PWM
Pulse Width Modulation

PID
Proportional, Integral, Derivative

Motor Control and H-Bridges

Fine and Accurate Varying Power Control with the Arduino
Map () function
map(value, fromLow, fromHigh, toLow, toHigh)
• Maps a value in the range fromLow and maxLow to the range toLow and toHigh. Very useful to process values from analog sensors.

Example:
val = map(analogRead(0),0,1023,100, 200);
// maps the value of
// analog 0 to a value between 100 and 200
• **Functions of a Programmable Controller – microprocessor architecture review**

• **Control relay functions:** input/output signals

• **Timing functions:** generation of output signals for a specific amount of time

• **Counting Functions:** an internal counter in the PC is used to sum the number of contact closures and generate an output signal when the sum reaches a certain level.

• **Arithmetic Functions:** basic arithmetic functions such as multiply, add and subtract

• **Analog control functions:** such as having a potentiometer as a sensor

• Some of the **features** that some of the most modern controller include maintenance and diagnostic functions on an LCD

• **Operator interface** – can provide performance operations about production rates, tool usage, equipment breakdown and other data – a printer can also be used for this

• **Safety monitoring** – using interlocks (sensors to monitor problems and stop them)
Two Methods of Controlling Motors and Electronics

- **Electromechanial (using relays and manual switches) 1960’s**
  - What are the *disadvantages* of using relays in control systems? Noisy, big and require higher voltages and amps to run. They can fail because they have so many mechanical parts. But relays.

  - **Advantages:** Very easy to interface. Can handle a ton of amps! Older electric wheelchairs used relays in their design to turn motors on-off. Can be used in other areas such as safety to remove power from all robots when a limit switch gets triggered for example. Simpler overall system design.

- **Programmable controllers – microcontrollers like the Arduino (Atmel)**
  - Introduced in the late 60’s as brains for automation and robots.
  - **Advantages:** smaller, flexible because they can be reprogrammed to perform different tasks. This is basically a microprocessor based controller or a PC that can be defined as a digitally operating device with programmable memory that is capable of generating output signals according to logic operations and other functions performed on input signal. Can be programmed using methods such as ladder diagrams, symbolic notation and higher level languages such as VB or C++.
PWM Pulse Width Modulation

• Controlling electrical power through a load by means of quickly switching it on and off, and varying the "on" time, is known as pulse-width modulation, or PWM. It is a very efficient means of controlling electrical power because the controlling element (the power transistor) dissipates comparatively little power in switching on and or, especially if compared to the wasted power dissipated of a rheostat in a similar situation. When the transistor is in cutoff, its power dissipation is zero because there is no current through it. When the transistor is saturated, its dissipation is very low because there is little voltage dropped between collector and emitter while it is conducting current.

• Can also be used to control brightness of DC light bulbs, temperature of heaters and heating elements, etc.
PWM, is a technique for getting analog results with digital means.

Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off.

The duration of "on time" is called the **pulse width** or **duty cycle**. To get varying analog values, you change, or modulate, that pulse width.

If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.
Press button manually really fast to get **10% output from the 9 volts.** 10% means is outputting .9 volts if it is coming out of a 9 volt battery.
Anatomy of a Pulse

+A-0 is the voltage and **Pulse Width is the Duty Cycle** or amount that is ON
Pulse Width: how much voltage (*Amplitude*) is putting out
Period: total duration of the pulse total amount of voltage allowed to put out
Duty Cycle: total percentage of the voltage you are putting out
The Arduino (Atmel) microprocessor uses built-in timers to control the timing of these pulses. Remember the 555 timer?
Arduino has 8 bit timers (0-255)

- Arduino produces its PWM at about 500 Hz or one cycle every 200 milliseconds

- `analogWrite(ledPin, fadeValue or duty cycle);`
- `analogWrite(11, 127);`
- This line produces a 50% duty cycle
Pulse Period (T)

Pulse Width (t)

Duty Cycle = Pulse Width / Pulse Period

Duty Cycle: 25%

Duty Cycle: 50%

Duty Cycle: 75%

PWM Signal Input

Average Voltage on the Motor

5 Volt

1.25 Volt

2.5 Volt

3.75 Volt

PWM Timing Diagram
analogWrite(). This function invokes the Pulse Width Modulation capabilities of our Arduino board. Pulse Width Modulation basically adjusts the power output at the pin. So you can have a lot of power or a little power applied at the pin, its your call, you just tell the analogWrite() function which pin to modulate and how much power you want applied. The scale is from 0 to 255 with zero being the lowest power setting and 255 being the highest (5 volts).

analogWrite(pin, value)

You can utilize pins 3, 5, 6, 10 and 11 with analogWrite() on most Arduino boards (there is a “PWM” or “~” next to the pin number on the board). The value range is from 0 to 255 (8bit).
Using PWM to fade an LED – sending varying power over time

With PWM

Without PWM on/off

1-LED on
2-LED Off (520us)
3-LED On (2.04mS) full cycle

50% Duty Cycle
10% Duty Cycle
90% Duty Cycle

5 V
GND = 0 V

With PWM

Medium
Dim
Bright

time
Varying voltage with PWM from 0-5 volts

Conversion of digital values into an analog voltage value
Arduino UNO – PWM pins 3, 5, 6, 9, 10, 11
// fades an LED from min to max then max to min: PWM example
int ledPin = 5;    // LED connected to digital pin 5

void setup() {
    // nothing happens in setup
}

void loop() {
    // fade in from min to max in increments of 5 points:
    for(int fadeValue = 0 ; fadeValue <= 255; fadeValue +=5) {
        // sets the value (range from 0 to 255):
        analogWrite(ledPin, fadeValue);
        // wait for 30 milliseconds to see the dimming effect
        delay(30);
    }

    // fade out from max to min in increments of 5 points:
    for(int fadeValue = 255 ; fadeValue >= 0; fadeValue -=5) {
        // sets the value (range from 0 to 255):
        analogWrite(ledPin, fadeValue);
        // wait for 30 milliseconds to see the dimming effect
        delay(30);
    }
}
Another Fade LED Example
/* Fade
This example shows how to fade an LED on pin 9 using the analogWrite() function.
This example code is in the public domain. */

int led = 10;  // the pin that the LED is attached to
int brightness = 0;  // how bright the LED is
int fadeAmount = 5;  // how many points to fade the LED by

// the setup routine runs once when you press reset:
void setup() {  
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {  
  // set the brightness of pin 9 – basically puts 0 volts on pin 9
  analogWrite(led, brightness);
  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;
  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
Code Explanation

• || means OR
• == is a comparison operator
• Example:
  • brightness = 0  //this statement assigns a value to the brightness variable
  • brightness == 0  //this statement compares the brightness variable with 0

How does the fading work?

```javascript
brightness = brightness + fadeAmount;
if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
}
```

Once the brightness variable gets as low as zero, it will switch the fadeAmount to positive and start the whole process over again! A fading LED
Introducing the Continuous Rotation Servos and Regular Servos

Servo Connector:
- Black – Vss
- Red – Vdd
- or Vin
- White – Signal

- Control horn
- Phillips screw
- Label should read “Continuous Rotation”
- Case contains motor, circuits, and gears
- Cable for power and control signal
- Mounting Flange
- Access hole for center adjusting feedback potentiometer
- Plug for RC servo connection ports on Board of Education
Expanded View of a Hobby class Servo

- Potentiometer helps to tell the position of the shaft
- Standard servos can only rotate up to 180 degrees
- You can modify them to make them continuous rotating motors
Regular servo motors have feedback control. They have a sensor to tell position and direction that the shaft is turning. Modified servo motors used in hobby robotics do not have. You can add an external sensor to provide speed and direction feedback.

**Servo Control**

- Most hobby servo motors use a standard three wire interface:
  - Power
  - Ground
  - Control Line

- Power supply is typically 5 to 6 V

The input to the servo motor is desired position of the output shaft. This signal is compared with a feedback signal indicating the actual position of the shaft (as measured by position sensor). An “error signal” is generated that directs the motor drive circuit to power the motor. The servo’s gear reduction drives the final output.
Many Servo Motor controllers will accept 1-2 ms pulses every 20-25 milliseconds.

Figure 4-15
Timing Diagram for 2.0 ms Pulses Every 20 ms
Servo horn is in 10 o’clock position.

Figure 4-16
Timing Diagram for 1.0 ms Pulses Every 20 ms
Servo horn is in 2 o’clock position.

Figure 4-17
Timing Diagram for 1.5 ms Pulses Every 20 ms
Servo horn is in 12 o’clock position.

(Adapted from http://team358.org/files/electrical/PWMGenerator.pdf. Not to scale.)
Arduino PWM

Problem – they do not match

What a servo wants

analogWrite() function produces a 490 Hz PWM signal, or a 2 ms period.

Full Forward: 2 ms pulse / 20 ms period = 10% duty cycle.
Neutral: 1.5 ms pulse / 20 ms period = 7.5% duty cycle.
Full Reverse: 1 ms pulse width / 20 ms period = 5% duty cycle.
Servo wants 50hz or 20ms periods
Centering the Servos before running code

Insert tip of Phillips screwdriver into potentiometer access hole.

Gently turn screwdriver to adjust potentiometer
Sample Servo Sweep and Knob  
Connect +, - and Signal Cable

```cpp
#include <Servo.h>  // using the servo library
Servo myservo;  // creates servo object
void setup()
{
    myservo.attach(9);
    myservo.write(90);  // set servo to mid-point
}
void loop()
{
}
```
Code Explanation:

```c
#include <Servo.h> // is the software library that takes care of converting the period frequency from 490hz to 50 hz.

Example
#define MIN_PULSE_WIDTH   544    // the shortest pulse sent to a servo
#define MAX_PULSE_WIDTH   2400   // the longest pulse sent to a servo
#define DEFAULT_PULSE_WIDTH 1500 // default pulse width when servo is attached
#define REFRESH_INTERVAL  20000  // minimum time to refresh servos in microseconds (20ms)

Servo myservo; // create servo object to control a servo
```

Where is the servo connected?
servo.attach(pin)

Write a value to move the servo to a particular position
```c
myservo.write(pos);
```
# Sample Servo Sweep from Arduino Samples

```c
#include <Servo.h> // use the Servo.h library that takes care of our 20ms frequency difference

Servo myservo; // create servo object to control a servo
    // a maximum of eight servo objects can be created
int pos = 0; // variable to store the servo position

void setup()
{
    myservo.attach(9); // attaches the servo on pin 9 to the servo object
}

void loop()
{
    for(pos = 0; pos < 180; pos += 1) // goes from 0 degrees to 180 degrees
    {
        // in steps of 1 degree
        myservo.write(pos); // tell servo to go to position in variable 'pos'
        delay(15); // waits 15ms for the servo to reach the position
    }

    for(pos = 180; pos>=1; pos-=1) // goes from 180 degrees to 0 degrees
    {
        myservo.write(pos); // tell servo to go to position in variable 'pos'
        delay(15); // waits 15ms for the servo to reach the position
    }
}
```
Sample Servo Sweep with Potentiometer
#include <Servo.h>

Servo myservo; // create servo object to control a servo

int potpin = 0; // analog pin used to connect the potentiometer
int val; // variable to read the value from the analog pin

void setup()
{
  myservo.attach(9); // attaches the servo on pin 9 to the servo object
}

void loop()
{
  val = analogRead(potpin); // reads the value of the potentiometer (value between 0 and 1023)
  val = map(val, 0, 1023, 0, 179); // scale it to use it with the servo (value between 0 and 180)
  myservo.write(val); // sets the servo position according to the scaled value
  delay(15); // waits for the servo to get there
}
Code Explanation:

Map usage:  \texttt{map(value, fromLow, fromHigh, toLow, toHigh)}
Ex. \texttt{val = map(val, 0, 1023, 0, 179)};
// scale it to use it with the servo (value between 0 and 180)

// Sample on running servo without using the servo library

\begin{verbatim}
void setup() // Built in initialization block
{ pinMode(13, OUTPUT); // Set digital pin 13 -> output }

void loop() // Main loop auto-repeats {
  digitalWrite(13, HIGH); // Pin 13 = 5 V
  delay(17); // ..for 0.17 seconds
  digitalWrite(13, LOW); // Pin 13 = 0 V
  delay(183); // ..for 1.83 seconds
}
\end{verbatim}
Servo Motors

- The advantages by this integrated servo motor solution are:
- De-central intelligence.
- Simple installation. No cables between servo motor and driver.
- Switching noise remains within servo motor.
- Single or dual supply.
- Compact. Does not take space in cabinet. Typically a 3/5 core cable is used from PLC or similar to MAC motor.

- 24 or 48VDC power source
  - From PC/PLC with drive commands via RS232/RS485/RS422
  - Pulse/direction or quadrature inputs.
  - 10 bit (MAC50-141), 12bit (MAC800) ±10V input for speed or torque control. A+B encoder output.
  - Register mode via 4 inputs or serial commands
  - Option for nano PLC built-in with graphical programming tools.
  - Multiaxis operation with up to 255 units on the same 460Kbit RS485 bus.
Robot Arms PMDC Servos
Microcontroller Motor Control

- Computer -> Microcontroller -> Speed Controller -> Motor
Controlling larger load with a transistor

```cpp
int fadePin = 9;
void setup(){
    pinMode(fadePin, OUTPUT);
}
void loop(){

    for(int i = 0; i<360; i++){
        //convert 0-360 angle to radian (needed for sine function)
        float rad = DEG_TO_RAD * i;

        //calculate sin of angle as number between 0 and 255
        int sinOut = constrain((sin(rad) * 128) + 128, 0, 255);

        analogWrite(fadePin, sinOut);

        delay(15);
    }
}
```
Electronic Control

H-Bridge Motor Driver Circuit

- Four transistors form the vertical legs of the H, while the motor forms the crossbar
- In order to operate the motor, a diagonally opposite pair of transistors must be enabled
- Transistors Q1 and Q4 enabled
  - Starting with the positive power terminal, current flows down through Q1, through the motor from left to right, down Q4, and to the negative power terminal
  - Results in motor rotating in a clockwise direction
- Transistors Q2 and Q3 enabled
  - Results in current flowing through the motor from right to left
Direction Control by H-Bridge

- Direction of current can be changed using microprocessor output ports and via software commands.
- Care must be taken to never have same side transistors on together.

Rotor coasts freely
Forward direction
Reverse direction
Braking

Motor Driver Chip L293D
Simple motor controller using an H-Bridge. Second circuit shows extra parts used to detect back-EMF and avoid damaging the MOSFETS and other parts. You can see how motor controllers can quickly become very complex. Other items that a controller should detect are:

- Detect for wrong polarity
- Wrong input voltage
- Wrong motor capacity (current)
• There are half bridges too:

• Where can we use this?
Speed Controllers or H Bridges

- **Dual Channel H-Bridge, 2 Amps per channel**
  Control the speed and direction of 2 DC motors from 5 to 28 volts
- **High Efficiency Mosfet design - No Heatsinks Needed.**
- **Short Circuit Protection, Current Limiting, Over Temperature Shutdown, Reverse Polarity Protected**
Sample: specs for a motor speed controller that accepts PWM input:

The Victor 884 is a speed controller specifically engineered for robotic applications. The 40A capacity, low voltage drop, and peak surge capacity make the Victor 884 ideal for drive systems while its braking options and precise control meet the demanding needs of arms and lift systems.

The Victor 884 is an improved version of our Victor 883 speed controller. The 884 has greater control of arms, mechanisms, and feed-back loops. The Victor 884 provides output control from 3% to 100%. The Victor 883 has output control from 10% to 100%.

The Victor 884 has a deadband with respect to the PWM signal, which is approximately 117 to 137 (127 center). Any PWM signal within the deadband results in no output (neutral). The PWM deadband accommodates joysticks that do not return to the exact same center.
Variable Speed Controller

**Specs:**
- **Supply Voltage:** 12 to 36 Volts DC
- **Load Current:** 80 AMPS Continuously, with Resistance and Inductive loads
  120 AMPS for short periods of acceleration (up to 10 sec.)
- **Range of regulation:** PWM 0 to 100% of power
- **Regulating frequency:** 1.6 kHz
- **Overheating protection:** When the module is overloaded, power is automatically decreased, in order to protect mosfets from destruction

In addition to the potentiometer, this controller accepts the following Inputs:

- Arduino PWM
• **Types of steering mechanisms:**

• **differential steering and car like.**

• Most robots use differential or tank like steering because you can turn on a dime while car like steering requires a bigger turn radius.

• **Dead Reckoning** in robotics means that when you tell the robot go from point A to point B by knowing its starting point then tried to go into its end point without checking other variables that might come up during the travel.
Differential or Tank Steering Systems
Differential Steering Maneuvers

Clockwise

Forward

Right Side

Backward

Left Side

Counterclockwise

Forward

Left Turn

Forward

Right Turn
Navigating a course using Dead Reckoning

There are several ways to navigate a course using a robot. A variety of sensors are available to help the robot accomplish this task. Methods for navigating the a course includes dead reckoning. Dead reckoning involves programming a robot with step by step commands to go in a particular path to a final destination. Ex, Go 3 feet forward, make left turn, go 2 feet left….

In dead reckoning

- Exact path must be known ahead of time. Start from a know position.
- Program tells the robot how far to travel before each turn
- Errors in distances and turning angles accumulate, so this method is best for simple, short courses.
- Distances can be calculated using servos or stepper motors or infrared sensors can be used to count wheel revolutions (using a wheel encoder).
Differential drive dead reckoning

Here are the dead reckoning equations for the coordinates \((x\) and \(y\)), and heading \((\theta)\) for a differential drive robot with encoders on both drives (motors):

\[
\Delta \theta = 2\pi \frac{R_W}{D} \frac{T_1 - T_2}{T_R}
\]

\[
\Delta x = R_W \cos(\theta) (T_1 + T_2) \frac{\pi}{T_R}
\]

\[
\Delta y = R_W \sin(\theta) (T_1 + T_2) \frac{\pi}{T_R}
\]

Where \(T_1\) are the encoder ticks recorded on drive one, \(T_2\) are the encoder ticks recorded on drive two, \(R_W\) is the radius of each drive wheel, \(D\) is the separation between the wheels, and \(T_R\) is the number of encoder ticks recorded in a full, in-place rotation. Heading:

The compass direction toward which a traveler or vehicle is (or should be) moving
Two Basic Types of Control Systems:

• **Open Loop**: controller tells your system to do something, but doesn’t use the result of that action to verify the results or modify the commands to see that the job was done properly.

• Example: tell the robot to go forward 6 inches but not measuring the speed or anything else so this open loop – nothing is being checked. So if a sudden obstacle gets in the way, you won’t know if the robot went 6 inches or not. Open loop works for short range projects but not for long range travels.

• **Closed Loop**: sensors and the microcontroller continuously monitors the performance of your system and changes the commands as necessary to stay on track. Ex. Cruise control in a car. What is being monitored here is the speed of the car to keep it constant. Another example would be to have a robot follow a wall by having a sensor measure the distance between the robot and the wall continuously. If there is an error, the controller would adjust the power to the motors.
Open-loop vs Closed-loop
• 4 Basic Control Actions (as it applies to motion) that are used singly or in connection with each other (there are entire classes dedicated to this and entire degree options also):

  ON-OFF, PROPORTIONAL, DERIVATIVE, INTEGRAL

• ON-OFF: is basically a switch or relay as we discussed earlier. There are only two states on or off. In robots ON means full power and OFF means no power so there is nothing in between. These does not allow us to have full manageable control so we need to add proportional control. Ex. Switch to turn a lamp on and off
PID and Controllers

PID stands for **Proportional, Integral, Derivative**. Controllers are designed to eliminate the need for continuous operator attention. Cruise control in a car and a house thermostat are common examples of how controllers are used to automatically adjust some variable to hold the measurement (or process variable) at the set-point. The **set-point** is where you would like the measurement to be. **Error** is defined as the difference between set-point and measurement.

The variable being adjusted is called the manipulated variable which usually is equal to the output of the controller. PID Controllers work in a **closed loop system**.
Proportional-Integral-Derivative Position Control

- PID control techniques produce a highly responsive, accurate, and stable system

The idea is to match PV to SP
PID Controller

Calculate control actions and multiply each by Error

(add up all 3)

Error

SP

PV

Controller Output
• **Proportional Mode**: primary and simplest control mode. The proportional amplifier generates a velocity command proportional to position error. Such large correction can cause overshoot.

• **Integral Mode**: provides a velocity command signal that reduces static error to zero and to provide stiffness for the system. (*sum of all the instantaneous values that the signal has been, from whenever you started counting until you stop counting*)

• **Derivative Mode**: (rate of change) quickly boosts a motor speed (for example) at the beginning of travel and provide a braking action at the end of the travel.
• Analysis of Control Systems – Terms used:
  • **Transcient Response of a System**: is the behavior of the system during the transition from some initial state to the final state.
  
  • **Steady-State Response**: is the behavior of the system as time approaches infinity. Is concerned with determining the response of the system after the transient response has disappeared.
Proportional

Proportional (reacts to the same distance of the error):
- You can have a mechanical or electronic speed control that can give you proportional control such as a dimmer switch. (think servo know example).
- Provides a control signal that is proportional to the error. The problem is a proportional controller tends to settle on the wrong corrective effort. As a result, it will generally leave a steady state error (offset) between the set-point and the process variable after it has finished responding to a set-point change or a load.

Also known as Gain
Integral Control (reacts to biases): In here the control signal is changed at a rate proportional to the error signal. For example: *if the error signal is large, then the control signal increases rapidly.* If the error signal is small, the control signal increases slowly. If there is no error then the output of the controller remains constant. Integral control is basically accumulated error. Integral is good to use when there are biases in the system. With integral action, the controller output is proportional to the amount of time the error is present (keeps adjusting as long as there is an error). Integral action eliminates offset.

- Also known as Automatic Reset or simply Reset
Derivative

Derivative Control (reacts at the speed of error):

It provides a control signal proportional to the rate of change of the error signal. Generates one large corrective effort immediately after a load change in order to begin eliminating the error as quickly as possible. Is rarely used alone because it needs to know the rate of change of the error signal. The derivative action doesn't produce a particularly precise corrective effort, but it generally gets the process moving in the right direction much faster than a PI controller would.

• Also known as Rate
P - I - D

Load Step Time Response

IAE = 192.32
SSE = 790.695

- P only - notice the offset
- P and I - offset gone
- P, I and D - best

Process Variable

Controller Output

Time (sec)
Combinations

• **Proportional+Integral**: sometimes it is necessary to combine control actions. A proportional control is incapable of counteracting a load on the system without an error. *Integral control can provide zero error but usually provides slow response.*

• **Proportional+Derivative**: Derivative control action provides a control signal proportional to the rate of change of the error signal. Since this would generate no output unless the error is changing, it is rarely used alone. The effect of *derivate control action is to anticipate changes in the error and provide a faster response to changes*. Example - Helps center a car by adjusting the rate little by little.
• Proportional+Integral+Derivative:
• PID is used to automatically adjust some variable to hold the measurement (or process variable) at a set-point.
• PID is used in cruise controls, thermostats.
• It provides quick response. PID is basically an error correcting system. PID can be implemented using hardware or software. Some motor controllers have those functions already built-in or you can create your own in software by inputting all the equations and letting the microcontroller handle the results.
• If you are using only one but not the combination of PID, the controller may over-correct for an error and create a new one of even greater magnitude in the opposite direction. When that happens, the controller will eventually start driving its output back and forth between fully on and fully off.
• Example of a **proportional** controller reducing both the rise time and the steady-state error, increased the overshoot, and decreased the settling time by small amount.
This plot shows that the derivative controller reduced both the overshoot and the settling time, and had small effect on the rise time and the steady-state error.
- **Ksteering**: used as the steering proportional gain
- **Kintegral**: used as the integral gain (bigger=faster fix)
- **Krate**: derivative: used as the rate gain
- **The Simulation Params** are used to configure the simulator settings.
- **Position(Y)**: Set the starting point of the robot
- **Angle**: Set the starting angle of the robot
- **Bias**: Set bias in the robot (error)
What The Gains Do? Another way to look at it – motor PID example

**Proportional Gain:** Gives fast response to sudden load changes and can reduce instability caused by high integral gain. This gain is typically many times higher than the integral gain so that relatively small deviations in speed are corrected while the integral gain slowly moves the speed to the setpoint. Like integral gain, when set too high, proportional gain can cause a "hard" oscillation of a few Hertz in motor speed.

**Integral Gain:** Ensures that under steady state conditions that the motor speed (almost) exactly matches the setpoint speed. A low gain can make the controller slow to push the speed to the setpoint but excessive gain can cause hunting around the setpoint speed. In less extreme cases, it can cause overshoot whereby the speed passes through the setpoint and then approaches the required speed from the opposite direction. Unfortunately, sufficient gain to quickly achieve the setpoint speed can cause overshoot and even oscillation but the other terms can be used to damp this out.

**Derivative Gain:** Can be used to give a very fast response to sudden changes in motor speed. Within simple PID controllers it can be difficult to generate a derivative term in the output that has any significant effect on motor speed. It can be deployed to reduce the rapid speed oscillation caused by high proportional gain. However, in many controllers, it is not used.
Pseudo Code

previous_error = 0
integral = 0
start:
error = setpoint - measured_value
integral = integral + error*dt
derivative = (error - previous_error)/dt
output = Kp*error + Ki*integral + Kd*derivative
previous_error = error
wait(dt)
goto start
Calculates the PID drive value. See PID Control Arduino.pdf for reference.

// you have to define your potentiometer position and motor drive point plus direction pin

Actual = analogRead(Position);
Error = SetPt - Actual;
if (abs(Error) < IntThresh){ // prevent integral 'windup'
    Integral = Integral + Error; // accumulate the error integral
} else {
    Integral=0; // zero it if out of bounds
}
P = Error*kP; // calc proportional term
I = Integral*kI; // integral term
D = (Last-Actual)*kD; // derivative term
Drive = P + I + D; // Total drive = P+I+D
Drive = Drive*ScaleFactor; // scale Drive to be in the range 0-255
if (Drive < 0){ // Check which direction to go.
    digitalWrite (Direction,LOW); // change direction as needed
} else { // depending on the sign of Error
digitalWrite (Direction,HIGH);
}
if (abs(Drive)>255) {
    Drive=255;
}
analogWrite (Motor,Drive); // send PWM command to motor board
Last = Actual; // save current value for next time
Arduino site has a library to experiment with PID
http://playground.arduino.cc/Code/PIDLibrary