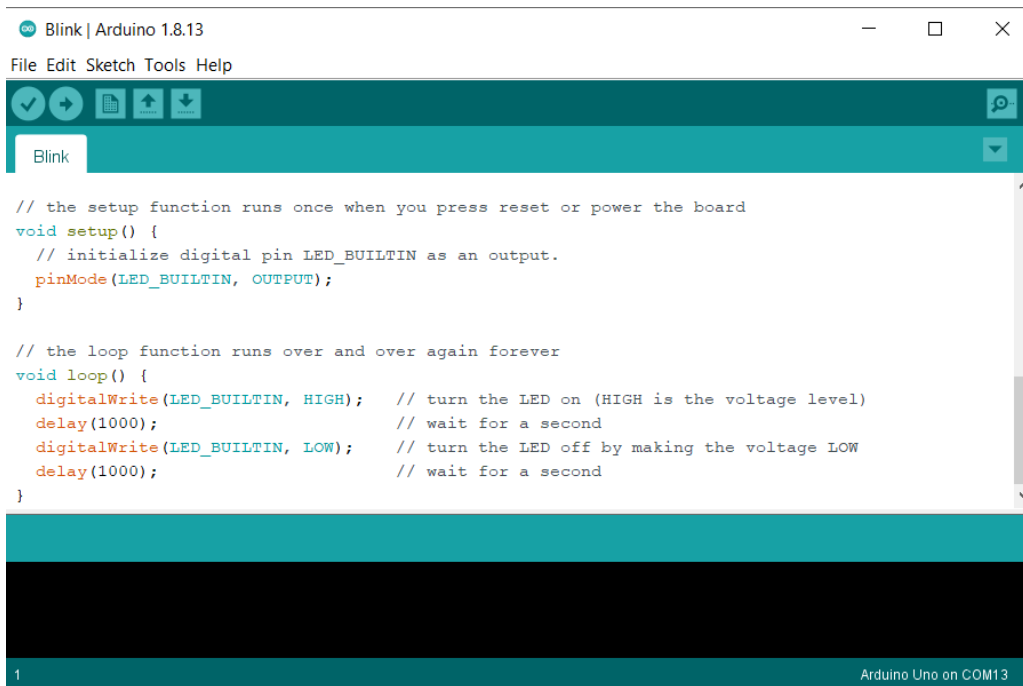
Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards can read inputs, amount of light on a sensor, a press on a button, or the arrival of a Twitter message and turn this into an output, for example, activating a motor, turning on an LED or publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do this use the Arduino programming language and the Arduino Software (IDE).



Arduino Uno is a microcontroller board based on the ATmega328P . It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header and a reset button.

Microcontroller	<a href="#">ATmega328P</a>
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz

The Arduino environment (IDE) has an extensive support for various micro controllers and includes a core library, which simplifies the programming of microcontrollers.

A screenshot of the Arduino IDE interface. The title bar reads "Blink | Arduino 1.8.13". The menu bar includes "File", "Edit", "Sketch", "Tools", and "Help". Below the menu bar is a toolbar with icons for opening files, saving, uploading, and downloading. A tab labeled "Blink" is active. The main text area contains the following C++ code:

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000); // wait for a second
  digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the voltage LOW
  delay(1000); // wait for a second
}
```

At the bottom of the window, a status bar shows "1" on the left and "Arduino Uno on COM13" on the right.

An introduction to the programming environment is available at

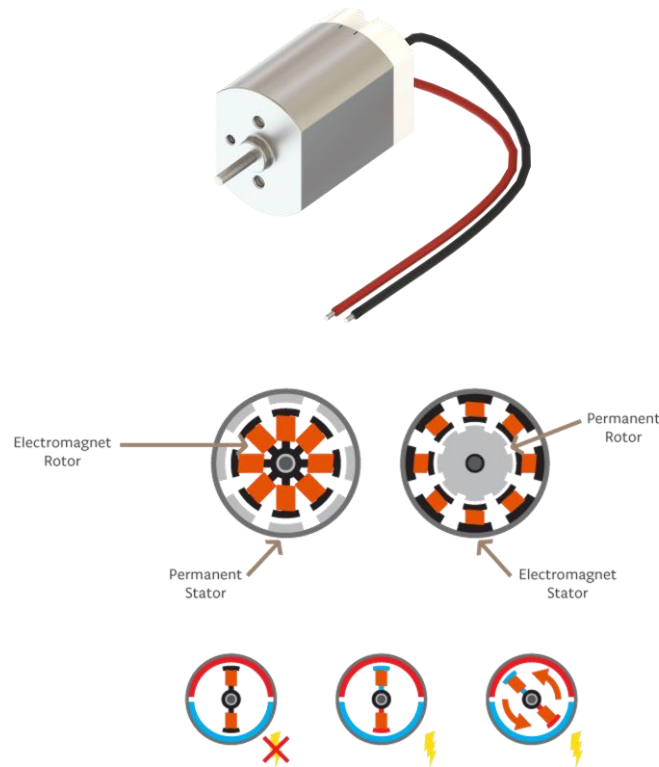
<https://www.arduino.cc/en/Guide/Environment>

The core function and variables descriptions are available at

<https://www.arduino.cc/reference/en/>

### 5.3 DC Motor

A DC (Direct Current) motor is a type of motor that will cause the motor shaft to rotate around its longitudinal axis when applying an electric current between its terminal pins. Thus, the DC motor is a type of actuator that transforms electrical current into rotational motion.



There are two parts inside a motor: the rotor (the shaft is part of this) and the stator. Looking at the cross-section of a motor, you can see that the rotor is the moving part and the stator is the static part. The stator and the rotor use both permanent magnets and electromagnets. Depending on the type of motor, the stator can be a permanent magnet while the rotor is an electromagnet, or vice-versa. Turning on the electromagnet creates attraction and repulsion forces that make the motor spin.

The DC motor spins when we apply DC voltage through its two terminal pins. We can vary the speed of the motor by changing the voltage level. Motors can run in both directions just by reversing the direction of the current.

## 5.4 Motor Driver H-Bridge

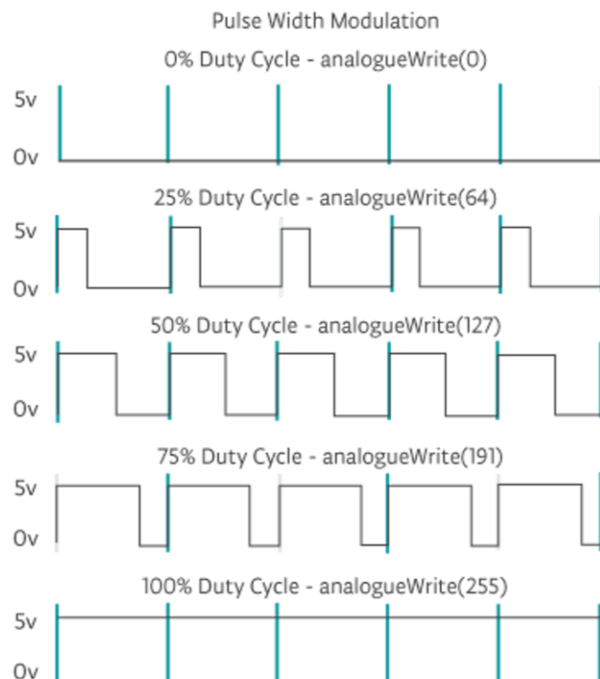
Most electric motors need a higher voltage than can be provided by a microcontroller. To control a DC motor from a microcontroller, you'll need to use a driver. This can be a transistor, a relay or an H-Bridge. The Arduino Motor Carrier uses H-Bridges to drive the DC motors.

The H-Bridge is an electronic circuit that contains four transistors arranged so that the current can be driven to control the direction of the spin and the angular speed. There is a more in-depth explanation of the H-Bridge in section 3.1.4. It is common practice to use PWM (Pulse Width Modulation) signals to control the speed of a motor instead of providing analog voltages.

With the Arduino IDE, if you want to just turn a DC motor on and off, you can use a `digitalWrite` command with `HIGH` for ON and `LOW` for OFF. If you instead want to modify the speed of the motor, you can use the `analogWrite` command and one of the PWM pins.

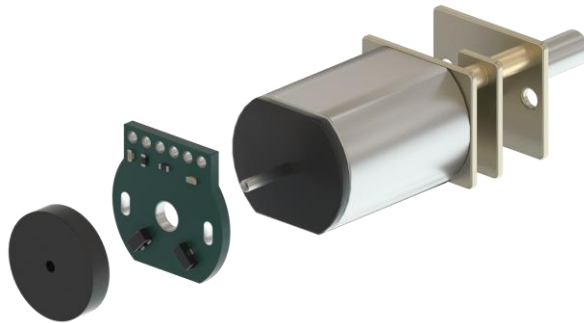
## 5.5 PWM

Pulse Width Modulation, or PWM, is a digital modulation technique commonly used to control the power supplied to electrical devices, like motors. The modulation technique consists of changing the width of a periodic signal's pulse. The width of the pulse is referred to as the duty-cycle and goes from 0% (minimum width) to 100% (maximum width).

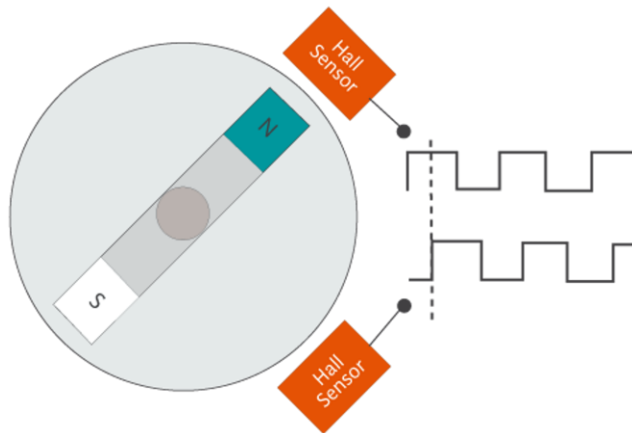


## 5.6 Encoder

Magnetic encoders are sensors that can report information about the rotation speed and the spinning direction of the motor when mounted on a motor. The magnetic encoders are composed of a module with two Hall-effect sensors and magnetic discs. As the motor turns, the disc rotates past the sensors. Each time a magnetic pole passes a sensor, the encoder outputs a digital pulse, also called a “tick”. By counting those ticks, the speed of the motor can be determined.



The encoder has two outputs, one for each Hall effect sensor. The sensors are positioned so that there is a phase of 90 degrees between them. This means that the square wave outputs of the two Hall effect sensors on one encoder are 90 degrees out of phase. This is called a quadrature output.



6. Credits / Resources:

<https://www.arrow.com/en/research-and-events/articles/engineering-basics-what-is-a-microcontroller>

<https://www.arduino.cc/>

<https://www.arduino.cc/reference/en/>

<https://www.arduino.cc/en/Guide/Environment>

<https://aek.arduino.cc/>