POLYTECH
Peter the Great
St. Petersburg Polytechnic
University

Course

« Introduction to Biomedical Engineering»

Dr. Kirill Aristovich

Section 3: Microcontrollers/Arduino
Lecture 3.3: PID controller

Lecture 3: PID controller

Hello, in previous lecture we've used some of the stuff we have learned previously. This lecture is as practical as it can be, and at the same time will utilize ALL of the stuff we have learned before: we are going to use arduino and tinkercad to design a PID controller for a DC motor with rotary encoder.

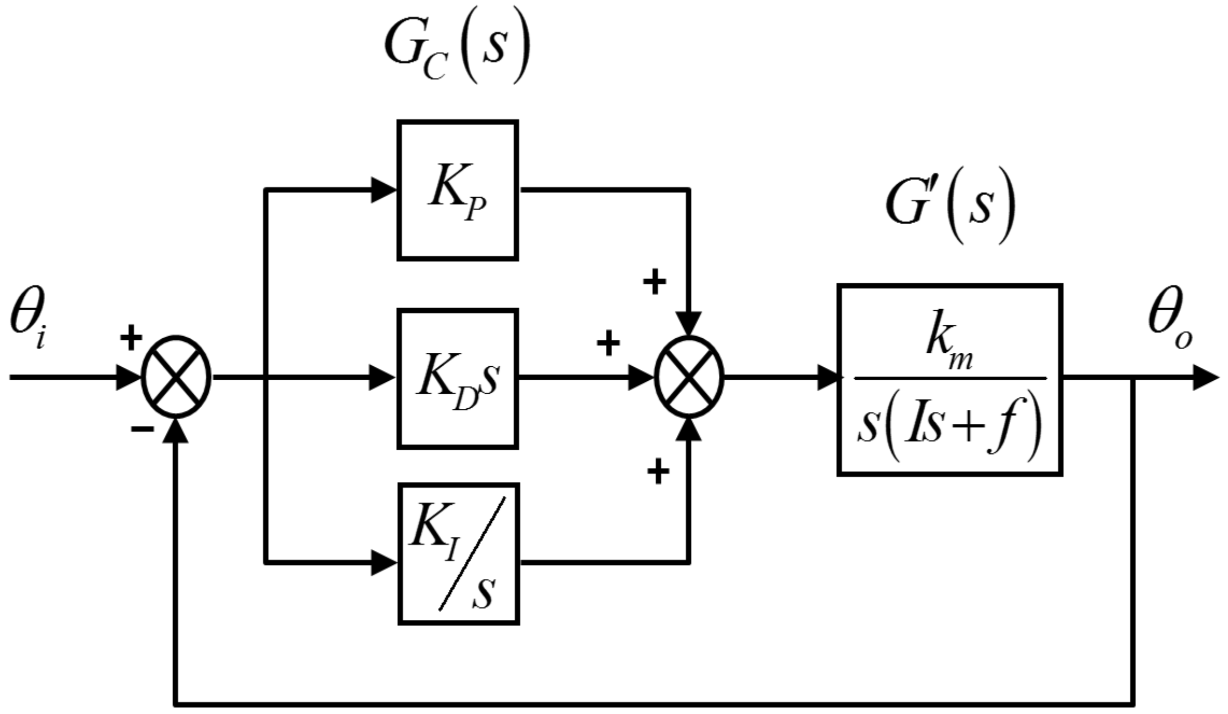
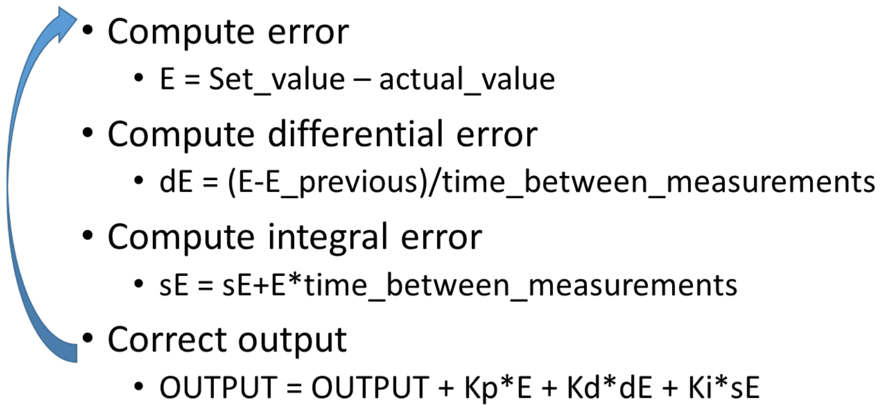


Figure 1 - PID controller

You need to remember the PID controller structure, and basic algorithm for implementing it. We need to set the demand: the desired position or velocity. Then we need to measure the motors actual position/velocity, and compute the error. We also compute the derivative of the error by subtracting previous error from it and dividing by the time between measurements. Remember the definition of the derivative?



- Compute error
    - E = Set_value – actual_value
- Compute differential error
    - dE = (E-E_previous)/time_between_measurements
- Compute integral error
    - sE = sE+E*time_between_measurements
- Correct output
    - OUTPUT = OUTPUT + Kp*E + Kd*dE + Ki*sE

Figure 2 - Basic algorithm

Moving on, we update the integral error, which in the very beginning was set to zero. We constantly keep the total by adding current error multiplied by the time between measurements, again remembering the rectangular rule for numerical integration. Then we compute the sum of all 3 multiplied by their respective gains, and send it to the motor. The rest is happening automatically provided that you have selected correct gains.

Right, let’s see what we are going to need: Arduino Uno, Motor with Rotary encoder, Power supply, TIP120 transistor (for PWM control of motor speed) and resistor.

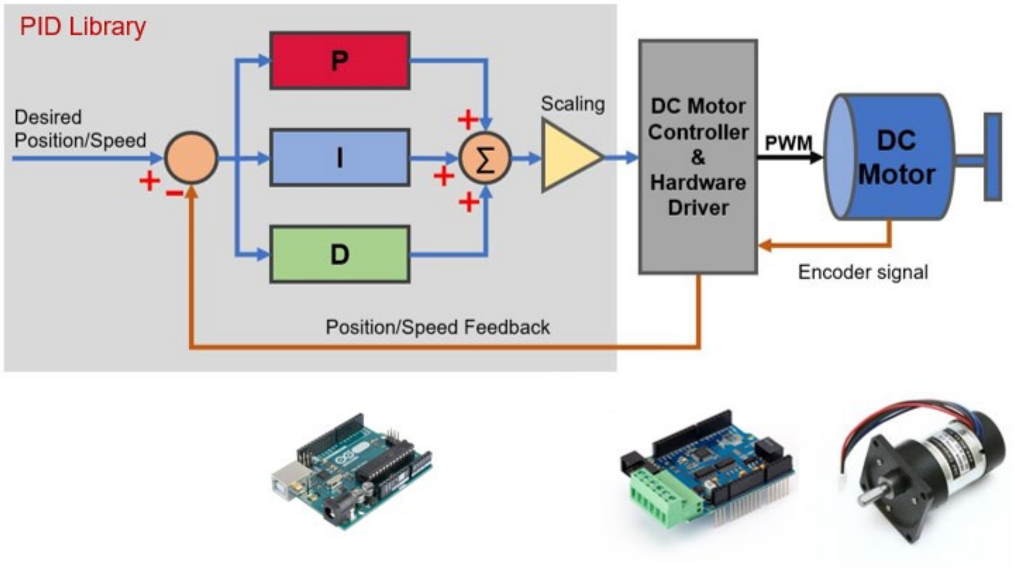


Figure 3 - One way of doing it

The first step we are going to do is to establish the measurement of the motor velocity using the Rotary encoder. The encoder shines the light through a slotted rotated disk, fixed to the shaft, and receives it using the photodiode on the other side of the disk. Thus, the diode produces the pulse every time the slot is in between the LED and the diode. Most encoders would have 2 of the LED-Diode pairs, located in different places to identify the direction of rotation. The time between pulses, and pulse duration is proportional to the angular velocity, or speed of the motor.

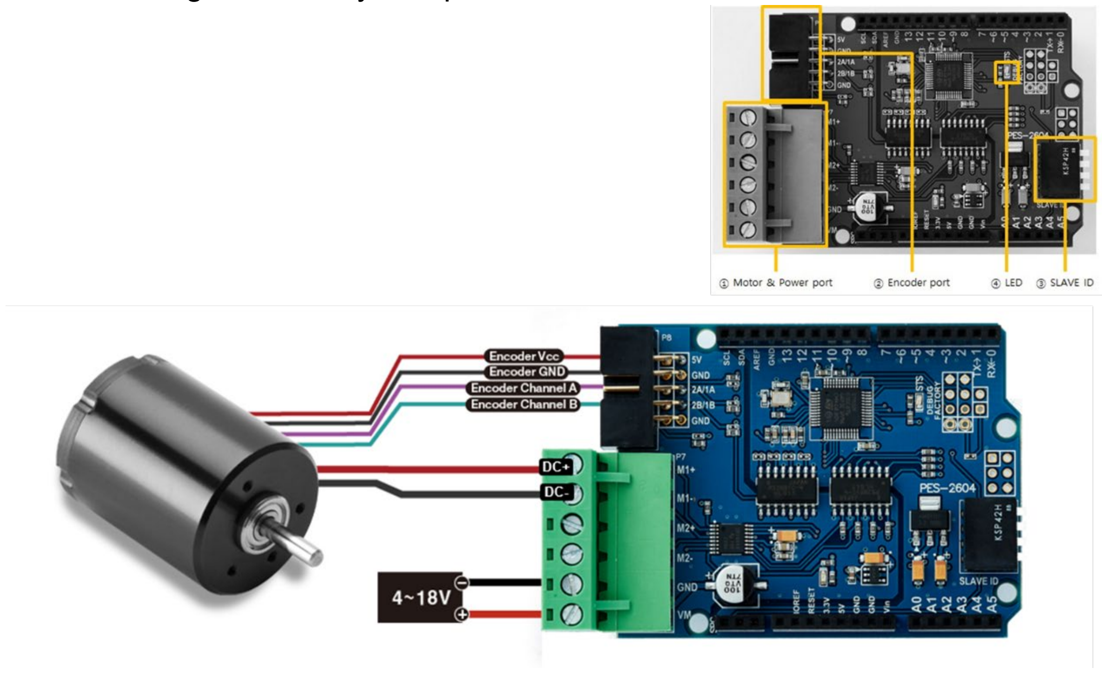


Figure 4 - Possible quick and reliable solution

https://www.phpoc.com/support/manual/pes-2604_user_manual/contents.php?id=layout

The simplest way to read the encoder is to treat it as a switch with the voltage divider, connected to 5V through a resistor, and read the voltage from one channel. This is by far not the best way to do it, if you are looking for better ways, Arduino has multitude of libraries for rotary encoders, but for now it would do for our purposes.

So we would drag the motor to the workspace, noting the pin-out and channels we want to use: encoder ground, power, and channel A.

Then we can make it talk to Arduino by connecting ground and power, 10k resistor between power and channel A, and channel A to the digital pin of Arduino.

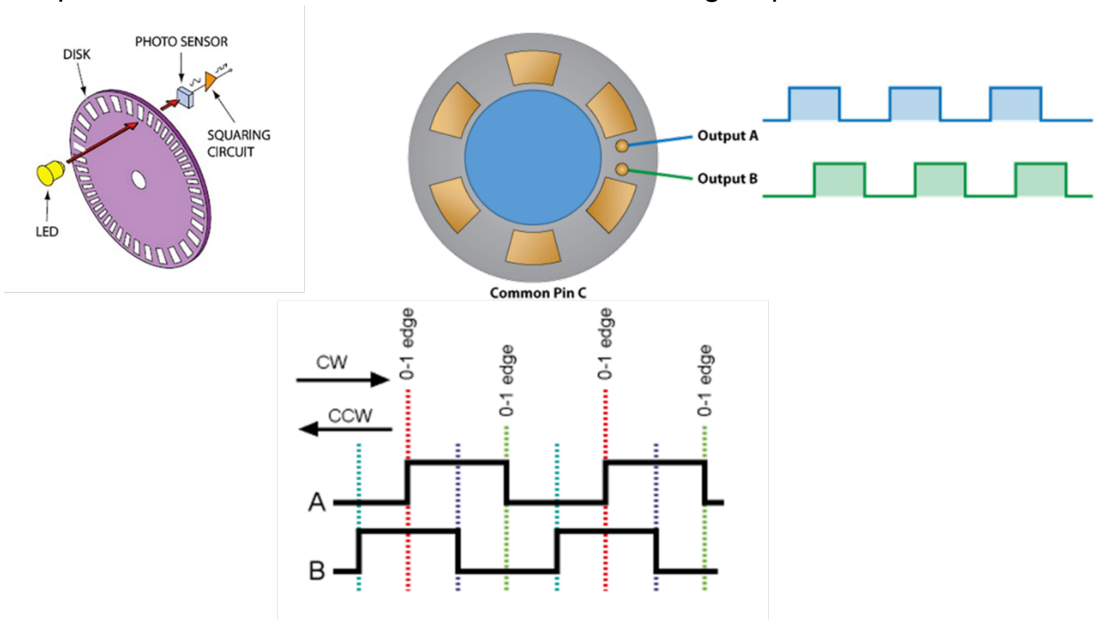


Figure 5 - Rotary encoder

Then we add some high-power to the motor, and share its ground with Arduino. We will not connect the Motor ground pin yet, as we need to understand how to control the speed. Well, remember the transistors lecture? High-power current can be easily controlled by voltage connected to the transistor base. We also obviously remember the strategy of controlling the speed through Pulse Width Modulation and thus connect the base of the transistor through a balance resistor to PWM output of Arduino.

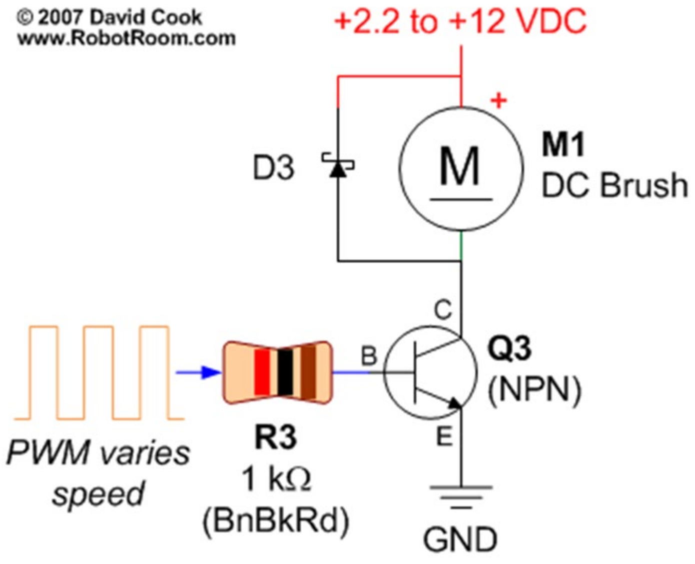


Figure 6 - Output to the motor: PWM

Now the high-power current driving the motor will also be pulse-wise modulated and controlled by Arduino. We will also insert some limiting variable resistor serial to the motor power circuit to adjust the maximal speed of the motor – after all, we do not want it to accidentally fly away, do we?

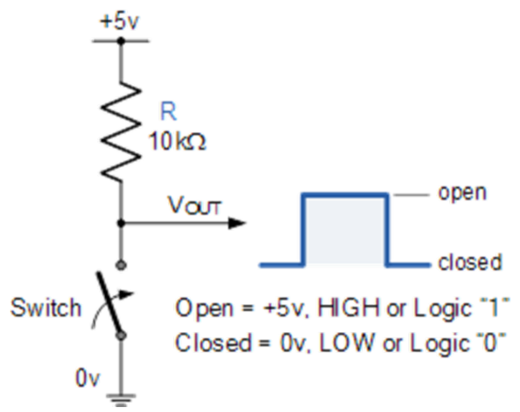


Figure 7 - Simplest way to read the speed (not the best, but simplest)

Now we want to add a potentiometer to set the desired motor speed.

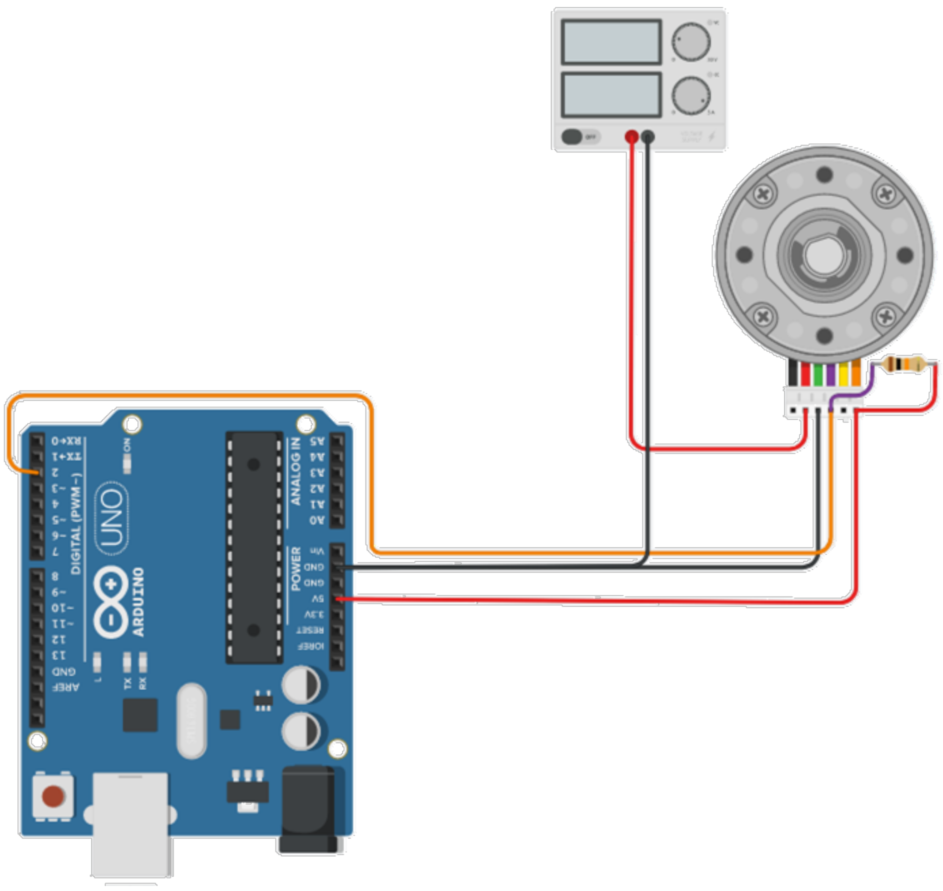


Figure 8 - potentiometer (Add some power)

After that we connect couple of scopes to see what the encoder is showing and what the controller output is, and write the Arduino code according to the algorithm we have explored before.

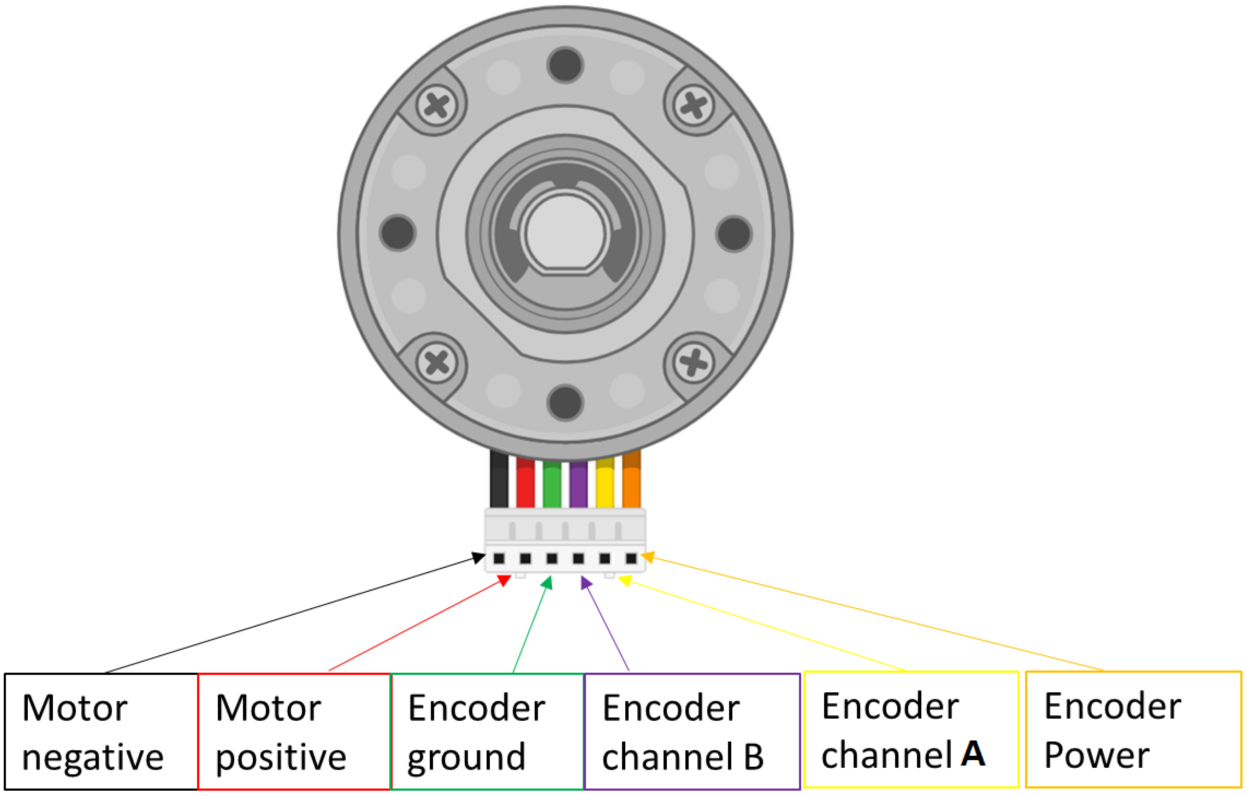
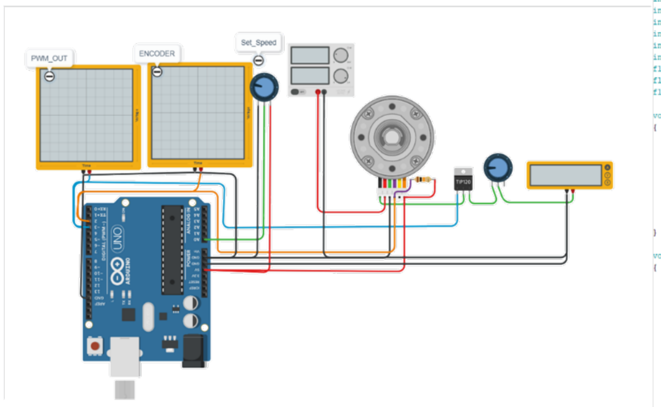
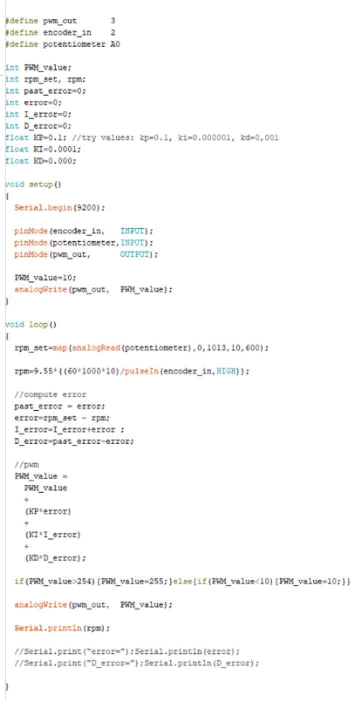


Figure 9 - Motor with encoder

As we have already learned about absolutely every single programming tool that is used in this example, I would advise you to write it yourself. The only additional information you would need to know is that this is a DC motor SPEED control. So think carefully!

If you have done it yourself, congrats! Alternatively you can find the link to tinkercad below. The whole program is not complicated at all, but now when you have implemented the PID controller in practice, you can start appreciating the usefulness of theoretical background and flexibility that it gives you in terms of designing new systems.



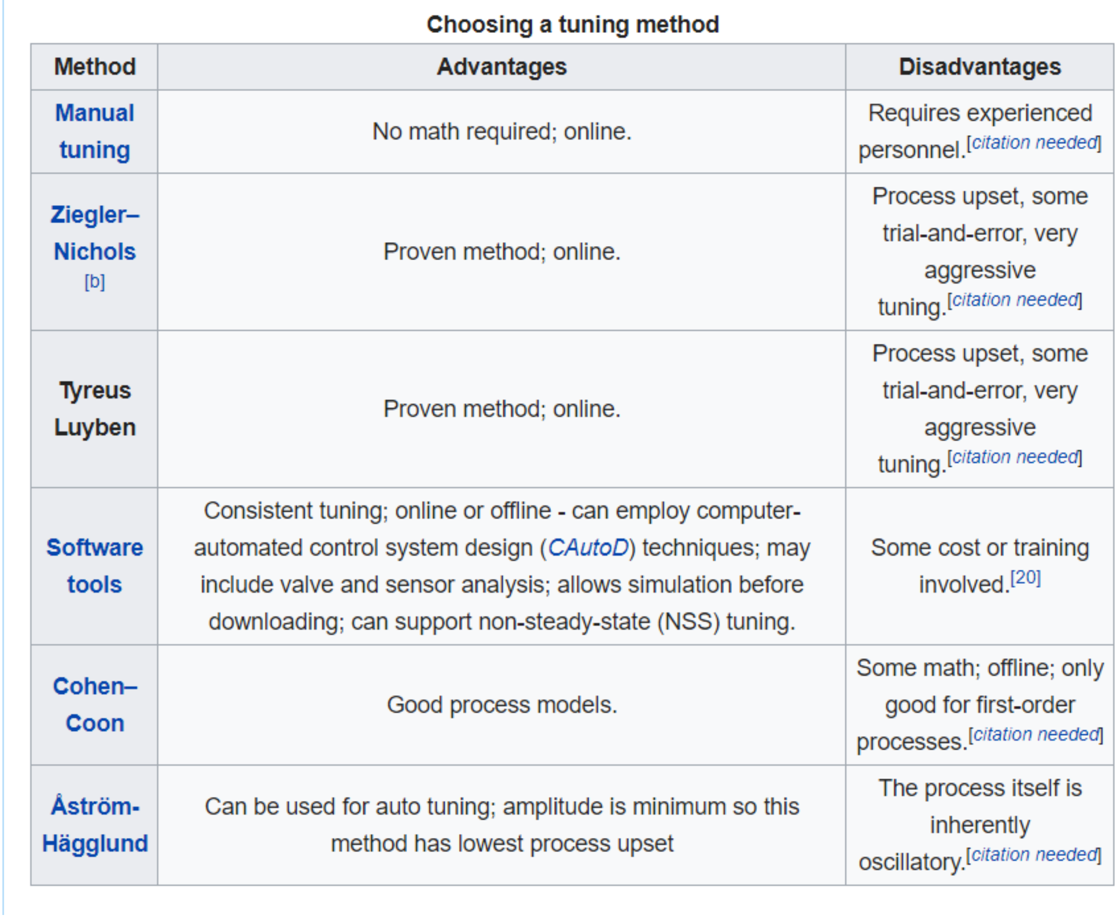


```
#define pwm_out       3
#define encoder_in    2
#define potentiometer A0

int PWM_value;
int rpm_set, rpm;
int past_error=0;
int error=0;
int I_error=0;
int D_error=0;
float KP=0.1; //try values: kp=0.1, ki=0.000001, kd=0,001
float KI=0.0001;
float KD=0.000;

void setup()
{
  Serial.begin(9200);

  pinMode(encoder_in,   INPUT);
  pinMode(potentiometer,INPUT);
  pinMode(pwm_out,      OUTPUT);

  PWM_value=10;
  analogWrite(pwm_out,  PWM_value);
}

void loop()
{
  rpm_set=map(analogRead(potentiometer),0,1013,10,600);

  rpm=9.55*((60*1000*10)/pulseIn(encoder_in,HIGH));

  //compute error
  past_error = error;
  error=rpm_set - rpm;
  I_error=I_error+error ;
  D_error=past_error-error;

  //pwm
  PWM_value =
    PWM_value
    +
    (KP*error)
    +
    (KI*I_error)
    +
    (KD*D_error);

  if(PWM_value>254){PWM_value=255;}else{if(PWM_value<10){PWM_value=10;}}

  analogWrite(pwm_out,  PWM_value);

  Serial.println(rpm);

  //Serial.print("error=");Serial.println(error);
  //Serial.print("D_error=");Serial.println(D_error);

}
```

Figure 10 - motor in Thinkercad

https://www.tinkercad.com/things/9F1gxJoFonI

The only remaining thing to do is to tune the gains. As we know, there are several ways of doing it. If you fancy, you can even use the formal control theory, setting up the parameters you want, and trying to find the transfer function which satisfies those parameters.

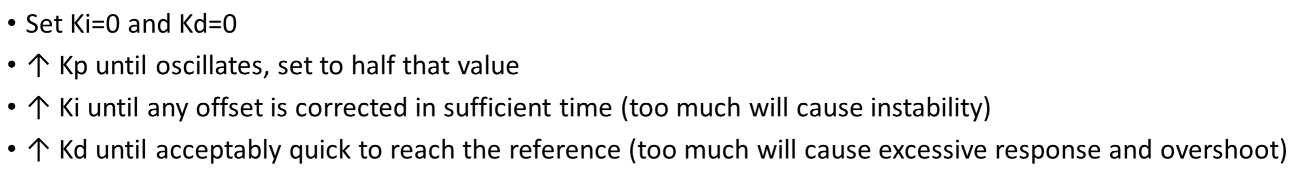


Choosing a tuning method

Method	Advantages	Disadvantages
Manual tuning	No math required; online.	Requires experienced personnel.[citation needed]
Ziegler–Nichols [b]	Proven method; online.	Process upset, some trial-and-error, very aggressive tuning.[citation needed]
Tyreus Luyben	Proven method; online.	Process upset, some trial-and-error, very aggressive tuning.[citation needed]
Software tools	Consistent tuning; online or offline - can employ computer-automated control system design (CAutoD) techniques; may include valve and sensor analysis; allows simulation before downloading; can support non-steady-state (NSS) tuning.	Some cost or training involved.[20]
Cohen–Coon	Good process models.	Some math; offline; only good for first-order processes.[citation needed]
Åström-Hägglund	Can be used for auto tuning; amplitude is minimum so this method has lowest process upset	The process itself is inherently oscillatory.[citation needed]

Figure 11 - Tuning the PID controller

Here, however, try to implement the simplest manual tuning algorithm, which is typically applied for such systems: First, set Ki and Kd to zero. Then increase the Kp until the system starts oscillating. Set Kp to half that critical value. Then increase Ki until any offset is corrected in sufficient time (too much will cause instability). Finally, increase Kd until the system is acceptably quick to reach the reference (too much will cause excessive response and overshoot).



- Set Ki=0 and Kd=0
- ↑ Kp until oscillates, set to half that value
- ↑ Ki until any offset is corrected in sufficient time (too much will cause instability)
- ↑ Kd until acceptably quick to reach the reference (too much will cause excessive response and overshoot)

Figure 12 - Manual tuning

Here is the simple illustration of the process observed on the step response. Now, when your controller is tuned and up-and-running, try to play with the parameters to see what happens if you deviate each gain up or down. Good luck, and see you on the next lecture!

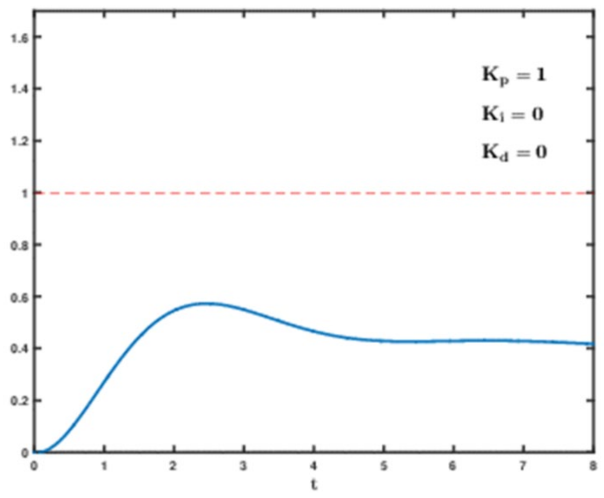


Figure 13 - Tuning the PID controller

Some graphic material used in the course was taken from publicly available online resources that do not contain references to the authors and any restrictions on material reproduction.