# ***PID Control Theory***

By:
Matt Krass
Email: mattkrass@gmail.com
Last updated: April 10th, 2006
Team 358
http://www.team358.org/

One of the more difficult programming problems to solve in FIRST robotics is accurate control of systems. For example in the 2006 Game ("Aim High") robots used a low-resolution digital camera with a serial output to track a green light. A series of globally accessible variables allowed the robot program to access the results of that tracking while processing. Now here's where control theory comes in, in order to target the goal with a shooter mechanism we need some way to get it there. Many people recommend PID and this paper goes over the basic math and principles behind using it. It's meant to be a general overview to help you to develop your own PID control programming. PID is made up of three main components:

P – Proportional control. The output varies based on how far you are from your target.

I – Integral control. The output varies based on how long it's taking you to get to your target.

D – Derivative control. The output varies based on the change in the error. Greater change is greater response, good for dampening spikes and jumps.

The simplest of the three is proportional. It has a varying output based on the error between current position and target position. It also has a "gain" value. This is essentially a sensitivity control; the higher this number is the more responsive the output is to the error. The formula for P is:

$$P_{OUT} = K_P * K_{ERR}$$

The formula for $K_{ERR}$ is:
$K_{ERR}$ = Target Point – Current Point

$P_{OUT}$ is the result, $K_P$ is the gain and the $K_{ERR}$ is the error. As you can see it's simply a multiplication of the error multiplied by gain. The higher the gain, the more response you get per unit of error. It suffers from a problem called "Steady State Error" and makes it unsuitable for most applications by itself. An example of this is using P to turn to a certain gyro heading on a robot. As you approach the target heading, motor power output goes down and eventually there's so little power, it stalls, and is stuck perpetually stalled short of its goal. It's still trying to move, but there just isn't enough power. Usually this is where Integral control is introduced to finish the job.

Integral (I) Control is similar to P control; however instead of the current error, it uses the integrated error. That is the sum of the error every cycle around. The longer it takes you to reach your target, the higher the integrated error becomes and the higher the output. This is most useful in completing a P based control. Using the above example of a turn, when the power level output by P control became low enough to stall, the integrated error starts to build up and keep the robot turning, as it approaches the target the P error continues to drop, and generally the I output will cause an overshoot and then drive it back. The formula for I control is:

$$I_{OUT} = K_I * I_{ERR}$$

The formula for $I_{ERR}$ is:
$I_{ERR}$ = Previous $I_{ERR}$ + $K_{ERR}$

By using both you will almost certainly reach your target, once your gains are properly "tuned" to the proper values. Tuning is a fairly simple procedure, but it takes time. First, set both your I and P gains to zero. Then increase P until the responsive is where you like it, and you're getting a consistent steady state error. Then start increasing I slowly. Your goal is to have it drive smoothly to your target point. Most of the time the best you can get is for it to overshoot the target slightly and back towards it.

Another problem is sometimes the optimal gain is not a clean integer number but a point between them. It's well known however that the

controller's processor does not enjoy floating point math so much. There is a trick to this though. Since gains are constants they do not need to be a variable, simply a pre-processor directive. An example follows of a very basic P control function.

```
#define KP 5 / 10

unsigned char PCntrl(signed int target, signed int current)
{
      signed int error = 0;
      signed int p_out = 0;

      error = target – current;

      p_out = error * KP;

      if(p_out > 127)
            p_out = 127;
      if(p_out < -127)
            p_out = -127;

      return (unsigned char)(p_out + 127);
}
```

```
This function is used as follows:

      pwmXX = PCntrl(100,Get_Analog_Value(rc_ana_in01));
```

This uses basic P control to drive a motor based on a potentiometer value connected to Analog Input 1, as it approaches a value of 100, it will slow down, and most likely stop shy of its target due to steady state error as described above. It also uses a trick to use a fractional gain. Pre-processor directives such as #define are essentially find and replace commands for the compiler. As such when the p_out line was processed, it was not read as shown, the compiler replaced KP with its defined meaning and processed this instead:

```
            p_out = error * 5 / 10;
```

Now the error was multiplied by 5 and then divided over 10 giving a very close approximation of the error multiplied by a gain of 0.5, as close as possible without the use of floating point math and is more than adequate. This is considerably faster and easier than the use of floating point operations.

Another problem is integral windup. This is common when robots are disabled for a period while the PID loop is trying to maintain a position. If left along a integral error will build up rapidly and cause a rapid and unstable reaction when suddenly released by enabling. A check of the competition port disable status before increasing the integral error is always a good idea. Also, setting a maximum integral error is good, to prevent the device in question from "over-responding" to an error.

The `if()` block in the above example is used as a basic sanity check to prevent the outputs from exceeding the limits of the `pwmXX` variables and causing a wraparound condition which can produce random and erratic results. I used signed math because it's neater than dealing with negative errors and all positive outputs, a simple typecast and addition of 127 to the number at the end makes it IFI compatible.

Now finally, the third musketeer, Derivative control, D, sometimes called Delta control because it's actually driven by the change, or delta, of the $K_{ERR}$. As such it can be used to react to sudden changes in error, and is good for maintaining a certain position or velocity on a closed loop system. The formula for D is:

$$D_{OUT} = D_{ERR} * K_D$$

The formula for $D_{ERR}$ is:

$$D_{ERR} = K_{ERR} - \text{Previous } K_{ERR}$$

Full PID control is simply the combination of the results of all three formulas. Different combinations of the formulas are good for different situations. For example, PI is good at getting you to a spot quickly, but is not the most accurate; PD can reach a spot fairly quickly and hold it fairly well. PID can accurately maintain a position, but is not the fastest or gentlest.

A common application in the 2006 season for PID controls were targeting based on the vision system. The included CMUCamII could provide a numeric value indicating which pixel the green mass of the light was centered on. Using the center pixel as a target point you could compute an error and tune the gains to control a gimbal motor, or even a rate of turn for a robot base to point at the goal. An example of such follows:

```
#define KP 15 / 10
#define KI  5 / 10
#define KD 10 / 10

unsigned char camera_pan_track()
{
      signed int error;
      signed int delta_err;
      signed int p_out;
      signed int i_out;
      signed int d_out;
      signed int output;

      static signed int integral_err;
      static signed int prev_err;

      error = (int)T_Packet_Data.mx - PAN_TARGET_PIXEL_DEFAULT;
      delta_err = prev_err – error;
      integral_error += error;

      if(integral_err > 200)
            integral_err = 200;
      if(integral_err < -200)
            integral_err = -200;

      p_out = error * KP;
      i_out = integral_err * KI;
      d_out = delta_err * KD;

      output = p_out + i_out + d_out;
      if(output > 127)
            output = 127;
      if(output < -127)
            output = -127;

      prev_err = error;

      return (unsigned char)output + 127;
}
```

This example uses PID to drive a motor to rotate the camera to track the light by panning only, this is *not* for servo's, nor is it complete, it's meant to illustrate the concept. It does not include countermeasures against Integral Windup beyond a simple sanity check on the size of integral error. Take note that the KP, KI and KD constants must be the last thing in the equation to ensure they are properly parsed through order of operations.

Possible applications for PID control are accurate position control, gyro based turns, velocity control, closed loop steering. With a small arrangement of sensors such as the camera, potentiometers, a gyro and encoders you can use PID and closed loop control for many aspects of your robots control such as:

- Closed loop steering, use gyro or encoders to adjust power to motors to drive straighter
- Closed loop turning, make precise quick turns using a gyroscope / angular rate sensor in autonomous.
- Drive a certain distance in a straight line using encoders.
- Maintain position by maintaining a certain encoder tick count, this causes the motors to fight back against being pushed with the precise power output required.
- Driving a tracking/shooting gimbal using the camera in the 2006 game.
- Maintain rotational velocity of impeller or collector wheels for balls by adjusting speed based on a target number of encoder ticks over time.
- Precision position of manipulators using encoders or potentiometers.

There are many more applications, and I’m sure there’s one I can’t even think of. I hope this document was of use and help to many of you FRC programmers out there.