

PID Controller application in robotics

Victor Abreu A42379353

4/5/2013

Executive Summary:

The proportional integral derivative controller is a generic control loop feedback mechanism that is widely used in industrial control systems. This application note intends to demonstrate how to calculate a PID controller into a modern mobile robot. This will be done by use of sensors to gather the desired input and use that to create a feedback system enabling for creation of a more desired output.

Introduction:

A PID controller calculates an error value as the difference between a measured process variable and a desired set point. The controller attempts to minimize the error by adjusting the process control inputs. The PID controller calculation involves three separate constant parameters, and is accordingly sometimes called three-term control: the proportional, the integral and derivative values. P depends on the present error, I on the accumulation of the past errors, and D is a prediction of future errors. This is all based on current rate of change. The weighted sum of these three actions is used to adjust the process via a control element such as the position of a control valve, a damper, or the power supplied to a heating element.

In the absence of knowledge of the underlying process, a PID controller has historically been considered to be the best controller. By tuning the three parameters in the PID controller algorithm, the controller can provide control action designed for specific process requirements. The response of the controller can be described in terms of the responsiveness of the controller to an error, the degree to which the controller overshoots the set point and the degree of system oscillation.

Objective:

To create a PID controller using a wheel encoders input to create a control system for the robots movement.

Hardware Needed:

Wheel/Optical Encoder <https://www.sparkfun.com/products/9208>

Arduino UNO Board <https://arduino.cc/en/Main/arduinoBoardUno>

Mobile Robot

Getting Started:

Initially you have to set a Wheel/Optical Encoder in order to measure the speed of the mobile robot. This is done by mounting the encoder on the shaft of the rotating object. A light source (LED) and a light sensor (Phototransistor) are used with the wheel between them such that light from the LED falls on the phototransistor only when the hole in the wheel comes in between. With this you can measure one electrical pulse produced by the phototransistor for one revolution of the wheel. You can then count the pulses in a counter for a unit period of time like a second or a minute to get rotations per second or rotations per minute.

Using this input you can begin to create a PID controller within the arduino microcontroller software.

Installing Arduino Drivers:

The software for the Arduino can be found in the IDE software from <http://arduino.cc/en/main/software>. The software must be downloaded to create a means of communication between the windows computer and the microprocessor. This will allow us to program the Arduino manually. First you must plug the board onto a USB port and wait for the windows to begin the driver installation process. If an error is displayed then the Arduino will need to be configured manually.

From the PC click Start, Control Panel, and System & Security. Click on the tab System and open the Device Manager. In the Ports (COM) subtab right click on the port named Arduino UNO (COMxx), and choose "Update Driver Software". Choose "Browse my computer for Driver Software" and navigate to the Uno driver file named ArduinoUno.inf in the Drivers folder from the Arduino IDE software that was downloaded.

Setting up the Arduino IDE:

Open the Sketch editor by opening the Arduino application. Sketch is where you apply your code and programming to the arduino board. Select the proper board by clicking tools, board, Arduino Uno. The serial ports the Arduino is using through its USB connection must be discovered. On the Arduino IDE, navigate to Tools, Serial Port. Disconnect the arduino board and re open the menu in order to discover what serial port the arduino is connected to. Whichever entry was previously selected that disappeared will be the correct serial port.

Creating the PID Code:

```

/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastErr;
double kp, ki, kd;
void Compute()
{
    /*How long since we last calculated*/
    unsigned long now = millis();
    double timeChange = (double)(now - lastTime);

    /*Compute all the working error variables*/
    double error = Setpoint - Input;
    errSum += (error * timeChange);
    double dErr = (error - lastErr) / timeChange;

    /*Compute PID Output*/
    Output = kp * error + ki * errSum + kd * dErr;

    /*Remember some variables for next time*/
    lastErr = error;
    lastTime = now;
}

void SetTunings(double Kp, double Ki, double Kd)
{
    kp = Kp;
    ki = Ki;
    kd = Kd;
}

```

Compute() is called either regularly or irregularly, and it works pretty well. To turn this into industrial PID controllers there are a few things that must be addressed. The sample time allows for the PID algorithm to function best if it is evaluated at a regular interval. The beginners PID is designed to be called irregularly and it keeps the PID with inconsistent behavior. Additionally the program will need to compute extra math in the derivative and integral. To fix this is to specify that the compute function gets called every cycle. This can be done in such a fashion.

```

/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastErr;
double kp, ki, kd;
int SampleTime = 1000; //1 sec
void Compute()
{
    unsigned long now = millis();
    int timeChange = (now - lastTime);
    if(timeChange>=SampleTime)
    {
        /*Compute all the working error variables*/
        double error = Setpoint - Input;

```

```

        errSum += error;
        double dErr = (error - lastErr);

        /*Compute PID Output*/
        Output = kp * error + ki * errSum + kd * dErr;

        /*Remember some variables for next time*/
        lastErr = error;
        lastTime = now;
    }
}

void SetTunings(double Kp, double Ki, double Kd)
{
    double SampleTimeInSec = ((double)SampleTime)/1000;
    kp = Kp;
    ki = Ki * SampleTimeInSec;
    kd = Kd / SampleTimeInSec;
}

void SetSampleTime(int NewSampleTime)
{
    if (NewSampleTime > 0)
    {
        double ratio = (double)NewSampleTime
                       / (double)SampleTime;
        ki *= ratio;
        kd /= ratio;
        SampleTime = (unsigned long)NewSampleTime;
    }
}

```

Additionally you will have to add the Derivative kick in order to fix the problems associated with the derivative term. Since $\text{error} = \text{Setpoint} - \text{Input}$ any change in Setpoint causes an instantaneous change in error. The derivative of this change is infinity. This number gets fed into the pid equation giving undesirable spikes in the output. To solve for this problem it turns out that the derivative of the Error is equal to negative derivative of input, except when the Setpoint is changing. This ends up being a perfect solution. Instead of adding, we subtract. Additional components in the code are inputted as follows

```

/*working variables*/
unsigned long lastTime;
double Input, Output, Setpoint;
double errSum, lastInput;
double kp, ki, kd;
int SampleTime = 1000; //1 sec
void Compute()
{
    unsigned long now = millis();
    int timeChange = (now - lastTime);
    if(timeChange>=SampleTime)
    {
        /*Compute all the working error variables*/
        double error = Setpoint - Input;
    }
}

```

```

errSum += error;
double dInput = (Input - lastInput);

/*Compute PID Output*/
Output = kp * error + ki * errSum - kd * dInput;

/*Remember some variables for next time*/
lastInput = Input

```

Reset windup occurs when the PID thinks it can do something it can't. For example, the PWM output on an Arduino accepts values from 0-255. By default the PID doesn't know this. If it thinks that 300-400-500 will work, its going to try those values expecting to get what it needs. The way to fix this is to actually tell the PID what the output limits are. In the code below you'll see theres now a SetOutputLimit function. Once either limit is reached, the pid stops summing (integrating.) It knows theres nothing to be done; since the output doesn't wind-up, we get an immediate response when the setpoint drops into a range where we can do something.

```

void SetOutputLimits(double Min, double Max)
{
    if(Min > Max) return;
    outMin = Min;
    outMax = Max;

    if(Output > outMax) Output = outMax;
    else if(Output < outMin) Output = outMin;

    if(ITerm > outMax) ITerm= outMax;
    else if(ITerm < outMin) ITerm= outMin;
}

```

At some point in the program you want to force the output to a certain value so you could certainly do this in the calling routine.

```
Void loop()
```

```
{
```

```
    Compute()
```

```
    Output=0
```

```
}
```

This way, no matter what the PID says, you will just overwrite its value. The way to fix this is to have a way to turn the PID off and on manually. Example code is displayed below

```

/*working variables*/
unsigned long lastTime;

```

```

double Input, Output, Setpoint;
double ITerm, lastInput;
double kp, ki, kd;
int SampleTime = 1000; //1 sec
double outMin, outMax;
bool inAuto = false;

#define MANUAL 0
#define AUTOMATIC 1

void Compute()
{
    if(!inAuto) return;
    unsigned long now = millis();
    int timeChange = (now - lastTime);
    if(timeChange>=SampleTime)
    {
        /*Compute all the working error variables*/
        double error = Setpoint - Input;
        ITerm+= (ki * error);
        if(ITerm> outMax) ITerm= outMax;
        else if(ITerm< outMin) ITerm= outMin;
        double dInput = (Input - lastInput);

        /*Compute PID Output*/
        Output = kp * error + ITerm- kd * dInput;
        if(Output > outMax) Output = outMax;
        else if(Output < outMin) Output = outMin;

        /*Remember some variables for next time*/
        lastInput = Input;
        lastTime = now;
    }
}

void SetMode(int Mode)
{
    inAuto = (Mode == AUTOMATIC);
}

```

In the last section we implemented the ability to turn the PID off and on. We turned it off, but now when you turn it back on the PID jumps back to the last output value it sent, then starts adjusting from there. This results in an input bump that we'd rather not have. The solution to this is to initialize things for a smooth transition. That means massaging the 2 stored working variables (ITerm & lastInput) to keep the output from jumping. The example code is as follows.

```

bool newAuto = (Mode == AUTOMATIC);
if(newAuto && !inAuto)
{
    /*we just went from manual to auto*/
    Initialize();
}
inAuto = newAuto;

```

```

}

void Initialize()
{
    lastInput = Input;
    ITerm = Output;
    if(ITerm> outMax) ITerm= outMax;
    else if(ITerm< outMin) ITerm= outMin;
}

```

Now to create the PID controller to work in the reverse process the signs kp, ki, and kd all must be negative. To make the process a little simpler, it helps to require that kp, ki, and kd all be ≥ 0 . If the user is connected to a reverse process, they specify that separately using the SetControllerDirection function. This ensures that the parameters all have the same sign, and hopefully makes things more intuitive. Sample codes are as follows

```

#define DIRECT 0
#define REVERSE 1
int controllerDirection = DIRECT;

if (Kp<0 || Ki<0 || Kd<0) return;

if(controllerDirection ==REVERSE)
{
    kp = (0 - kp);
    ki = (0 - ki);
    kd = (0 - kd);
}

void SetControllerDirection(int Direction)
{
    controllerDirection = Direction;
}

```

With this last addition the PID controller is complete. Now with this apply your inputs as the motor you would like to control in order to control its movement and the robot will become autonomous.