Experiment 11

Cruise Control by PID

Theory

A proportional–integral–derivative controller (PID controller or three-term controller) is a control loop feedback mechanism widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an *error value* $e(t)$ as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (denoted *P*, *I*, and *D* respectively), hence the name.

In practical terms it automatically applies accurate and responsive correction to a control function. An everyday example is the cruise control on a car, where external influences such as hills (gradients) would decrease speed. The PID algorithm restores from current speed to the desired speed, with small delay and overshoot, by controlling the power output of the vehicle's engine.

Fundamental operation of PID

The distinguishing feature of the PID controller figure 1 is the ability to use the three *control terms* of proportional, integral and derivative influence on the controller output to apply accurate and optimal control. The block diagram on the right shows the principles of how these terms are generated and applied. It shows a PID controller, which continuously calculates an *error value* $e(t)$ as the difference between a desired setpoint $SP = r(t)$ and a measured process variable $PV = y(t)$, and applies a correction based on proportional, integral, and derivative terms. The controller attempts to minimize the error over time by adjustment of a *control variable* $u(t)$, such as the opening of a control valve, to a new value determined by a weighted sum of the control terms.

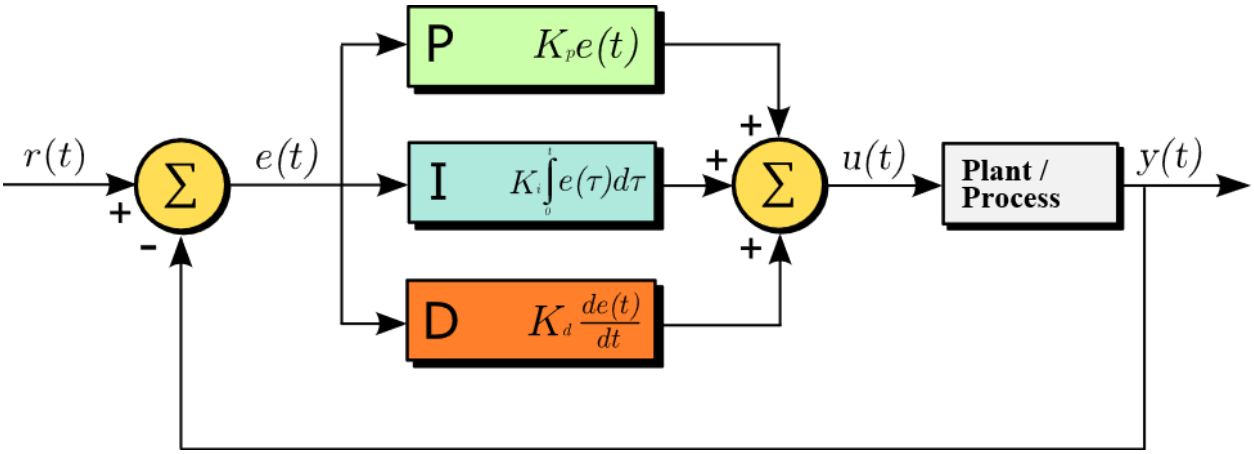

Fig 1 A block diagram of a PID controller in a feedback loop

So, in this model:

- Term **P** is proportional to the current value of the SP − PV error $e(t)$. For example, if the error is large and positive, the control output will be proportionately large and positive, considering the gain factor "K". Using proportional control alone in a process with compensation such as temperature control, will result in an error between the setpoint and the actual process value as shown in figure 2, because it requires an error to generate the proportional response. If there is no error, there is no corrective response.

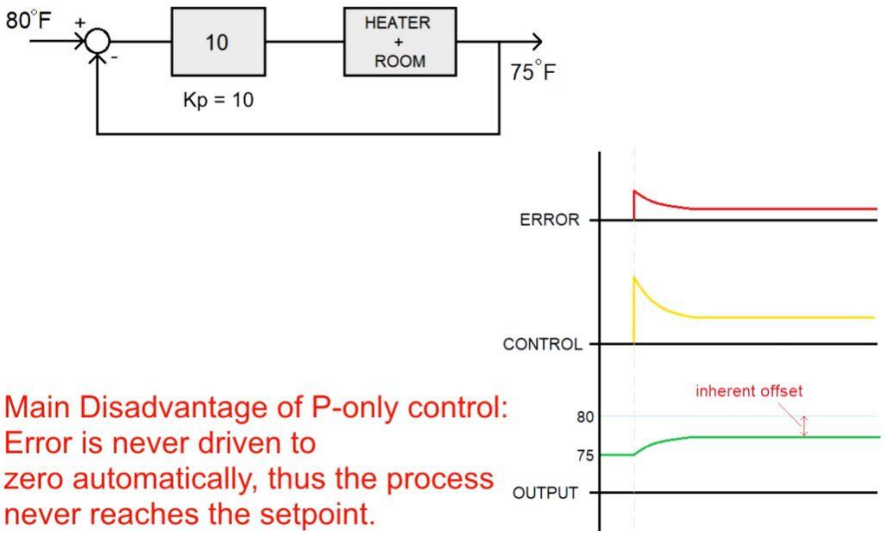

Fig 2 P-controller explaining the steady state error

- Term **I** accounts for past values of the SP − PV error and integrates them over time to produce the *I* term. For example, if there is a residual SP − PV error after the application of proportional control, the integral term seeks to eliminate the residual error by adding a control effect due to the historic cumulative value of the error. When the error is eliminated, the integral term will cease to grow. This will result in the proportional effect diminishing as the error decreases, but this is compensated for by the growing integral effect. So, in the case of a steady state error, the integral of this is a linear rise or slope and we can see why this would eliminate the error; the control signal is either accumulating for a positive error or dissipating for a negative error. And it will keep accumulating or dissipating the control signal until the error reaches zero. This is the main advantage of integral control is that it allows us to eliminate the steady state error completely. Now by tuning the Ki term we can adjust the control effort of our integral controller. Now in this case for a high Ki value, we can see that we get a steeper slope, and this would make our control signal more aggressive as in figure 3.

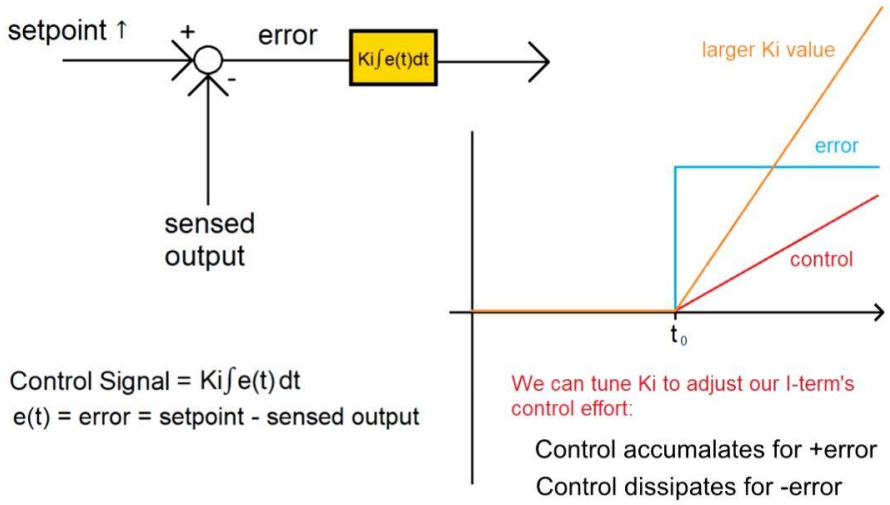

Fig 3 I-controller explaining the effect of Ki value

For the temperature control example, turning on the integral control we can see that the control signal accumulates and rises until the error is zero. We have now eliminated the offset that is appeared in p-controller and we have finally reached a set point of 80 degrees. But if K_i is too low, we end up with the large overshoot and overall a very slow response. This is bad when it comes to rejecting disturbances or constant changes in the setpoint. We can add the I-controller to P-controller to overcome the disadvantages of both, but another problem will appear which is overshoot. Figure 4 explain the step response of PI controller for room temperature example.

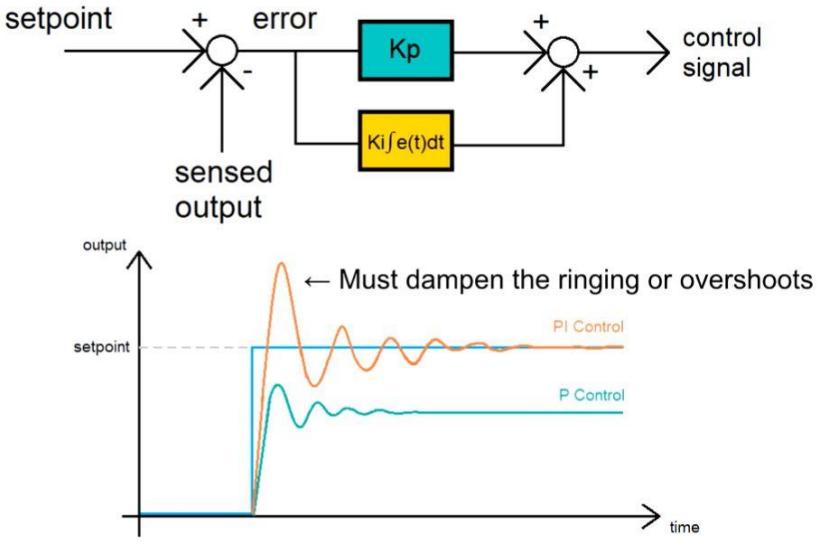

Fig 4 PI-controller explaining the overshoot

We learned that the proportional controller provided us with a fast rise time but with a steady state offset. The PI-controller solves that using the integral control to reduce the error to zero. But when neither the controller can solve is the large overshoots and rising time in order to reduce or dampen this response, we will need to add a controller and they'll quickly respond fast enough to keep the output from overshooting or undershooting the setpoint.

- Term **D** is a best estimate of the future trend of the SP − PV error, based on its current rate of change. It is sometimes called "anticipatory control", as it is effectively seeking to reduce the effect of the SP − PV error by exerting a control influence generated by the rate of error change. The more rapid the change, the greater the controlling or dampening effect. The derivative term takes the rate of change of the error as its control signal by applying the step change in the setpoint. In this case we again get a positive error in term in response produces a control signal. The main benefit of the derivative term is that for a sudden change maybe due to disturbances or a change in the setpoint the controller reacts quickly and aggressively, but this is a response to a stepper when an overshoot takes place. The error tends to be more sinusoidal as opposed to a step. Figure 5 shows the control signal is leading the error signal, before the error goes up the control goes up and before the error swings back down the control is already been there. In this way the D-control is predicting the future error over and it's in this leading behavior that allows it to reduce the overshoots before they become significant. Adding the derivative term can reduce the overshoots.

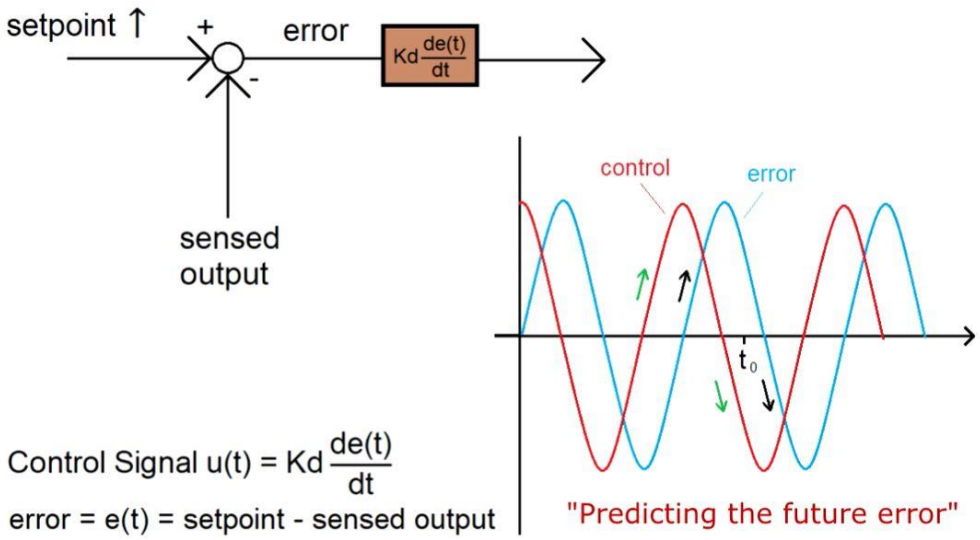

Fig 5 D-controller explaining the leading of control signal

As a conclusion we can say that the proportional term decreases rise time, Integral term eliminates steady state error and the derivative term reduces the overshoot. Figure 6 explain the step response of a PID controller.

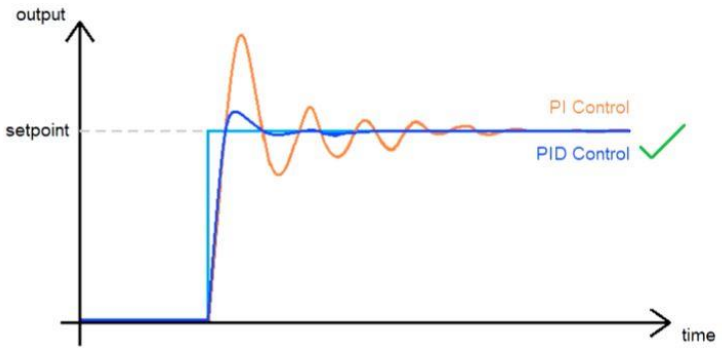

Fig 6 PID-controller explaining the step response

Mathematical form

The PID controller $u(t)$ in fig 1 can be mathematically written as shown:

$$u(t) = K_P e(t) + K_i \int_0^t e(t)\,dt + K_d \frac{de(t)}{dt}$$

where a control signal is a summation of three mathematical operations.

Part I Implementation and testing the PID algorithm

Procedure

1) Implement the PID algorithm

The mathematical equation of PID written above is a controller expressed in continuous time or in the analog domain. In order to make the signal generated by the microcontroller, we need to implement it digitally in software. The discrete time of the PID will be expressed as:

$$u[n] = K_P * e[n] + K_i * \sum_{K=0}^{n} e[K]T + K_d * \frac{(e[n] - e[n-1])}{T}$$

Where T is essentially the delta t of the continuous domain. Because in the loop we're going to call the function in intervals of t seconds, we can treat this as a fixed constant.

2) Make the PID function code

```
double sensed_output, control_signal;
double setpoint;
double Kp; //proportional gain
double Ki; //integral gain
double Kd; //derivative gain
int T; //sample time in milliseconds (ms)
unsigned long last_time;
double total_error, last_error;
int max_control;
int min_control;

void setup(){

}

void loop(){

 PID_v1(); //calls the PID function every T interval and outputs a control signal

}

void PID_v1(){

  unsigned long current_time = millis(); //returns the number of milliseconds passed since the Arduino started running

  int delta_time = current_time - last_time; //delta time interval

  if (delta_time >= T){

    double error = setpoint - sensed_output;

    total_error += error; //accumalates the error - integral term
    if (total_error >= max_control) total_error = max_control;
    else if (total_error <= min_control) total_error = min_control;

    double delta_error = error - last_error; //difference of error for derivative term

    control_signal = Kp*error + (Ki*T)*total_error + (Kd/T)*delta_error; //PID control compute
    if (control_signal >= max_control) control_signal = max_control;
    else if (control_signal <= min_control) control_signal = min_control;

    last_error = error;
    last_time = current_time;
    }
}
```

3) Build the equivalent circuit of the cruise control

For this experiment we will build a circuit to control the brightness of a LED using PID algorithm. We consider this circuit as a demo for the cruise control that is a system automatically controls the speed of a motor vehicle. The system is a servomechanism that takes over the throttle of the car to maintain a steady speed as set by the driver. The driver must bring the vehicle up to speed manually and use a button to set the cruise control to the current speed. The cruise control takes its speed signal from a rotating driveshaft, speedometer cable, wheel speed sensor from the engine's RPM, or from internal speed pulses produced electronically by the vehicle. The vehicle will maintain the desired speed by pulling the throttle cable with a solenoid, a vacuum driven servomechanism, or by using the electronic systems built into the vehicle (fully electronic) if it uses a 'drive-by-wire' system. Some modern vehicles have systems for adaptive cruise control (ACC), which is a general term meaning improved cruise control. These improvements can be automatic braking or dynamic set-speed type controls. The automatic braking type use either a single or combination of sensors (radar, lidar,and camera) to allow the vehicle to keep pace with the car it is following, slow when closing in on the vehicle in front and accelerating again to the preset speed when traffic allows.

For our circuit we consider

- Photo sensor as headway sensor
- Arduino as a control unit of the automobile
- Potentiometer as an input to set the desired speed
- Brightness of the LED as a speed. The circuit is shown in figure 7

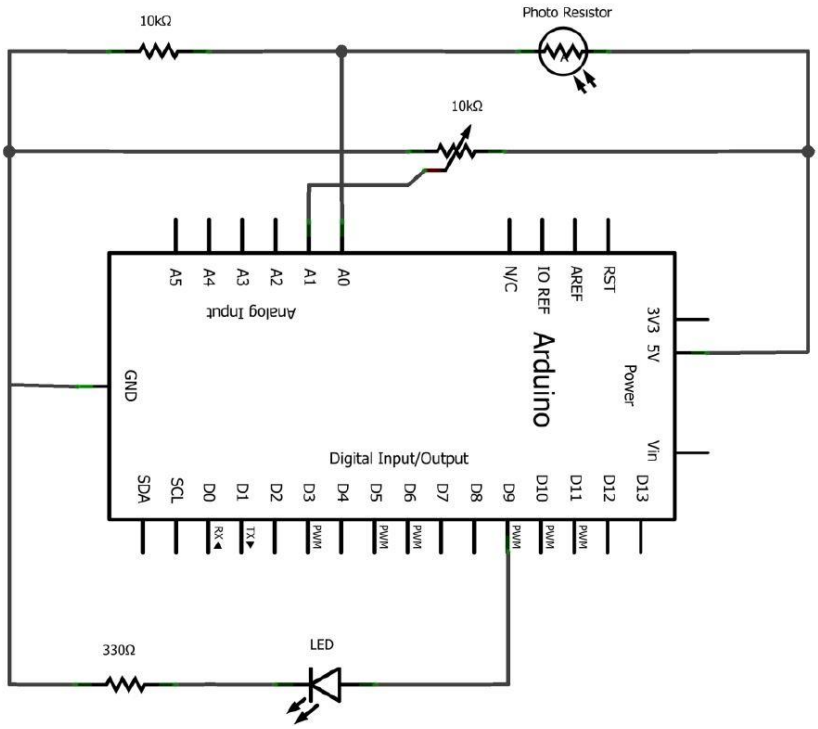

Fig 7 Circuit diagram

4) Program

After building the PID code, we save it as a function in the library so we can call it while coding the UNO to control the circuit above.

```
#include <PID_v1.h>
const int  photores = A0;          // Photo resistor input
const int  pot = A1;               // Potentiometer input
const int  led = 9;                // LED output
double  lightLevel;                // variable that stores the incoming light
                                      level

// Tuning parameters
float Kp=0;                        //Initial Proportional Gain
float Ki=10;                       //Initial Integral Gain
float Kd=0;                        //Initial Differential Gain

double Setpoint, Input, Output;  //These are just variables for storingvalues
PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
                                   // This sets up our PDID Loop

                                   //Input is our PV
                                   //Output is our u(t)
                                   //Setpoint is our SP
const int sampleRate = 1; // Variable that determines how fast our PID loop
runs

// Communication setup
const long serialPing = 500;     //This determines how often we ping our loop
// Serial pingback interval in milliseconds
unsigned long now = 0;           //This variable is used to keep track of time

// placehodler for current timestamp
unsigned long lastMessage = 0;   //This keeps track of when our loop last
spoke to serial
// last message timestamp.

void setup(){
  lightLevel = analogRead(photores);          //Read in light level
  Input = map(lightLevel, 0, 1024, 0, 255);   //Change read scale to analog
                                                out scale

  Setpoint = map(analogRead(pot), 0, 1024, 0, 255);
                                              //get our setpoint from our pot

  Serial.begin(9600);                         //Start a serial session

  myPID.SetMode(AUTOMATIC);                   //Turn on the PID loop

  myPID.SetSampleTime(sampleRate);            //Sets the sample rate

  Serial.println("Begin");                    // Hello World!
  lastMessage = millis();                     // timestamp
}

void loop(){
  Setpoint = map(analogRead(pot), 0, 1024, 0, 255); //Read our setpoint
```

```
  lightLevel = analogRead(photores);          //Get the light level
  Input = map(lightLevel, 0, 900, 0, 255);    //Map it to the right scale
  myPID.Compute();                            //Run the PID loop
  analogWrite(led, Output);                   //Write out the output from the
                                                PID loop to our LED pin

  now = millis();                             //Keep track of time
  if(now - lastMessage > serialPing) {  //If it has been long enough give us
                                          some info on serial
    // this should execute less frequently
    // send a message back to the mother ship
    Serial.print("Setpoint = ");
    Serial.print(Setpoint);
    Serial.print(" Input = ");
    Serial.print(Input);
    Serial.print(" Output = ");
    Serial.print(Output);
    Serial.print("\n");
    if (Serial.available() > 0) { //If we sent the program a command deal
                                    with it
      for (int x = 0; x < 4; x++) {
        switch (x) {
          case 0:
            Kp = Serial.parseFloat();
            break;
          case 1:
            Ki = Serial.parseFloat();
            break;
          case 2:
            Kd = Serial.parseFloat();
            break;
          case 3:
            for (int y = Serial.available(); y == 0; y--) {
              Serial.read();  //Clear out any residual junk
            }
            break;
        }
      }
      Serial.print(" Kp,Ki,Kd = ");
      Serial.print(Kp);
      Serial.print(",");
      Serial.print(Ki);
      Serial.print(",");
      Serial.println(Kd);  //Let us know what we just received
      myPID.SetTunings(Kp, Ki, Kd); //Set the PID gain constants and start
running
    }

    lastMessage = now;
    //update the time stamp.
  }

}
```

5) Compile the program and upload to Arduino UNO board
    - Open the serial monitor/plotter
    - fix the setpoint by the potentiometer
    - apply a physical light to the photo sensor
    - observe the values of the input and output while changing the brightness of the applied light
    - adjust the values of $K_P, K_i \; and \; K_D$ repeat the above step