PID CONTROL LOOP

By Collin Stiers
VEX Team REX 1727B Dulaney High School

I. Introduction: A PID control loop is used to maintain a constant position or value on a sensor in changing circumstances, or to move to a position or value from a current position. The PID loop operates on three terms: P, I , and D, which stand for proportion, integral, and derivative, respectively. You can also make control loops based on any combination of these terms; not all 3 terms are required. The entire PID loop operates on a value known as **error.** Error is the difference between the desired state and the current state. The current state has to be measured by a sensor as it is a representation of where we are now. This value needs to be something that is measured. A desired state value will be inputted by the user; you can tell the control loop what you want the value to be. We will call this value "**target.**"
Error = Target - CurrentState

here is an introductory video on PID loops, I recommend you watch this

https://www.youtube.com/watch?v=NVGKVpcHL7U

II. Terms:

The first term is called the **P** term, or the **proportional term** because it is a proportion of the error. The proportional term value is calculated by multiplying the **constant of proportion**, or **Kp** by the current error. The value of Kp is determined experimentally. Therefore:
P = Kp * Error

Next is the **I** term, or the **integral term**, and it is the **integral of the error**. That is, it is the **total error** that has accumulated over the entire time the loop has been running. The total error for time step X would be: (total error of timestep X-1)+ (error of timestep X). We then multiply this value by the **constant of integration**; we do this because the total error will often be too high, making the **I** term too high to be useful unless we multiply by the constant. We call this constant **Ki**. Therefore:
I = Ki * totalError

As a note, the values for the integral term will often accumulate out of control or become so high that they override all the other terms, this is called **integral accumulation** and can be a bad thing. Controlling integral accumulation is one of the most difficult things to do, and we will discuss this later in section **VII.**

The final term is the **D** term, or the **derivative term**. It operates on the **derivative of error**—that is, it is the current **rate of change of error**. The **derivative** for time step X would

be: (Error for timestep X) - (Error for timestep X-1). Again, we multiply this by a **constant of derivation** or **Kd.** Therefore:

Derivative = CurrentError - LastError

D = Kd * Derivative

Note: We will go over why each term exists and how to use it in the next sections. We will also go over how to determine the values for **Kp, Ki,** and **Kd** in section **VIII.** This is called **tuning** the PID loop.

Finally:

Power = P+I+D

where **Power** is the value we send to our motors (or whatever device we are using to change our state).

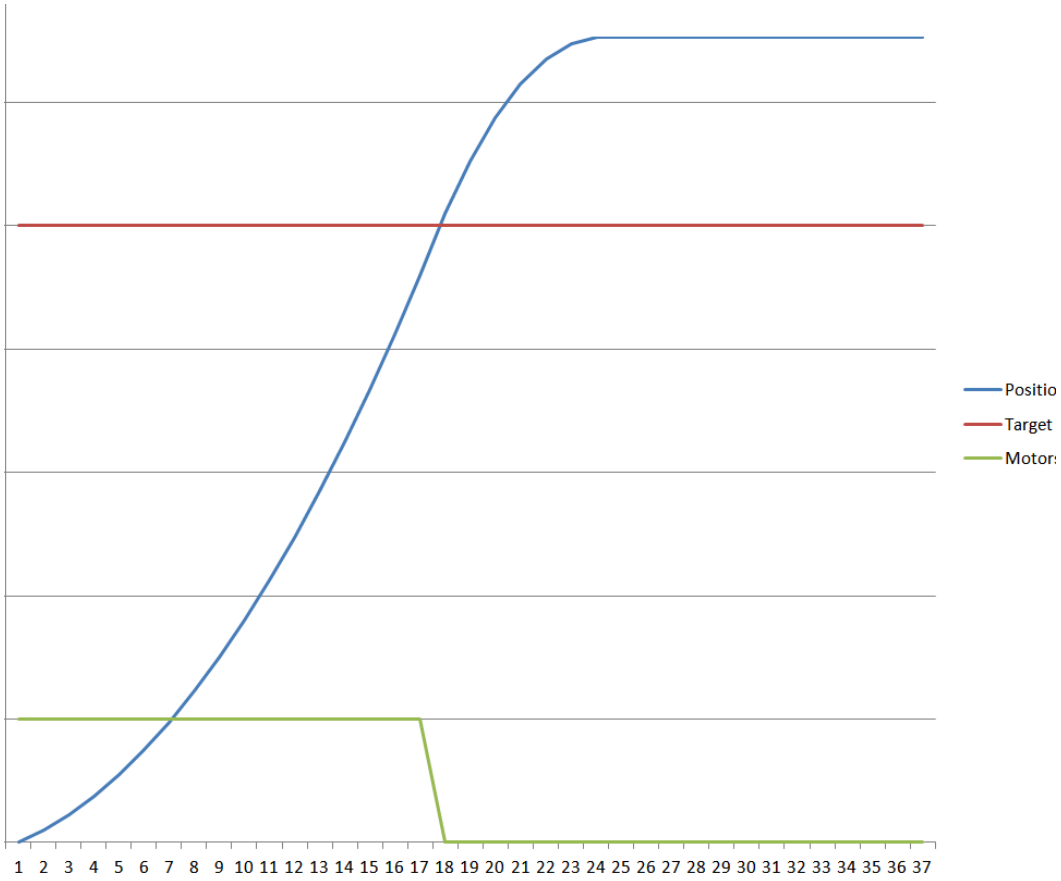
III. Why use a PID loop?

The question now is, why do we use this complex system, and what is the benefit of all these mathematics to calculate values?

Let's start with a simple example: you have a robot and you want it to move forward X distance, in our example 1000 units, and you have a sensor that will tell you how far you have gone. So what you could do is write this code:

```
While(Distance < Target){
motor = 100;
}
motor = 0;
```

The problem with this solution is that once **distance = target,** you turn the motors off. However, the robot is still moving and will overshoot the target. See below, where the blue, red, and green lines show the robot's position, target position, and motor power, respectively.



As you can see, we overshot our target, leading to significant error with where we wanted our robot to be, so now let's add in each term and build a PID controller to see the benefits.

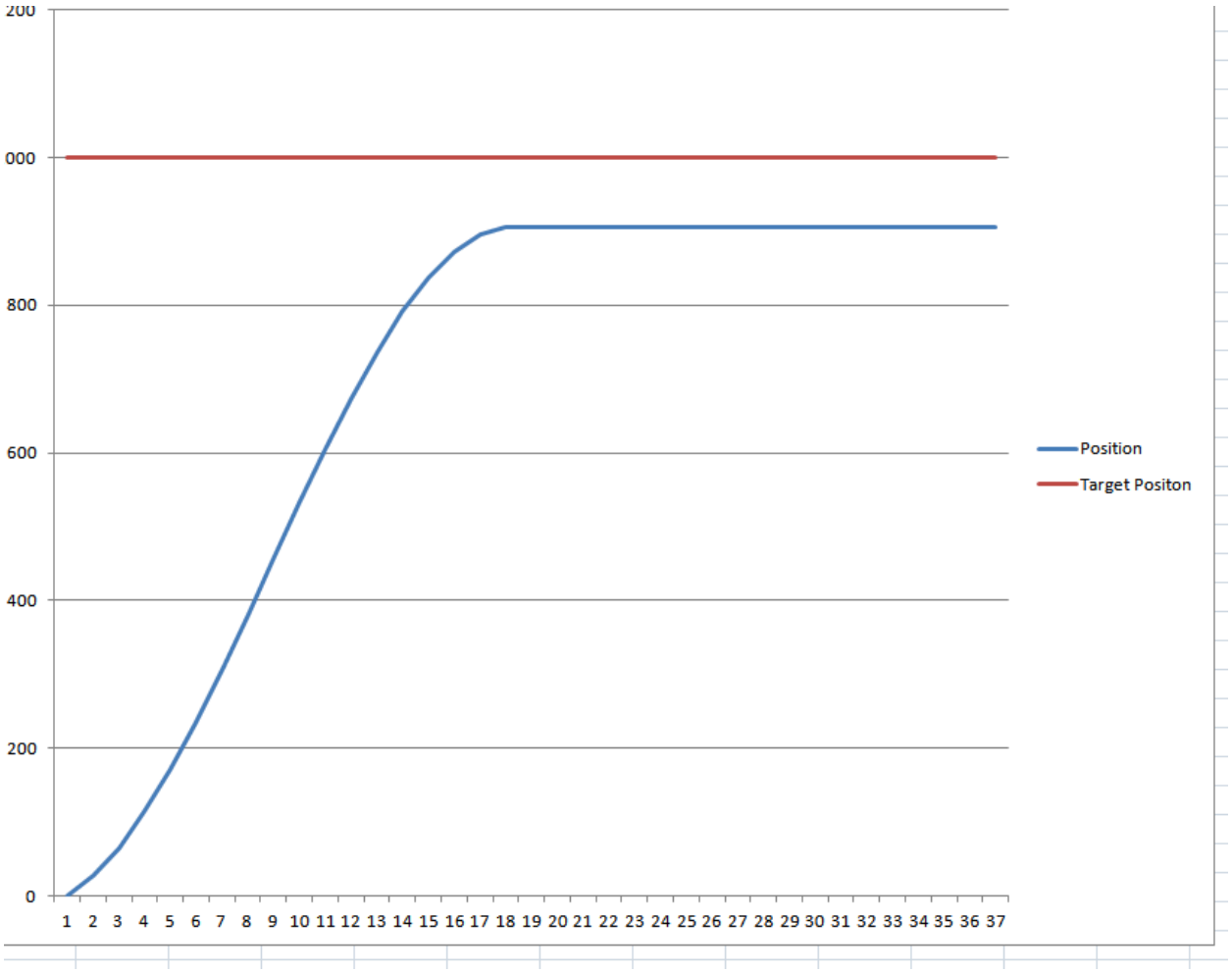
IV. Proportional Term

_____The proportional term of the PID controller is a large improvement on our previous example. Often, P controllers are used all on their own, as they can work well in simple cases and are simpler to program, but a full PID loop will be faster and more accurate when tuned correctly.

Let's demonstrate again by using the same example as from before. Our robot is trying to drive forward to distance target, but now we can use the proportional term:

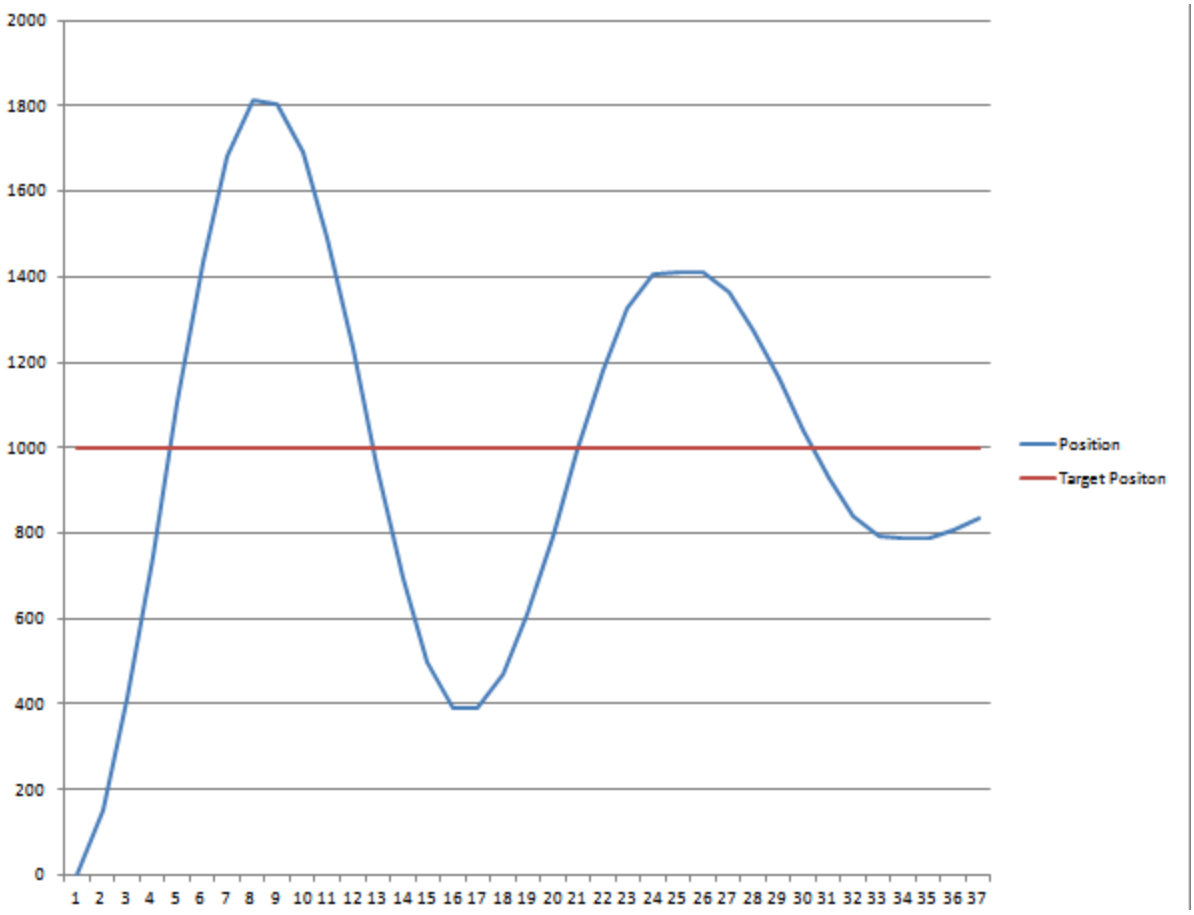
```
while(true){
error = target - distance;
power = error*Kp;
}
```

Now let's look at this graph of the robot's position using this code, where **Kp** is 0.027**:**



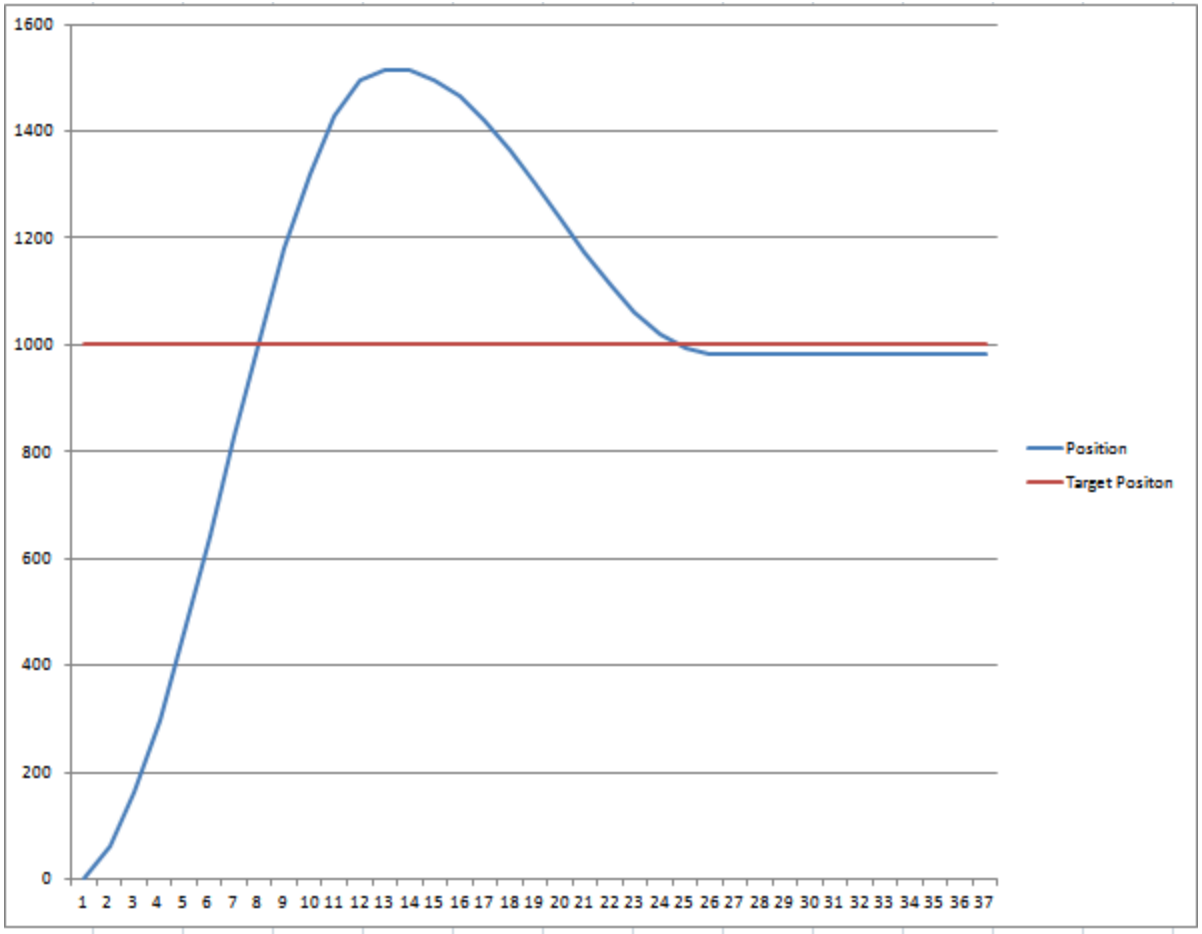
If **Kp** is too low, we get something that looks like this, at some point **Power** is to low to actually move the robot because of some resistance or friction in the system so we undershoot the target. However, having **Kp** too high will also result in problems; it will cause an oscillation about the target.

Let's take a look at another graph running the same code, but this time with a **Kp** of 0.15:



Oscillation, in terms of our example, means that the robot will move back and forth over the target location and will take a long time to settle into a stable position.

A properly tuned P loop will overshoot the target by a little bit, but then settle quickly back into the target position with little oscillation. Let's take at a look of a graph for a well-tuned P loop using a **Kp** of 0.06:



You can see that there is still one oscillation and some error at the end, called **steady-state error** because that is the error at the point that the state is "steady" and not changing. Here the P controller has led to much better results than the non-PID controller. Soon you'll also see how adding the derivative and integral terms can improve the loop even further.

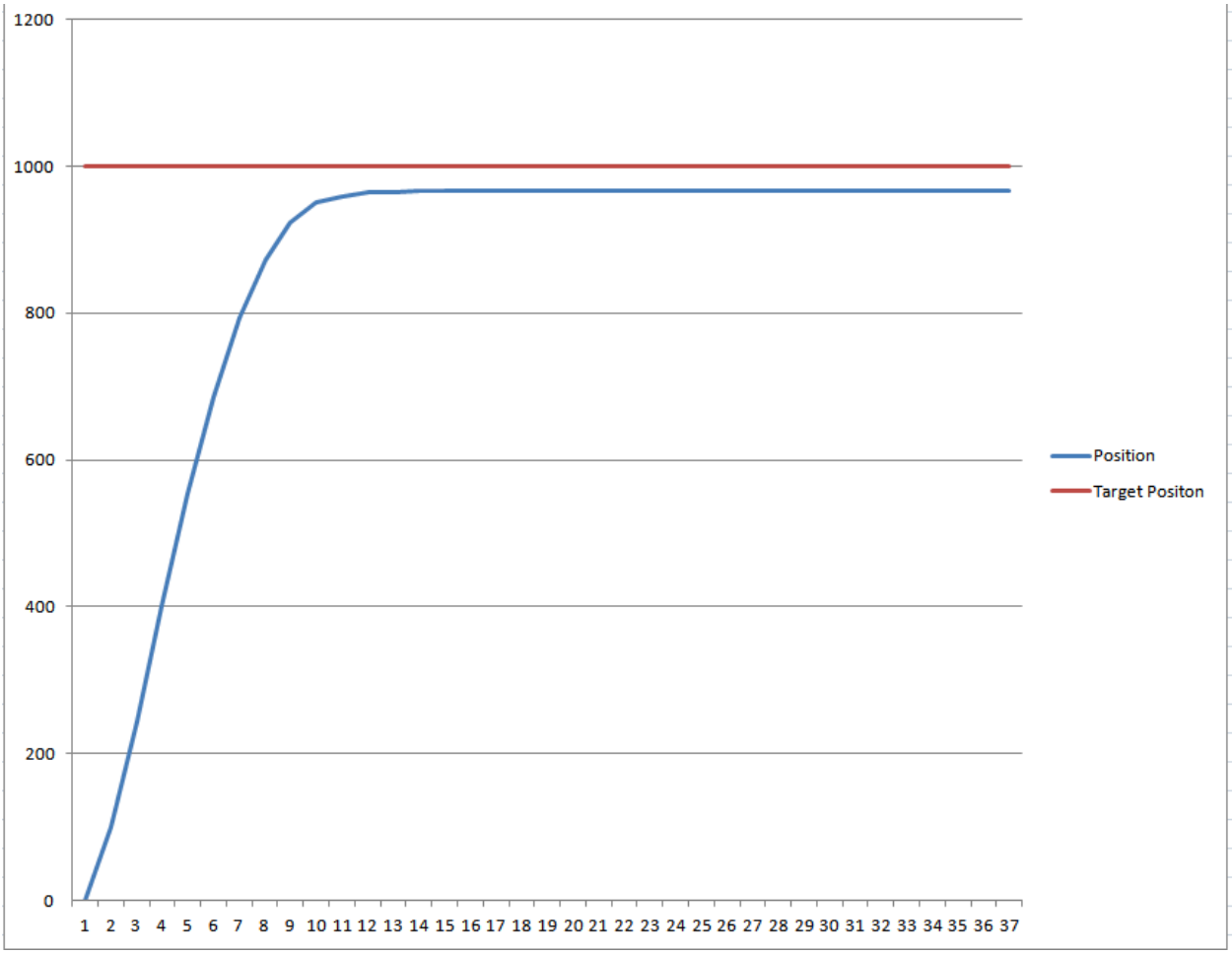
V. Derivative Term

The next part of the PID loop that is usually added is the Derivative term, the PD controller offers some advantage over the loop that only contains a P term.

Remembering back to the last section
Derivative = CurrentError - LastError
D = Kd * Derivative

This means that if we are approaching our target, the current error will be less than last error so the **derivative term** will be negative, meaning that it is actively subtracting from the rest of the loop. As the **derivative** gets larger, the **D term** will subtract even more from the motor power level. The reason we want this to happen is so that the **D term** will prevent the robot from approaching the target too fast, helping prevent overshoot. Once you add the D term and tune

the value for **Kd**, you can eliminate overshoot while also getting to your target faster than the P loop. Here is a graph for a **Kd** of 0.32 and a **Kp** of 0.1:



You can see that the robot still undershoots the target by some, but the steady-state error is not as high as with the P loop. The PD loop also reached its final position much quicker. Using the P loop, the robot reached its final position in about 27 timesteps, while the PD loop enabled the robot to reach its final position in about 8 timesteps. By adding the derivative, we were also able to increase **Kp**, because the derivative was able to keep us from overshooting our target and/or oscillating.

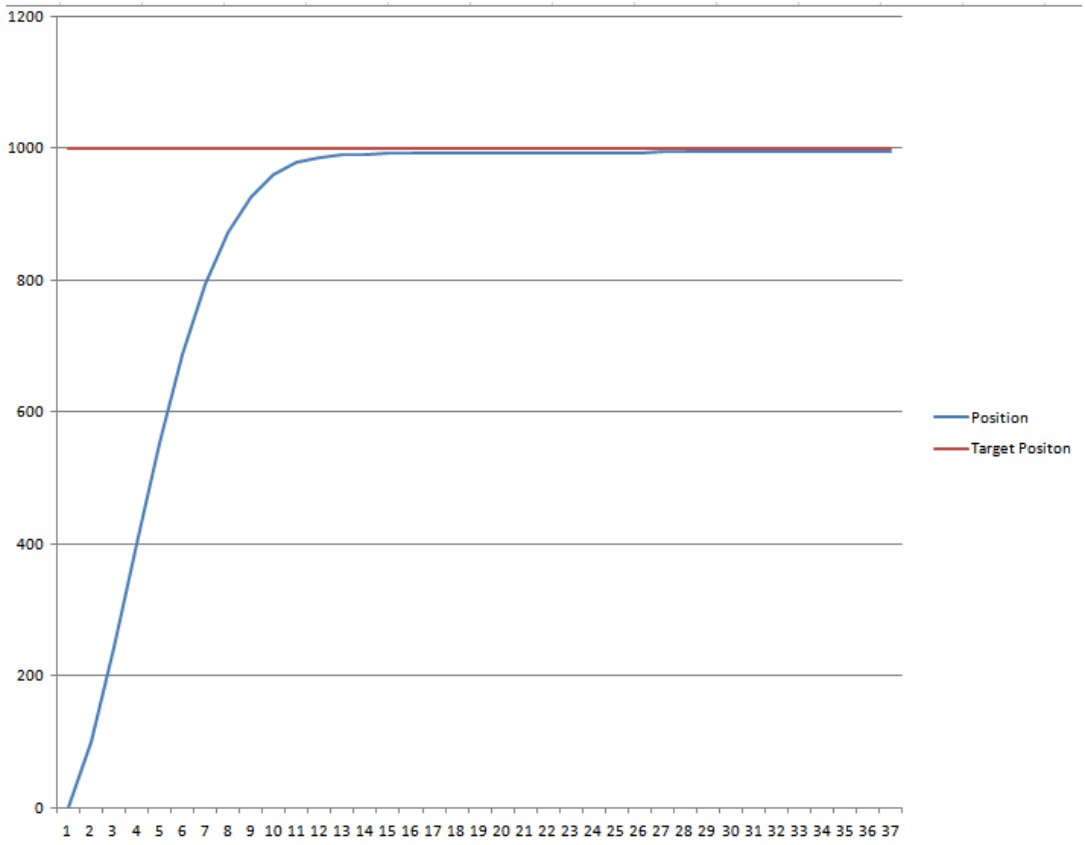
Below is basic pseudocode for the P&D loop:

```
Kp = .1;
Kd =.32;
lasterror = error;
while(true){
Derivative = error-lasterror;
P = error * Kp;
D = Derivative * Kd;
Power = P+D;
lasterror = error;
motors = power;
wait1Msec(20);
}
```

VI. Integral Term

The purpose of the integral term is to fix the small undershoot that we had at the end of the of the P&D controller. The **integral** is the accumulated error over the life of the program. During the loop we add current error to the total error. This means that error will accumulate. So if we look back at the last example, the robot undershot the target because the P term at that point was not strong enough to move the robot against the friction and inertia in the system. However, using the **integral term** if we have X error, the first time through the loop **integral** will be X; the next time it will be 2X, then 3X, then 4X, and so on. Eventually the **integral term** will get large enough to move the robot. Be careful with this term, however, as it can easily get out of control and cause oscillations, or cause damage if the robot gets stuck.

Once we added the I term we have a full PID loop, and get the graph below

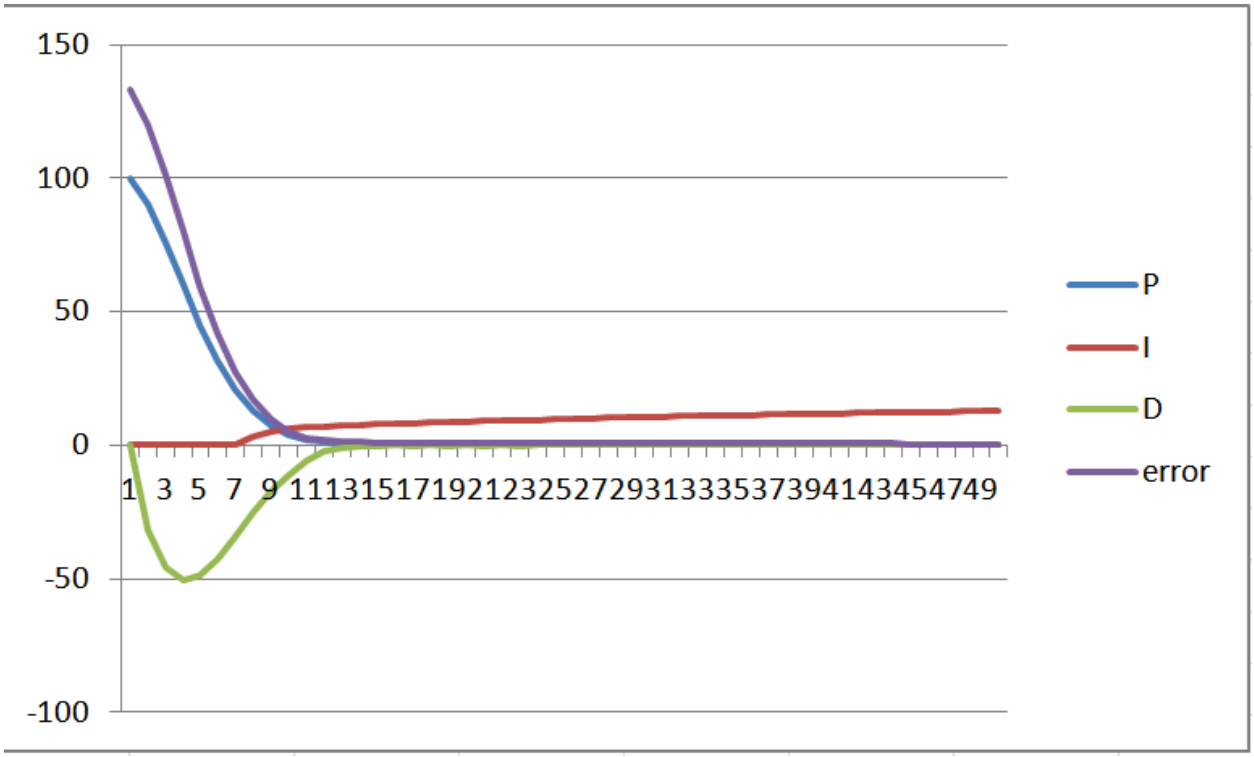


As you can see, the robot very quickly reaches a location near the target with very low error. Then, it slowly pulls itself into a range of almost zero error. **Ki** in this example is 0.026.

Below is the pseudocode for the full PID loop:

```
Kp = .1;
Kd =.32;
Ki = .026;
lastError = error;
totalError = 0;
while(true){
totalError = totalError+error;
Derivative = error-lastError;
P = error * Kp;
D = Derivative * Kd;
I = totalError *Ki
Power = P+D+I;
lastError = error;
motors = power;
wait1Msec(20);
}
```

Below is a graph of the P, I and D terms over the life of the code. Error is also shown, but it is scaled so that it fits on this graph.



You can see here that P falls rapidly along with the error. As P falls quickly, D becomes larger and decreases, causing the robot to begin slow down so that it does not overshoot the target. The I term jumps in after the P and D terms have done most of their work and reduces the error almost if not completely.

VII. Controlling the Integral Term

_____As we have mentioned here before, the **Integral Term** can spiral out of control (this is called **integral wind-up**). There are several methods to control this. The first is what is called the **Integral Active Zone**, where the integral is only allowed to accumulate while the error is within a certain range. We have to be sure that this value is a value that we can get to without using the integral term. For example, here is some pseudocode that prevents the integral from accumulating unless the error is below 200:

```
if(error<200){
        Totalerror+=error;
}
else{
Totalerror = 0;
}
```

The next thing we can do is add a cap for the integral. Imagine the case where we have an arm that we want to raise with the PID loop, but the arm gets jammed. In this case the integral term would continue to accumulate forever, until it was jamming the arm at max power into whatever is stopping it. This would stall the motors, potentially causing damage. To prevent this from happening, we can add code to ensure that **integral** does not get too high.
Pseudocode:

```
if( I> 50){
I = 50;
}
```

Make sure that the maximum value is enough to cause a change in the system (i.e. enough to move the arm).

The final control method is only useful in some cases. Let's go back to our example of a robot driving toward a target. When you reach the target, then the error is zero, and at that point you don't want to move the robot. Otherwise, the robot would overshoot, causing the integral to de-accumulate until it caused the robot to overshoot in the opposite direction. In other words, the robot would begin to oscillate. This can be solved by setting the integral to zero whenever the error is zero, causing the robot to stop before any oscillations can occur.

However, sometimes you don't want the motor power to be zero if the error is zero. For example, if you have an arm that would sink to the ground if power were zero, you would want a constant input from the I term in order to keep the arm in position. Therefore, this method should only be used if there are no external constant forces acting on the robot.
Here is the pseudocode for this method of preventing integral windup:

```
if(error == 0){
totalerror = 0;
}

//we can combine two of these control methods to make this code:

if(abs(error) < 200 && error !=0){
totalerror += error;
}
else{
totalerror = 0;
}
```

VIII: Tuning a PID loop

You must individually tune every PID loop you make. T**uning** the PID loop refers to finding the correct values for **Kp**, **Ki**, and **Kd**. There is no way to find these values except to determine them experimentally. You cannot use values from a different system.

To tune a PID loop, the first thing to do is to set all of the constants to zero. Then set **Kp** to a very low value, and increase it very slowly. The smaller the increments you use, the more accurate your values will be when you are done.

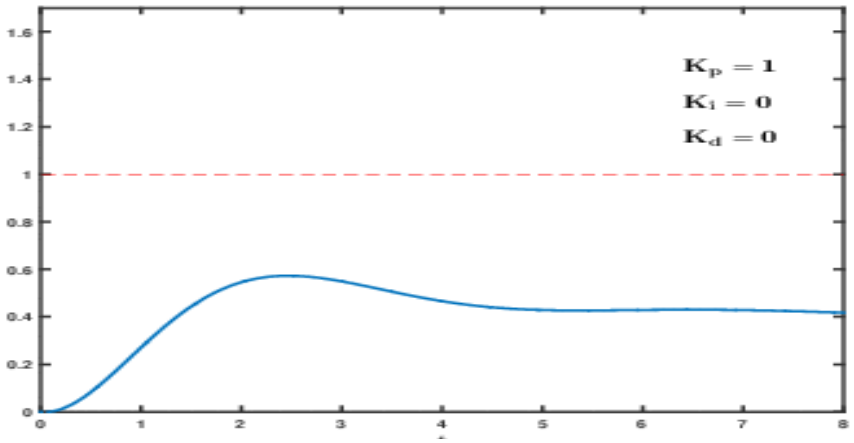
Keep increasing **Kp** until you get an oscillation about the target (similar to the graph shown in section **IV**), then increase **Kd** until the oscillation stops.
Once you reach this step, repeat the process of increasing **Kp** until you get an oscillation, then increasing **Kd** until it stops.

Repeat this as many times as you can. Eventually you will find that you get an oscillation that you cannot stop no matter how high you increase **Kd**. Once you get to this point, you need to go back to the last values for **Kp** and **Kd** that worked. These will be your final values for **Kp** and **Kd**.

Hopefully at this point you will be reaching your final position quickly with little steady-state error. Now you need to slowly increase **Ki** until the loop runs smoothly without steady-state error. The larger you can make **Ki**, the better, but make sure you don't compromise the stability of the loop, tuning the **Ki** value is very touchy and it can be somewhat subjective as to what the best value might be. Try to find a value that ends the loop quickly, doesn't cause oscillations, and ends with very low error.

Values for **Ki** will often be very, very low, as small as 0.00001. You may need to start with very low values for **Ki**, and work your way up very slowly.

Here is a good animation of the effects of changing Kp, Ki, and Kd:



If the animation doesn't work,click here
https://drive.google.com/file/d/0B1LLlSCW4Hm5MndnX01idjBZRGc/view?usp=sharing

Another method to tune a PID loops is the ziegler - Nichols method. First set Ki, and Kd to 0, then start with Kp at zero and slowly increase Kp until the output developes an oscillation. This value for Kp gets saved as Ku, then measure the period of the oscillation and save this value as Tu, then use the following table to calculate the rest of the constants

Control Type	K_p	K_i	K_d
P	$0.50K_u$	-	-
PI	$0.45K_u$	$1.2K_p/T_u$	-
PID	$0.60K_u$	$2K_p/T_u$	$K_pT_u/8$

IX. Additional Info

below is code written in RobotC that forms a basic PID loop. It is designed to keep a flywheel shooter spinning at a target speed.

```
task shooterControl(){
	float kp = .22;				// proportional konstant
	float ki = .000004;			//konstant of integration
	float kd =  .07;			//konstant of derivation

	float current = 0;		//value to be sent to shooter motors
```

```
	float integralActiveZone = 2;  // zone of error values in which the total error for the
integral term accumulates
	float errorT;                                          // total error accumulated
	float lastError;                                    // last error recorded by the controller
	float proportion;                                          // the proportional term
	float integral;                                            // the integral term
	float derivative;                                          // the derivative term
/*//////////////////////////////////////////////////////
	NOTE:
	Error is a float declared at global level, it represents the difference between target
velocity and current velocity

	power is a float declared at global level, it represents target velocity

	velocity is a float declared at global level, it is the current measured velocity of the
shooter wheels

	//////////////////////////////////////////////*/
	while(true){
		float error = power - velocity; // calculates difference between current velocity and
target velocity

		if(error <integralActiveZone && errorR != 0)// total error only accumulates where
/                                                          //there is error, and when the error is
                                                            //within the integral active zone
		{
			errorT += error;// adds error to the total each time through the loop
		}
		else{
			errorT = 0;// if error = zero or error is not withing the active zone, total
/                                  //error is set to zero
		}

		if(errorT > 50/ki) //caps total error at 50
		{
			errorT = 50/ki;
		}
		if(error == 0){
			derivative = 0; // if error is zero derivative term is zero
		}
		proportion	=	error		* kp; // sets proportion term
		integral		=	errorT		* ki;// sets integral term
		derivative	=	(error - lastError)* kd;// sets derivative term
```

```
		lastError = error; // sets the last error to current error so we can use it in the next
loop

		current =  proportion + integral + derivative;// sets value current as total of all
terms

		motor[shooter2] =  motor[shooter3] = current; // sets motors to the calculated
value

		wait1Msec(20);// waits so we dont hog all our CPU power or cause loop instability
	}
}
```
