How To PID

AN OVERVIEW FOR VEX ROBOTICS

FOURTH EDITION

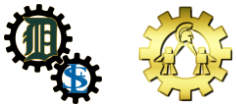
RAY SUN

21 AUGUST 2017

IN COLLABORATION WITH

VRC 6526D 2013-2016

VRC 6526F 2015-2017



DAMIEN HIGH SCHOOL ROBOTICS

DAMIEN HIGH SCHOOL

MRS. CHARITY MARICIC
MR. DOMINIC MARICIC

DAMIEN HIGH SCHOOL ROBOTICS

DAMIEN HIGH SCHOOL ENGINEERING
DAMIEN HGH SCHOOL COMPUTER SCIENCE

Contents

Summary

In this text the author, one of the founding members of the Damien High School Robotics organization and one of those who facilitated Robotics' stellar triumph in the VEX Robotics Competition (VRC) during its third year, explains PID control (the proportional-integral-derivative feedback controller) in non-technical terms to aid less experienced members of Robotics in their robot programming endeavors. PID control, as utilized in Damien Robotics, is a method of controlling a motor-driven robot mechanism with use of sensors. PID attempts to eliminate error, the deviation of the measured value of some quantity (e.g. the position of a motor) from the desired value, by iterative summation of three quantities: P, I, and D, which depend on the value of the error. This technique enables precise control of motor position and speed and has become one of the primary factors responsible for the phenomenal rise of Damien Robotics. Any Damien roboticist aspiring to excellence in programming will benefit from a mastery of PID.

Foreword

This text is intended to introduce PID and its applications in VEX Robotics and the VEX Robotics Competition (VRC) to Damien High School Robotics members with little or no experience with calculus. The text encompasses concepts and code that I learned in my junior and senior year at Damien as a programmer and systems engineer for 6526D, the first VEX Robotics Competition (VRC) team founded at Damien.

This text will present a non-rigorous approach suitable to those without formal experience in control theory or control systems. However, I will assume a basic familiarity with ROBOTC commands and control structures (while loops, etc.).[1] I will also assume knowledge of Algebra 1, and a tad of Algebra 2. Exercises are present for more advanced readers, who may wish to skip the sections on calculus.

I will use examples and code to arrive at PID gradually rather than presenting the results of PID without proof. I will use ROBOTC (without PLTW Natural Language) only, as ROBOTC is the standard programming language used in Damien Robotics to this day.[2]

In short, read this if you want to learn a revolutionary way to program for VEX, one that has taken Damien to Worlds, and you are not taking engineering in college yet. I will describe the necessary calculus concepts when needed so that you can understand the principles of PID even without prior calculus knowledge.

[1] A fine place to learn about any code in this text that might be unfamiliar, and perhaps a bit more than that: < http://help.robotc.net/WebHelpVEX/index.htm >

[2] There exist others – some of which are free, and/or much more capable than ROBOTC (Python for VEX, PROS). However, the core of PID in code are the same no matter the programming language.

Damien Robotics and PID

THE DISCOVERY

I first stumbled into PID in the aftermath of my second season of robotics, the second year of Damien Robotics. Back then, we had four teams – 6526D through 6526G – a lot less aluminum, even less programming skill, no integrated motor encoders (IMEs), gyroscopes, or accelerometers, no trophies from States, Nationals or Worlds, no teams having attended States … and no knowledge of PID. All that was to change, of course.

The two leading Damien teams at the time – 6526D (mine) and 6526E (with a remarkable development and programming department spearheaded by one Mr. Daniel Enright) – were beset with the usual problem that robots with bar lifts (or N-bar linkages – whatever, those parallelogram lifts, you know) face: Our lifts could not hold themselves up against gravity. Even with the help of rubber bands, our drivers found it necessary to tap the lift controls to keep our lifts aloft when extended while carrying scoring objects. As every driver knows, the necessary repetitive tapping of the up-button to keep the arm up against gravity is despicably annoying.

At this point in the season, late March, I had finished that year's tour with an acceptable performance in the league-concluding tournament at the California High Desert League.[3] Mr. Enright and his team had qualified for Nationals – the CREATE U.S. Open Robotics Championships[4] – for the first time in the history of Damien Robotics. So the stakes were higher, and we (6526E and what was then the cutting edge of Damien Robotics' technical expertise backbone) needed a solution.

That solution turned out to be PID. Initially it sounded too good to be true – a piece of robot code, just about ten lines long, with a while loop and nothing fancier than basic math and if-condition checking. It promised to permit a motor be controlled by position, and not power – instead of "run at a power level of 127"[5], "go here". In addition, PID claimed to resist efforts to push the motor away from the position – unheard of to us neophytes – with robotics programming techniques.

I do not remember who – whether it was Mr. Enright or me – was the first in Robotics to discover PID or feedback control theory.[6] I do know that PID had been applied in VEX Robotics before (sources exist on the

---

[3] A VRC league consisting of six events – a practice scrimmage, four qualifying competitions, and a tournament – held each year at Sultana High School in Hesperia, CA from September to February (or so). Damien Robotics has competed there for the first three years of its existence. Now we compete in a league at the Martin Luther King Jr, Memorial High School in Riverside (whose mentor Mrs. Maricic knows well).

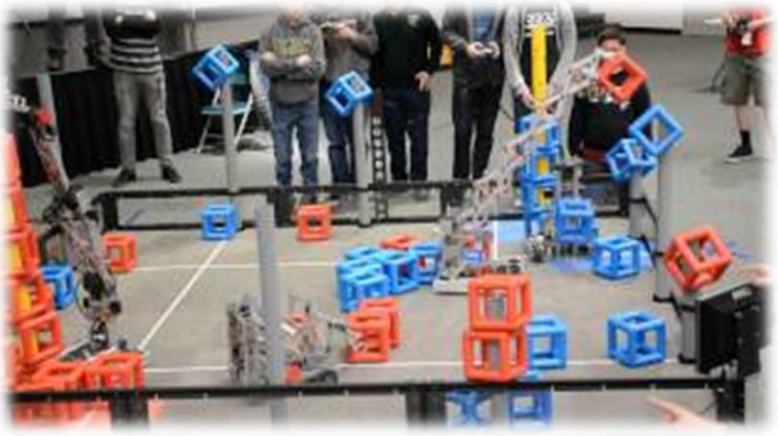
[4] Then (and usually) held at Council Bluffs, Iowa. Very close to Omaha, Nebraska, in fact. In addition to VRC and VEX IQ, also features an Open Division where many robot materials illegal in VRC are permitted. You want to build a giant crane to hoist your partner's robot out of the field for extra style points? (See 2016 finals.) Sure!

[5] Although a motor assignment is usually called "speed", I prefer not to use that term. There exists a nonlinear relation between motor power level (what you send to the motor in ROBOTC, from -127 to 127) and true motor rotational velocity. Motor power does not correspond to actual speed.

[6] PID is a form of feedback control. We will expound this later.

VEX Forum), and quite frequently. Regardless of the origin of PID at Damien, Mr. Enright and I strived to understand the workings of PID and the implementation of PID in code.

With a little tuning, PID functioned as promised. We bolted a potentiometer to each of our test machines – for me, the decommissioned 6526D robot (6526D-5603B)[7]; for Mr. Enright, the champion 6526E machine. With code, we simply, and quite literally, commanded our lifts to move to potentiometer readings corresponding to the positions that we wanted the lifts to move to – and they did. Even with the weight of two cubes, the machine of 6526E and Mr. Enright hardly complained when equipped with PID. By April we had robust PID code that could accurately control the position of a motor with user commands, and, crucially, hold up the lift of the 6526E robot against gravity.



The robot of 6526E in March 2015, attempting to descore a blue cube in VRC Skyrise play at the California High Desert League. Photo Credit: Daniel Enright

6526D-5603A as of March 2015. Note the double-reverse bar linkage lift.

Armed with PID, Mr. Enright and 6526E[8] proceeded to tie for fifth at Nationals with their alliance, an unprecedented victory for our youthful Robotics program in its second year.

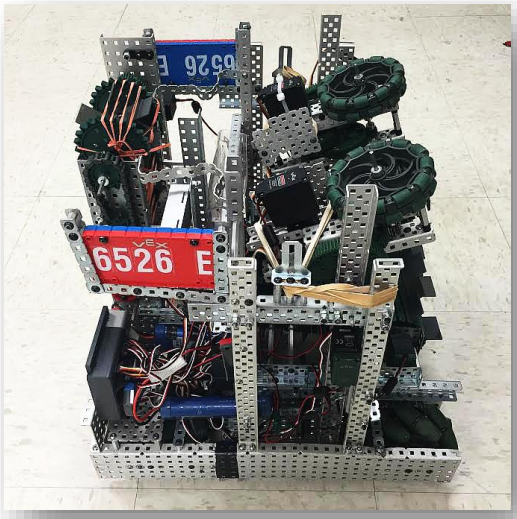
[7] My robot designation scheme is the team number followed by a dash, then four digits. The first two is the year since Damien was founded, and the latter two denote the number of the major iteration of the robot. Appended letters are used to denote minor iterations. For example, 6526D-5705 represented the zenith of my VEX career; it was built in "Damien year 57" (2015-2016) and was the fifth major iteration of that year's robot.

[8] Members of this successful team include Mr. Enright, Marco Capistrano, Davis Carrillo, and Christian Zamora

What happened after then in Damien Robotics is fairly well remembered, and documented. We expanded to five teams, with the addition of 6526H. We acquired far more aluminum, sensors, and power expanders (and blew up a few of them, in addition to a few Cortexes, and lit a Motor Controller 29 on fire). We found out exactly how unreliable IMEs were.[9] And – oh yes – two of our teams, 6526D and 6526F, qualified for the VRC World Championships for the first time in the history of Robotics.[10] Additionally, three of the teams, the abovementioned with 6526E, qualified for Nationals.

That year had been an unprecedented triumph in our time, unrivaled in school history as well. PID was responsible for much of that. However, only two teams – D and E – utilized PID; this owes perhaps to the respective programmers of the teams – rather unsurprisingly, me (D) and Mr. Enright (E).

Both of us used PID to control the angle of the flywheel shooters on our robots to ensure accuracy when launching the Nothing But Net balls.[11]



The robot of 6526E in 2016, built for VRC Nothing But Net (no name). This picture depicts the final state of the robot after Nationals. Note the integrated motor encoders (IMEs) mounted on the flywheel motors, which were utilized in various experimental velocity control algorithms.

[9] I concluded that year that IMEs are particularly susceptible to static electricity buildup on robots. Future ROBOTC firmware releases may improve IME reliability.

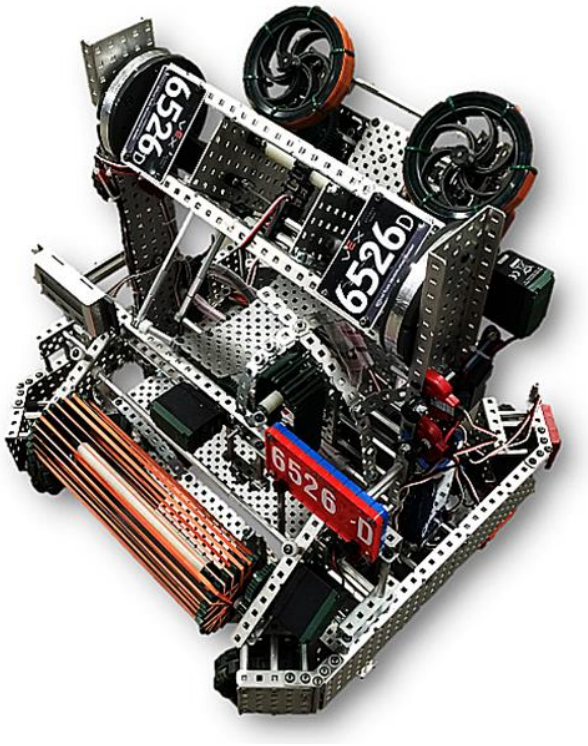
[10] It should be noted that, according to Mrs. Maricic in 2016, this event marks the first occasion where Damien High School competitive teams qualified for a world-level event. 6526D qualified through an Excellence Award, the highest award in VRC, from States (at the Northern California site – don't ask why), while 6526F qualified through additional Worlds spots awarded to California teams for Programming Skills scores. Cheers for Robotics!

The reader should also make note that Mr. Enright was a member of 6526D when they went to Worlds, and that I brought an engineering notebook to States that may or may not have been somewhat responsible for the Excellence Award. Such a notebook has been outclassed by the documentation of 6526F collected during the subsequent year. I am proud of our meticulous engineering standards.

[11] The thoughtful reader should inquire about the fate of the red Nothing But Net net with Mrs. Maricic.

We also devised PID "presets" that allowed our drivers to position our launchers for shooting from a specific position on the field with PID by a button press.[12] Furthermore, both of us pioneered the use of such PID in autonomous.

I continued further by dabbling into controlling motor velocity, experimenting with various algorithms and eventually adapting PID to control the velocity of the flywheels on the 6526D robot. This allowed me to run flywheels at a very precise rotational speed, improving the accuracy and reliability of the 6526D robot. In addition, PID also enhanced the firing rate of the flywheels, allowing 6526D to shoot balls faster. The robot that I took to Worlds eventually had potentiometer-based PID that controlled the angle of the flywheels, encoder-based PID for the velocity of the flywheels, and hybrid encoder and gyroscopic sensor-based PID that controlled chassis movement in autonomous.[13] Complicated code, sure. But the nature of PID reduces the complex logic of keeping mechanisms under control to one simple command: "go here".



6526D-5705 in 2016, a robot built for VRC Nothing But Net. This picture depicts the state of the robot that 6526D took to Nationals. A slightly upgraded version boasting improvements made by a confederation of 6526D and some members of 6526E attended Worlds. Note the quadrature encoder visible under the right flywheel rotor (two were present on the flywheel, one attached to each rotor, utilized in PID velocity control. Note also the encoder visible on the rear of the X-holonomic drive, used for PID drive control in autonomous.

[12] See "PID in User-Control"

[13] My code is freely available for the interest of the reader. Please request by contact.

PID IN ROBOTICS TODAY

We in Robotics have utilized PID to control the position of robot lifts and arms. We have adapted it to control the speed of flywheels down to a fraction of an RPM. We have tuned it to drive chassis wheels for precise distances in autonomous. Obviously, PID was a major contributing factor to the successes of Damien Robotics in its second and third years. PID has been essential in scoring Skyrise sections. It has been even more so in catapulting balls into Nothing But Net nets.

Today, almost no frontline competition programming by the Robotics varsity escapes the realm of PID. PID is one of the greatest tools in the virtual toolbox of the VEX programmer. The power of PID cannot be underestimated, but PID is not magic. PID has its limitations, and they must be understood in light of the capabilities of PID control.

Harnessing the potential of PID requires knowledge of the workings of PID. Grabbing some pre-written PID code from someone else and plugging it into your robot program, without any insight into how PID works, is sure to fail.[14] To prevent such is my rhetorical purpose in writing this text. I hope that you will find this text useful in establishing intuition about PID and learning to wield that knowledge in the form of spectacularly effective robot code that appears to be magic to the uninitiated.[15]

The history ends here. I will now begin the explanation of PID by addressing various limitations with motors that we frequently encounter in VEX Robotics, difficulties that PID can easily solve.

[14] This will become very evident later. If you choose to borrow others' code, make sure to understand it first! Otherwise you might as well be programming a black box, or a stubborn dinosaur.

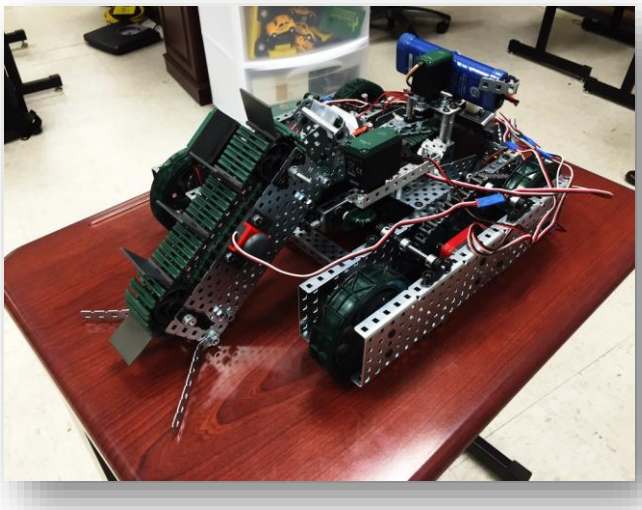
[15] Or to any non-programmer in Robotics unfamiliar with PID.

Limitations of Motors: Why PID

MOTOR MOVEMENT IS UNCERTAIN

Let us say you were attempting to write some code that autonomously moves a Clawbot forward for a distance of precisely ten meters.

This accurate movement problem is a very common objective in an autonomous routine, of which you are probably aware.



An autonomous "harvester" robot built for a project in Principles of Engineering, the second Project Lead The Way (PLTW) course in the Damien engineering course sequence. I provided principal programming for this machine (sadly without PID). Such robots were designed to autonomously "harvest" small decorative stones and sort them according to color. This is an example of a robot that could benefit greatly from accurate autonomous movement. Notice the quadrature encoders mounted on the chassis sprocket axles.

Your first instinct might be to use a simple set speed and time delay:

```
task main()
{
  motor[rightMotor] = 127;
  motor[leftMotor]  = 127;

  wait1Msec(10000);

  motor[rightMotor] = 0;
  motor[leftMotor]  = 0;
}
```

You fidget with the speed and delay time, and find values that work. Everything seems well. However, as you test further, you find that the Clawbot winds down, not able to fully traverse the ten meters. Why?[16]

To prevent this winding-down, you might introduce a distance sensor. For chassis movement, the sensor of choice is an encoder, either the quadrature encoder or the integrated motor encoder (IME). You might attach an encoder to the Clawbot chassis gear train and program the Clawbot to stop after the correct number of encoder counts has been read. This code might look like this, for a quad encoder in digital ports 1 and 2:

```
task main()
{
  // Clear encoder
  SensorValue[dgtl1] = 0;

  motor[leftMotor] = 127;
  motor[rightMotor] = 127;

  // Do nothing while the encoder value is less than our desired rotation
  while( SensorValue[dgtl1] < 4000 )
  {
    ; // Empty statement
  }

  // Once the encoder value indicates the right distance, stop
  motor[leftMotor] = 0;
  motor[rightMotor] = 0;
}
```

There is, of course, a more concise equivalent in PLTW Natural Language.

Assuming that your encoder value increases as the robot moves forward, this will work after fidgeting with the number in the while loop (which represents the number of encoder counts it takes for our robot to reach the correct distance). But even this does not guarantee that the Clawbot reaches the appropriate distance. The Clawbot, in fact, overshoots the target distance of ten meters. Why?[17]

To reduce the overshoot, you could run the chassis slowly, or command the robot to slow down once it approaches the target distance. However, this increases the complexity of the code. Is there a better way? Can we somehow directly command the Clawbot chassis motors to turn exactly the correct number of rotations that equal 10 meters of movement[18], and then stop dead there? Not likely with code that is not extravagantly complex.

These scenarios suggest to us that the following are true about controlling motors

---

[16] Battery charge

[17] "Inertia is a property of matter."

[18] 10 m = 393.7 in. In one revolution, a wheel travels 4π inches (circumference) on the ground. Therefore, the necessary number of revolutions of the chassis wheels so that the Clawbot covers 10 meters of distance is $\frac{393.7}{4\pi}$ revolutions, or about 31.3.

We may convert this to any desired units we wish – this is how we might obtain the proper number of encoder counts in our example with math instead of trial and error. But don't just trust the math – test it out too!

- With basic motor and delay commands, we cannot account for the rotational position of a motor when it moves. This is equivalent to not being able to determine the distance the Clawbot has traveled, and whether such a distance is right. This is essentially navigating your robot blindly.

    - Most Damien Robotics teams, if programming autonomous with only motor and delay commands like the above, try to ensure that the batteries used are always at the same charge. However, I find this a poor stop-gap measure and discourage reliance on such blind navigation should you need a precise autonomous program.[19] At least use a multimeter… or the ROBOTC built-in battery voltage reader.

- Even with sensors, we cannot program our robot mechanisms to move to some position with a great degree of accuracy without more complicated code[20]

We see that, even with sensors, the position of a motor is essentially indeterminate – uncertain – when it is commanded to move. We can read the rotational position of a motor axle – and thus calculate quantities such as lift height or chassis distance traveled - with sensors, but we cannot control the movement, the change in position, of a motor accurately.

[19] I respect the skill and tenacity of the programmers of the 2015-2016 6526F, among who are Eren Bikmaz, Federico Parres, and Lorenzo Scaturchio, whose code in Programming Skills, after their robot had scored one set of match-loads, moved their robot across the Nothing But Net field and positioned it to face the other net precisely – with nothing but dead reckoning. They should have used some PID though, which would have vastly reduced their debugging time. The subsequent PID systems of 6526F have all proven to be robust and reliable in competition.

[20] Exceptions to this include VEX IQ (Oh come on, motors with fully integrated feedback modules?) and the ROBOTC built-in PID; see "The Built-In PID in ROBOTC".

MOTOR VELOCITY IS UNCERTAIN

Many elementary VEX programmers assume that the motor set commands

```
// PLTW Natural Language
startMotor(myMotor, 127);

// or
motor[myMotor] = 127;
```

are instructions that set the actual *speed* of a motor. They assume that 127 is some measure of speed. [21] However, this is not the case. It is more appropriate to consider the effect of such commands as setting a level of motor *power*, where 127 and -127 correspond to full forward and reverse, and zero indicates no power.

This is because the motor set commands do not actually correspond to motor speed, but rather motor voltage.

---

[21] Why is "127" the maximum value that can be sent to a motor, anyway?

Well… 127 is the largest integer that can be written with eight bits of computer memory. As you might know, eight bits is an organization of memory called a *byte*. Bits are a means of storing information (thus, memory), like our alphabet, but instead of our twenty-six characters, punctuations, and numbers, bits can only be off or on, zero or one – like a light switch. Computer memory is very much like millions of miniscule light switches, where "off" and "on" of particular light switches mean something, such as the color of the pixels on your computer screen., or the frequency of the sound spilling out from your speakers.

Each bit has a total of two states. So the maximum number of states (in this case, different integers) that you could represent with one bit is two (0 and 1), two bits is four (0, 1, 2, 3), three bits is eight, and so on… $2^{\#\ of\ bits}$. For the number 127 stored as an eight-bit (one byte) integer in memory, one bit is used to specify the sign of the number (zero for positive, one for negative), and the other seven… well, 127 is one less than 128, which is the number of integers that you could represent with *seven* bits — $2^7 = 128$. It turns out that 01111111 – zero (a zero on the first bit indicates a positive number) followed by seven ones – the largest positive integer that you can store in eight bits – is 127.

Negative numbers in binary are somewhat different – the maximum negative motor value is -127, while the most negative integer that can be written with a byte is -128.

Notice that $2^8 = 256$ – the maximum number of different integers that 8 bits can represent – and that the range [-128, 127] contains 256 integers. I assume that ROBOTC sets the negative value limit for motors at -127 for symmetry. This also explains why the VEX remote joystick values are [-127, 127].

There is actually a numeric data type in ROBOTC (and most other programming languages) known as the `byte`, with precisely the bounds of an integer that can be represented in a byte. Note that the `int` data type in most computer systems is either two or four bytes in size – with the Cortex, it is four – thus the maximum value that `int` can store is much larger (for four bytes, the range is approximately -2,000,000,000 to 2,000,000,000).
<http://help.robotc.net/WebHelpVEX/index.htm#Resources/topics/General_C_Programming/Data_Types.htm>

HOW MOTOR CONTROLLERS WORK

You may have wondered why Motor Controller 29 modules are necessary on the majority of Cortex motor ports, or why the actual Cortex ports have three wires, whereas the motors have two.

Motors themselves are fairly simple devices. At their heart is an electromagnet – a magnet that can be switched on and off by electricity. The portion of the motor that actually spins is connected to a magnet on the axle, which is encased in a fixed electromagnet in the housing of the motor. If the electromagnet is powered with alternating current – electricity whose flow direction reverses at regular intervals – its north and south poles of the electromagnet will "flip" repeatedly. The magnet on the shaft of the motor will try to follow the electromagnet. Since the north pole of the shaft magnet always wants to go to the south pole of the electromagnet, and the south pole to the north, the shaft magnet will spin to try to follow the switching poles of the electromagnet – and the motor will turn. However, we feed (most) motors, including the VEX motors, with direct current. The motor contains a mechanism that causes this current to switch as the motor turns, creating the necessary alternating current. Such motors are known as **brushed DC motors**. A vast majority of motors – including ours – are brushed DC motors. The VEX motors require about 6 volts to run at full power.

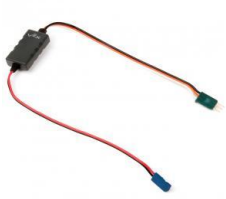
So we know that a motor requires direct-current voltage to run. This implies that motor control requires changing the amount of voltage that the motor receives in order to change the speed. This makes sense – no voltage, no power and no spinning. Reverse the polarity of whatever powers a motor, and the motor turns the opposite way. This is not a problem for simple circuits like those in small toys.

However, in more complex systems (such as VEX, where the Cortex must regulate battery power and control motors all while its own internal circuitry runs your code and does housekeeping) we would often like to keep our electronics at the same standard voltage. This is desirable when circuits are extremely complicated (such as those in the Cortex, and in your computer), consisting of millions of transistors. Having a uniform voltage standard for circuit electronics simplifies circuits considerably.

This concept is why digital electronics are called **digital**, in contrast to **analog** (where the voltage can vary). Today, there are two customary voltage standards, 3.3V, and 5V. The Cortex digital sensors use 3.3V as a standard. Behind the scenes, the microcontroller (a small computer processor) and other circuitry components inside the Cortex probably run at 3.3V too. Most sensitive electronics use 3.3V, whereas 5V is found in sturdier devices (such as the Arduino Uno, a popular microcontroller that is essentially a much cheaper Cortex for $20) and the large IC chips used in the Digital Electronics PLTW course.

However, we know that motors are inherently analog – we must vary their input voltage in order for us to run them at different speeds. How do we go from the digital side of things – the programmable circuitry and transistors inside of the Cortex microchips – to the analog voltage that we need to power the motor? How do we vary motor speed with digital circuitry?

This is why we need a motor controller. The motor controller translates digital instructions from the Cortex to analog voltages to drive the motor.



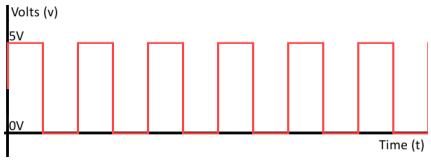
If we examine a Motor Controller 29, we see that the input side - the side that comes from the Cortex – has the three-wire pin. The other side, the pin that connects to the motor, has only two wires – red and black. It is quite obvious what they are – positive and negative.

Fun fact: if you need to reverse a motor and don't quite feel like changing your code, you can physically reverse the connection to the motor here.

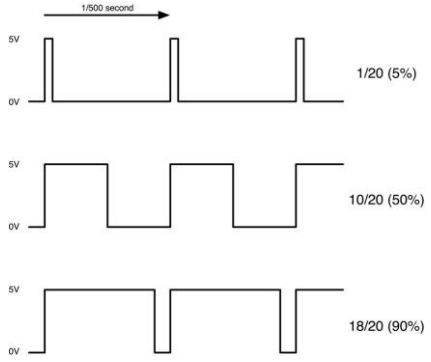
However, what is on the digital side, the input from the Cortex? We have no way of knowing exactly which wire is which, unless we measure with electronics testing equipment, or look online: <http://www.vexrobotics.com/276-2193.html>

According to VEX, the black and orange wires correspond to ground and power – this is another way of stating negative and positive. "Ground" is analogous to "the negative terminal of a battery". This is expected – we can't have power output if no power is put in in the first place. However, where the digital-to-analog translation occurs is in the white wire.

If we had the test equipment to examine the current in the white wire as the motor connected to the motor controller runs, we would see something like this graph, where the vertical axis is voltage and the horizontal axis is time.



In electronics, this is known as a **digital signal** or a **digital wave**. Because the peaks are square, these are also termed **square waves** (but not all are digital). The Cortex passes a digital signal like this – by sending electricity in an on-off pattern – to the motor controller, which then translates that signal into an analog voltage to control the motor power. If the shape of this signal in the white wire is varied in this fashion



the motor controller outputs different voltages. We see that the changing quantity is the percentage of time that the signal is "on". This quantity is known as **duty cycle**. The motor controller translates the duty cycle of the white wire input into a particular analog voltage for the motor.

This method of controlling the motor – by varying the duty cycle of an input signal to control an analog voltage – is known as **pulse-width modulation** (**PWM**), which is a fancy way of saying "change the width of the digital wave"). The actual digital signal is called a PWM signal. PWM is extremely common in electronics that drive analog devices: in fact, servos are controlled by a PWM signal too (hint: three wires!)

Now, what about the motors in motor ports 1 and 10? They don't need motor controllers! In fact, the motor controller circuitry for these motors are built into the Cortex already. The circuitry often used in motor controllers – the same circuitry that drives motor ports 1 and 10 – is called an **H-bridge**. This is suggested in ROBOTC: the Motors and Sensors Setup code for a motor on one of these ports looks like this:

```
#pragma config(Motor,  port1,           myMotor,       tmotorVex393_HBridge, openLoop)
```

[22]

The code in the motor set commands serves to set the duty cycle of the PWM signal going to the motor controller. This means that the motor commands affect motor voltage – not motor speed. Motor voltage determines the **torque** – roughly speaking, rotational force[23] – that the motor provides – but not the speed.

This is significant because the motor set commands do not guarantee the actual speed of a motor. They do not even guarantee that the same value, assigned to two motors, will cause both motors to run at the exact same speed![24]

True motor speed depends on a myriad of factors in addition to voltage, among which are

- Load (anything that the motor is driving, and the forces that such imposes on the motor)
- Friction
- Air resistance
- Internal inconsistencies in motors, small deviations in dimensions, etc.
- Battery charge
- Electron tunneling? Heisenberg Uncertainty Principle?

There is also a certain power threshold in which a motor will not move. The minimum power (i.e. motor set command value) required for a motor to overcome friction and actually move is usually between 11 and 15, depending on the attached load.

We conclude that one of the limitations of motors is that they cannot be driven at a constant true speed with ease, with the simple ROBOTC that we know.

---

[22] A note on programming: I use the name `myMotor` here as an example, but in actual programming, you should NEVER name motors, sensors, or variables with non-obvious names: "*motor*", "*myMotor*", "*x*", etc. The justification for why is clear.

If you happen to be interested in electronics, take Digital Electronics at Damien. It teaches practical logic-based circuit design involving projects like a LCD display for your birthday, a toll booth, and a voting machine. No, you won't be building computer chips – but you will get to do some pretty fun stuff, including learning Arduino (an invaluable piece of kit for DIY folks) and controlling VEX motors without a Cortex. Digital Electronics made me into an electrical engineer. You do not choose the EE life; the EE life chooses you.

[23] Any high school physics course will provide a rigorous definition of torque.

[24] Ask anyone in Robotics who was a programmer during the Nothing But Net season.

MOTORS AREN'T SERVOS

Another limitation of motors is that they cannot be commanded to rotate to a particular position – and stay there – like servos. This behavior is not useful for some applications (e.g. chassis, flywheel, etc.). But this is very much desirable for many, especially for systems like lifts (where being able to control a lift by position would be incredibly useful in autonomous – we could say "Go to the height of the tall goal." and not "Turn on at full power for some number of seconds, and then stop."). If this were possible, we could solve our Clawbot movement dilemma trivially – with commands to "go here".

From a different perspective, this is simply a desire for stronger servos[25] without a rotation limit.[26]

MOTOR POWER AND BATTERY LEVEL

One of the factors behind some of the aforementioned motor limitations is that battery charge undoubtedly affects motor power. This affects the true speed of a motor, which in turn affects how far the motor will move when run at a certain speed for a certain amount of time – all limitations which we have previously discussed.

[25] Servos, unlike motors (unless you recall the ancient VEX Motor 269), have plastic internal gearing, and can tolerate much less load than motors can (VEX claims 6.5 inch-pounds of stall torque – a measure of the "rotational force" necessary to stall out the motor – versus 14.76 for the 393 motor). At the time that this text is read, there may be new, improved motors and servos with greater load tolerance.

[26] This exists outside of VEX; such servos are known as continuous rotation servos. However, they are not truly position-controllable; they are controlled much like motors (They use PWM, but the PWM sets a "speed" rather than a position). They are also very weak compared to motors.

PROBLEMS

1. (**Introduction to PID**) Consider the problem of programming a robot to traverse ten meters presented at the beginning of the section. Write a program that accomplishes this task to a better extent than the bad solutions that we have considered. Justify this by either programming an actual robot, or by explanation based on principles of robot mechanisms. You may use PID if you know it. *You do not need to use sensors.*

2. (**Introduction to PID**) Consider the problem of programming a robot to traverse ten meters presented at the beginning of the section. Explain how sensors can permit this robot to make decisions about how much motor power to apply depending on how far away from the target distance it is.

3. (*Algebra II*) Using a quadrature encoder, write a program that reads the speed of a motor (Hint: speed is the slope of a graph of motor position over time. Why?). Using this program, plot a graph of motor power (ROBOTC command value) versus true speed. Obtain at least ten data points and use an application like Excel, Mathematica, MATLAB, etc. to make your plot. (You do not need to plot negative power.). Suggest, or generate, an equation that closely fits the data.

4. (*Calculus*) The VEX accelerometer outputs three acceleration values: one for the x-axis, the y-axis, and the z-axis. Suppose that a robot is moving in the direction of the positive x-axis.

    a. Derive an equation for the position of the robot $x(t)$ as a function of x-direction acceleration $a(t)$ as it changes with time, and time t. (Hint: Acceleration is the second derivative of position)

    b. Play with an actual accelerometer. Will the equation from (a) be an appropriate means to determine the robot's position to fairly good accuracy (less than say, a foot of error) from the accelerometer readings? Explain why or why not. You do not need to build an actual robot.

    c. What characteristics of the accelerometer, or of a robot in general, play into the result of (b)? Is (b) problematic for accurate movement during autonomous? How might you improve the accuracy of an autonomous routine that relies on an accelerometer, if so?

5. (*Digital Electronics*) ***With permission***, use a multimeter to examine the voltage and current across a motor running at some set motor power for a full and nearly-dead battery. You may need to use a breadboard to provide external connections to the motor-Cortex circuit. **For the current, be sure to connect the multimeter in series and to use the large-amperage terminal on the meter** (otherwise you will ruin the meter). If you are not confident in your ability to safely measure current, skip this step. Are the results what you would expect? Does the motor display ohmic behavior? Should it?

6. (*Digital Electronics*) Research PWM. Why is PWM important? Are there any devices besides the motor and the servo in VEX that might be controlled by PWM? List these and justify.

7. (*Mechatronics*) Research current motor and servo technology. Is the VEX motor good for what it does, compared to similarly-priced motors on the market? Are there technologies accessible to us that might eliminate some of the issues with motors discussed in this section? If so, analyze why these technologies solve these issues.

What is PID and What Does It Do?

PID is a way to fix all that, a method to empower a motor to surpass all (most) of those limitations. In the simplest terms, PID is code that lets us control a motor based on any measurable quantity we want. PID for motors in Robotics is usually used to control

- Position: PID allows a motor to behave like a servo, rotating to a precise angle (and staying there).
- Velocity: PID also allows us to control the speed of a motor very precisely.

We could even have a motor controlled by how hot it is in the room if we wanted to! (we would just need a temperature sensor, and we must ensure that the room temperature responds to our motor as well, through something like an AC unit with heater.)[27].

Most of the PID that we have used in Robotics deal with position (e.g. how to move a robot arm[28] to some angle, and keep that arm there). Therefore, I will introduce PID through its typical application with position.

Fundamentally, PID is no different from the motor set commands.

```
// PLTW Natural Language
startMotor(myMotor, 127);

// or
motor[myMotor] = 127;
```

PID requires use of these commands. The difference is how the power value is determined. Rather than being some value you decide (i.e. `motor[] = 127`), the value is calculated by code. Elegant, simple code, yet based on fairly complicated math.[29] But the results of PID are impressive. With PID control, instead of fooling around with time-based motor code or sloppy sensor code, we can literally tell your motor "go to this position". It will go there, stay there, and even resist attempts to push it away from there automatically. PID will laugh off differences in battery level. With either a full battery or a nearly dead one, PID will get your motor to the same place.[30]

[27] This is how air conditioning works fundamentally, but I doubt that most AC systems use PID.

[28] By "robot arm", I mean a simple rotating body, like the VEX Clawbot arm, with one degree of rotational freedom. If you desire an overview of robotic arms with n-degrees of freedom, I recommend reading < http://www.cds.caltech.edu/~murray/books/MLS/pdf/mls94-complete.pdf/>. College mathematics background is recommended.

[29] For anyone without knowledge of elementary calculus. If you have taken or are taking an equivalent of Calculus I or AP Calculus AB (single variable calculus), you will comprehend just fine.

[30] Of course, battery level will still affect motor torque (or, loosely speaking, the "power" available to a motor to drive mechanisms). Therefore, PID will not offer identical performance with a nearly dead battery versus a full one. PID will just help to compensate for the power difference.

What really is PID? It is essentially an algorithm – a set of instructions[31] – that tell a **system** (i.e. a robot mechanism) how to respond to input commands and **disturbances** that might knock the system off course.

The best analogy for what PID is like and does is how we balance on two legs. Before reading about the intricacies of PID, think about how we keep our balance. What algorithm – what sort of rules – do we follow to keep upright? What does our balance intuition depend on?

- How upright we are at any moment
- Whether the ground is tilted and we need to adjust our "equilibrium" balance angle
- Whether we are falling or losing balance, even if we are upright (e.g. keeping balanced after being pushed forward by someone)

As we will see later, these three aspects of our balance behavior correspond exactly to three components of PID: P, I, and D.

REASONS FOR PID

Motor Control

PID enables precise position and velocity control of motors. With PID, we can command motors to move to any particular sensor value from a potentiometer or encoder connected to the motor axle. We could even read motor velocity from an encoder and command a motor to spin at any particular speed, in RPM.

Accuracy

PID can keep a robot system near some desired position or speed easily.

Robustness

PID allows robot mechanisms to resist changing conditions. In our autonomously moving robot example, battery power affected how far a time-delay-programmed robot would move. If we applied PID to drive a robot chassis forward for some set distance, even with changing robot mass, PID will still ensure that the robot will get fairly close to that location. PID allows these robotic systems to still function as desired even as circumstances change.

In the PID literature, this sort of reliability is called **robustness**.

Besides this, PID will work as intended even with a not-so-perfect algorithm. As we will see later, there is no such thing as "the" PID algorithm. PID code must be **tuned** – constants in the code must be adjusted – for optimal performance. Even if the constants are not perfectly adjusted (indeed, there is no such thing as a perfect PID), PID will still work.

[31] Technically, anything you program onto your robot is an algorithm. However, PID has a greater resemblance to the things usually associated with the word "algorithm". It's a programmed set of instructions that lead to a solution.

Control Theory

PID is known in technical circles as a form of **closed-loop negative feedback control**.

Closed-loop control refers to controlling something – in our case, a motor – with information from sensors. For example, suppose that your AC cools the air continually until the temperature in your room is 70 degrees Fahrenheit, then cools the air as necessary to maintain that temperature. Contrast that to simply running the AC for a set amount of time (this is known as **open-loop** control).[32]

The obvious VEX analogy is running a robot autonomously for a set distance based on encoder readings (closed loop) versus "dead-reckoning", telling your robot to move for a set amount of time (open loop).

Feedback refers to the sensor reading aspect of PID – control is based on feedback from sensors. But why negative feedback? **Negative feedback** refers to feedback where the goal is to "remove" the influence that "caused" the feedback. Consider the autopilot on an airplane – its job is to keep the plane at a constant altitude and heading (direction) despite external influences (wind, and so on). If the plane deviates from its desired altitude and heading, the autopilot corrects for this deviation, returning the plane to the desired altitude and heading. This is negative feedback. Examples of negative feedback are found in many engineering and scientific fields, and in everyday life too. How do you adjust the temperature of the water in the shower?

This contrasts with **positive feedback**, where the cause of the feedback is amplified. A familiar example is audio feedback (a little bit of feedback triggers more noise – very unpleasant noise – which in turn triggers even more noise). Positive feedback mechanisms are very common in biology: for example, in the human body, the release of small amounts of certain hormones and neurotransmitters causes the release of a lot more. PID is not a form of positive feedback.

The technical name for "a PID", "one implementation of PID" is a **PID controller.** The term *controller* here has nothing to do with remotes or joysticks. It is a reference to the feedback aspect of PID. A **feedback controller** tells its system how to respond to feedback. A PID controller quite literally controls the system at hand. Sure, there might be manual inputs, like joysticks or buttons, but the PID controller is ultimately responsible for directing our system, assigning the final output to the motor(s). A PID controller, strictly speaking for Robotics, is the code that controls a motor with PID, together with the sensors, motors, and hardware involved. We might as well consider "controller" as analogous to "algorithm" in our case; in Robotics we usually consider the PID purely in terms of the software that makes up the PID. However, in t PID literature, a PID controller includes the physical components of the system too. For us, that means motors, motor controllers, the Cortex, and so on.

Most of the Robotics programmers tend to refer to "one PID" as a **PID loop**, since PID usually involves a while loop at its heart.

---

[32] The dead-reckoning (motor on for a set amount of time) approach to movement that we first discussed in the *Limitations of Motors* section is an example of open-loop control.

PID is part of a field known as **control theory**, which deals with the logic of controlling systems (anything from a robot arm, in our case, to airplanes to AC units to nuclear power plants) that need control. An example of what control theory can deal with is how an airplane autopilot will manipulate control surfaces on a plane's wing and tail to keep the plane flying straight and level.[33]

The Grumman X-29, an experimental forward-swept-wing aircraft used in NASA research. The aircraft achieved a high degree of maneuverability from its inherent instability. Feedback control was used to keep the plane flying properly.
Photo Credit: NASA

Control theory is vital in fields like aerospace engineering (so how do you think those guys at SpaceX make their rockets fly perfectly autonomously? What about Boeing and the autopilots in their jetliners?), industrial engineering (that machine needs to attach the screws at the right place at the right time, always…), and, of course, robotics. Originally, PID was developed for ship steering – and it's most likely still used for such today.

It's reasonable to claim that the modern world depends on PID and feedback control as much as it does computers and the Internet (which also depend on feedback control, to some extent).

[33] There is actually a course involving that and more at MIT. There are several courses, undergraduate and graduate, on that on MIT OpenCourseWare – free academic resources (but probably much too advanced for most)! At Caltech (where I'm at now), there is a control theory minor, Control and Dynamical Systems (which I intend to take). You will probably have to take controls classes if you do engineering, particularly industrial or mechanical.

But seriously, how much does PID interest you, beyond programming your robot to get to Worlds? Probably not much, I suppose.

The Saturn V launch vehicle that ferried the Apollo 11 spacecraft to lunar transfer orbit on rollout from the Vehicle Assembly Building at Kennedy Space Center, May 20, 1969. How might feedback control principles have been used to guide the massive rocket on its way to the Moon? Later, think about how PID might have been utilized to control its ascent. Photo Credit: NASA[34]

Most PID literature addresses its most widely known application: industrial engineering. It is a frequently touted statistic that some 95% of all industrial feedback controller processes utilize PID in one form or another. In the industrial engineering world, the impetus behind PID is the need to control a **plant** (perhaps a nuclear reactor, or a furnace, or a heat exchanger…) in order to keep operation safe, efficient, and profitable.

[34] For a non-technical treatment of the topic of launch vehicle ascent control, among much more, please see SAS control in *Kerbal Space Program*. Fun fact: The Space Shuttle did in fact use PID on ascent.

PID Fundamentals

ERROR

Before we can understand how PID operates, we need to become familiar with a concept called **error**.

The fundamental principle of PID:

ELIMINATE ERROR

Error is deviation from the desired. More deviation, more disturbance, equals larger error.

Suppose you want your room at 68° Fahrenheit. It's 88°. Not only do you have error, you also have a relatively large error – one that is presumably making you sweaty if it's summer.

Mathematically, error is usually defined as the **difference between the desired value of some quantity and the actual value of that quantity.**

$$error = desired\ value - actual\ value$$

Some examples:

- The airplane autopilot example: The autopilot error would be the difference between the desired altitude and heading (what the pilot enters into the flight computer) and the actual altitude and heading of the plane.
- The robot arm that we started off with: The error would be the difference between the desired arm position (angle) and the current position.
- The hot room: The error is the difference between 68° and 88°, or -20° F. Note that error can be negative.

Intuition with error suggests larger error means more deviance from the desired, or "target" quantity. However, note that with the math, error can be negative (if the actual value is greater than the desired value), and what the sign of error means depends on how you set your system up. We will see more of this later, but for now, let's consider an example:

You have a robot arm with an attached potentiometer whose reading increases as you go up, raising the arm. You want the arm to be at sensor value 3000; however, it is currently at 1500. The error, therefore, is 3000 – 1500 = 1500.

What does the positive error imply, in this case? The current position of your arm is below where you want to go. What if the error were negative instead?

With PID, the "desired quantity" or "desired value" is usually termed the **target value** or **setpoint** (in Robotics, the former is preferred). The target value is what your PID attempts to follow. We see this since we want error to be zero; therefore, the "actual value" has to equal whatever the target is.

The "actual value" is usually called the **process value** or **process variable**. In Robotics, we prefer to use "**sensor value**" since the actual value is literally a sensor reading in most cases.[35]

In the PID literature and its industrial applications, the setpoint and process variable are usually abbreviated SP and PV.

In any PID software, the error must be repeatedly calculated whenever PID runs.

[35] Sometimes, we will need to calculate a process variable. For example, the quadrature encoder does not directly output (rotational) velocity data. If we wished to program a PID loop to control motor velocity, we would need to calculate the velocity of the motor from the position data that the encoder gives us.

THE PID SUM

With some error, a PID controller will calculate a motor **response** value that, when assigned to the motor(s) of our PID system repeatedly, will eliminate the error over time:

$$response = P + I + D$$

P, I, and D are quantities that depend on the `error` and how it changes over time. I like to refer to this equation as the **PID sum**. The response is, quite literally, P plus I plus D. I also term this `response` the **output**. Some authorities on PID call `response` a **control output**. We simply assign this `response` to a motor much as we would with the number 127.

If we keep repeating this calculation in code and setting our motor(s) to that `response` value, our `error` will disappear to zero, our arm will hold itself up, our robot will magically travel the same distance in autonomous all the time, etc.

Some key points about the nature of implementing PID follow:

- **PID must repeat**. In order to eliminate `error`, the calculation has to be run over and over again as time goes on, and the `response` sent to the motor that controls our mechanism. This also means that `error` has to be calculated repeatedly too.

  In programming, this means that we probably need a while loop to house our PID calculations and to cause them to repeat.

- **P + I + D is a number that becomes the motor power**. The value that PID calculates is sent to the motor(s) in our system. This causes the system to move, which changes the `error`, which affects the PID `response` sent, and so on.

- **The error must be re-calculated for every PID repetition**. This must be done since error will change as PID repeats. As we will see later, the error calculation is the first action done on every repetition of the PID loop.

- **Ideally, PID should give us no error as fast as possible.** No error is not simply reaching the target position at maximum speed – then the mechanism will overshoot. The mechanism must settle down around the target.

PID Theory

PID is really that simple. Just P + I + D. But what **ARE** P, I, and D?

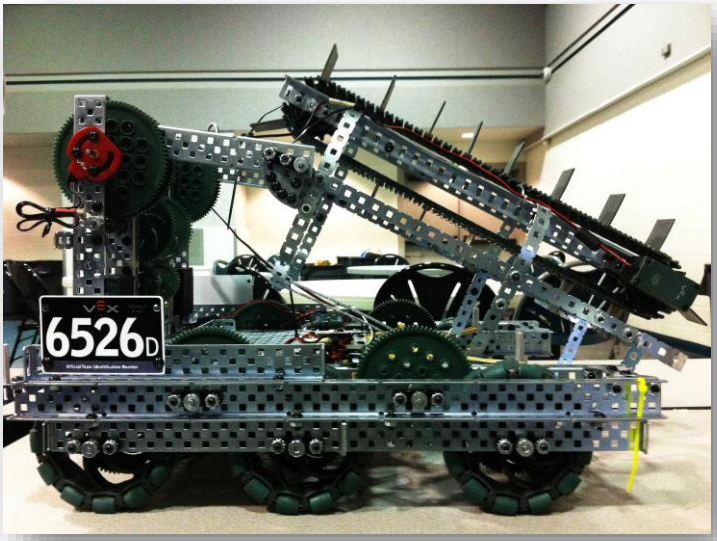
P, **I**, and **D** are three different terms – math expressions – that depend on the value of the error and how it changes over time.

They are what PID is named for: the **proportional term**, the **integral term**, and the **derivative term**.

To illustrate the different aspects of PID, we will consider one example from now on:

Suppose we are attempting to rotate a robot arm upwards for exactly 90 degrees. We have a sensor mounted on the arm capable of reading its angular position.

This type of task – moving a mechanism to a very specific position – is very common in autonomous, where an arm or a lift may require precise rotation in order to score objects in goals of a specific height or accomplish some other objective, like robot hanging in VRC Toss Up and VRC Nothing But Net.



6526D-5504 in 2014, built for VRC Toss Up. This robot is one of the first Damien Robotics machines to compete at the Hesperia league championships and took tournament finalists in the first year of Robotics. The large arm was attached to a conveyor system which could carry three scoring object balls. The arm would then be raised and its contents scored. This is an example of a robot that could benefit from a solution to our scenario (i.e. an autonomous routine with accurate arm movement behavior). Note that the potentiometer on the arm was not used.[36]

Now we can immediately begin defining the PID quantities for our robot arm system:

- The **target value** is 90 degrees, or whatever 90 degrees is in terms of the sensor being used. Since 90 degrees is relative to our starting position, in practice, we would have to find the target value from our starting sensor value.
- The **process value** or **variable** is the sensor value, a measure of the current angle (position) of the arm at any time.
- The `error` is the target value minus the sensor value. As the PID runs when time goes on, it should decrease to zero.

[36] No one in Robotics, except for the coaches, knew how to code back then. Like not just not ROBOTC, but no computer programming at all. Not even the graphical "puzzle-block" languages. Nothing at all. Such humble beginnings we have…

My advice to the reader is to avoid graphical languages like the bubonic plague, for your own sake. Learn proper programming! I found that learning programming from robotics (and then going C » C++ » Java » Python) was more beneficial for me as a programmer than beginning from, say, Python. C and C++ (and ROBOTC) teaches you to be diligent and careful. Otherwise, segmentation fault to you.

Before we delve into PID, however, we must first determine some sense of direction and sign for error based on the sensor that we are using:

- If, according to the sensor, a decreasing reading corresponds to the arm going down – and vice versa, the `error` will be positive when the arm is below the target value (and vice versa).
- If a decreasing reading indicates a rising arm, the `error` will be negative when the arm is below the target value.

Determining this when implementing PID is crucial.

KNOWING THE SIGNAGE OF THE ERROR IS CRITICAL.

We have referenced this earlier when defining the concept of `error`. The sign of the `response` is the same as that of the `error` (this will be more evident later on when we have established the math of PID). You don't want your motor going the wrong way!

If it does go the wrong way, the `error` will increase (not decrease!) in size when the motor moves according to the `response`, and the `error` will become infinitely big very quickly. You have positive feedback, and your arm will either shoot for the moon or slam itself in the ground.

If this occurs, we may multiply the PID sum by -1, or the error calculation by -1, or reverse the motor, to fix the signage. That way, the presence of `error` will cause a `response` that acts to make the `error` closer to zero. For example,

Suppose that the sensor value of the potentiometer that you are using for PID on your robot arm decreases as the arm goes up. So the error is negative when the arm is below the target value position and positive when the arm is above the target. Now suppose that a positive motor power causes the arm to go up. We have a problem – if the arm is above the target, the positive motor power drives the arm farther up, away from the target (and vice versa if the arm is below the target). To fix this, we might

- Multiply the error calculation by -1
- Multiply the PID response value by -1, before we send it to the motor
- Reverse the motor in the ROBOTC Motor and Sensor Setup

Performing any one of these actions will be sufficient to fix this. You could do all three (or any odd number of such sign reversals), but why?

We will assume, in our case, our arm moving up corresponds to an increase in our sensor value. So the error will be positive when the arm is below the target. This is a useful convention to stick to, in order to reduce the chance that you will misinterpret your code later on and cause disaster.

PROBLEMS

1. (**Introduction to PID**) Explain why PID is a form of closed-loop negative feedback control. Be specific. Give an argument on why it is necessary that PID is negative-feedback.

2. (**Introduction to PID**) Suppose that you are implementing PID control, but accidentally load a positive feedback algorithm on your system. What would happen? You may explain with an example or in general terms. Perhaps the easiest example to use would be moving a robot arm to a target position, the example that we presented earlier.

3. (**Introduction to PID**) Argue that the whole concept of eliminating error that lies behind PID is especially suited to robotics. (This is open-ended).

4. (*Economics*) The renowned macroeconomist John Maynard Keynes argued that government intervention is necessary to ensure a stable, thriving economy. Compare the major principles behind Keynesian economics (e.g. the behavior of aggregate supply and demand, with or without regulatory action) with principles of feedback control. What sort of feedback (positive or negative) is embodied in Keynesian economics?

5. (*Feedback Control*) Devise a feedback controller (in written English) for a household central heating unit to regulate the temperature in a room. Try to be as realistic as possible – for example, account for the fact that there are lags between switching the heater on and off and how the temperature of the room responds.

6. (*Feedback Control*) Propose a robotics application where positive feedback may be useful.

The Proportional Term

When considering how we might eliminate `error`, the ultimate goal of PID, it is apparent that the size of the PID motor `response` should first and foremost depend on the size of the `error`. If we have a larger error, we should have a larger response to deal with that error.

Say, in our example, our sensor says that we are at degree 0, and our target sensor value is degree 90. Then your `error` is 90. Now suppose we were at degree 89. Obviously, we don't need as much `response` to get to degree 90 from 89 than from 0!

This is the essence of the **proportional term**, the P component of P + I + D. Its value is directly proportional to the `error` – hence its name. We define the `P term` as

$$P = k_p \times error$$

where k_p is just some arbitrary constant value that scales the value of `P`.

You will probably recognize this as the **formula for direct variation**, $y = kx$. That is exactly what this is. The `response` directly varies with the `error` – or the size of the `response` is directly proportional to the size of the `error`.

Remember that P is simply added into the PID sum. When the `error` is large, the `P term` is large, and the `response` (remember, the sum of P + I + D – let's ignore the other two for now) is also large. When the `error` is small, the `P term` is negligible, and thus the `response` is small.

We will need to determine a value for k_p that is right for your system after we have constructed our PID.For example, an error of 4095 (the maximum value on a potentiometer) is simply an error of 270 degrees (in the real world). Let's ignore the I and D terms of PID for now. If I and D were not included in PID, we get this result

$$response = P = k_p \times error$$

Well, if there was no k_p there, the `response` that gets sent to the motor would be 4095 – far greater than the 127 maximum for a motor!

k_p should scale that error variable so that the `response` is more sensible in size. So k_p as a scaling constant is needed – similarly, constants are necessary for the I and D terms later on.

The value of k_p must be determined carefully:

- If k_p is too small, the PID response will be unreasonably small with respect to the error, and the system will respond too slowly.
- If k_p is too large, the PID will overshoot the target (which is usually acceptable, depending on the particular PID system and application) and oscillate – "bounce" – around the target while settling down. In extreme cases, the PID will oscillate forever. The system will never reach a **steady state** – that is, not moving (in our usual case of positional PID). Such a system is usually called "unstable", although the precise definition of stability is vague.

In various PID literature, k_p is usually termed the **proportional gain**[37] or the **proportional gain constant**. The term **gain constant** refers to similar arbitrary constants for each of the three PID terms.

You may also opt to scale the `response` with a constant as well. Some prefer to calculate their PID in terms of decimal percentage (where 1 and -1 would be the maximum allowed response) and then scale that up to motor range. Yet no one in Robotics prefers this method.

The proportional term contributes the most to the PID `response`; even in a full PID loop (P + I + D), the size of the proportional term will be the main determinant of the size of the `response`.

You can even forget the I and D part of PID, keeping P only, and you will still have a feedback controller that will get rid of error for you. This is called a **P controller**. It works and is much simpler than a PID controller, but it won't get you as much mileage (as we will see). It will be less precise than full PID. Additionally, with VEX motors, a little error will always be left over.[38]

[37] This use of the word "gain" might be unfamiliar to you if you have not taken electrical engineering courses. It owes its origins to electronic amplifier design, where "gain" is a factor that scales the amplification.

[38] Why? We will soon see…

PROBLEMS

1. (**Introduction to PID**) Build a 1 degree of freedom (DOF) robot arm with a 84 : 12 gear ratio, a motor on the 12 tooth gear, and a potentiometer on the arm joint (connected to the 84-tooth gear). Write a P controller for it. Set the target value to a constant[39] and let the arm move to the target when you turn on the arm. If you do not know exactly how to write the PID code, read the section on programming. Or try to figure it out yourself!

    a. Tune the PID controller by changing the value of k_p so that it is as large as possible, but the arm does not bounce around the target after the system is run.

    b. Analyze how the response changes if you add weight to the end of the arm. Does the response slow down? Does the arm not reach the target exactly?

2. (**Introduction to PID**) Build a robot chassis with a quadrature encoder on one of the wheel axles. Your machine does not need turning ability. Write a P controller that will move the robot forward for three meters, autonomously.[40] Play around with k_p and see how accurate you can get. If you do not know exactly how to write the PID code, read the section on programming.

3. (**Introduction to PID**) Suggest a robotics application where a P controller would provide poor performance. Suggest an application outside VEX Robotics as well.

4. (*Calculus*) Suppose that the position $\theta(t)$ that a PID-controlled motor covers in some time t while error is positive is directly proportional to the accumulated PID response over time $u(t)$:

$$\theta(t) = C \int_0^t u(t)\, dt$$

    a. Suppose we have a P controller. Find an expression for the motor position using the definite integral of error.

    b. Is this model physically accurate? If not, explain why.

5. (*Differential Equations - Physics*) Suppose that we are using a P controller to control the motion of a heavy steel ball hanging on a spring of spring constant k. The ball's vertical position with respect to its equilibrium position (while connected to the spring) is given by y and the spring is damped with a damping force $B\frac{dy}{dt}$. The P controller is connected to an arbitrary mechanism that exerts a force $k_P y$ on the ball.

---

[39] Remember, potentiometers have a fixed range of values… You may simply code in the target in the error calculation with something like `error = 400 - SensorValue[Pot];`, where the target is 400.

[40] You must convert 3 meters to an appropriate number of encoder counts and use that as your target. (You want the wheels to rotate for some number of encoder counts that equals three meters traveled). Don't forget to clear your encoder when your code starts running!

a. Write the Newton's Second Law force equation for the ball in the vertical direction, assuming that the P controller is absent. Express it as a homogeneous second-order differential equation.

b. With P control, the force equation becomes a nonhomogeneous second-order differential equation

$$(\text{Part } a) = k_P y$$

Solve this differential equation for the position y as a function of time t.

6. (*Analysis*) Let C = $\{k_p\}$ be a set of positive[41] rational numbers for which a particular P controller system with that value as the proportional constant converges to near its target value. (You might think of this as a sort of power series). Prove that C is bounded. (Hint: This is trivial, but if you are still stuck, read the footnote[42]

[41] For good reason, the proportional gain is never negative – assuming your signage is fine.

[42] Manufacture a Cauchy sequence and use the Bolzano-Weierstrass theorem.

The Integral Term

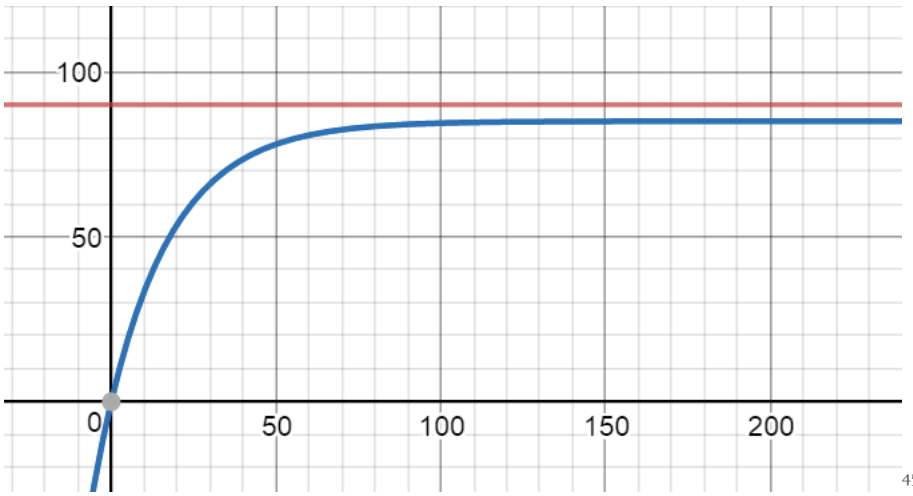
THE STEADY-STATE ERROR PROBLEM

Let us return to that example of moving the arm 90 degrees. Armed with our P term, this will get us fairly close to our goal of zero error. I am ignoring I and D for now; let's suppose that this is just a P controller or loop:

$$response = P = k_p \times error$$

Let's see what running our arm looks like on a graph over time. We start with our arm at position 0, and the `response` is repeatedly sent to the motor, driving the arm and causing the sensor reading (and thus `error`) to change. This is a poor depiction of actual data (made by a graphing calculator[43], not actual sensor measurements) – arms do not start off at full speed when commanded to run (they must speed up) – I put forth this graph only to establish a point.

The red line is the target value, 90 degrees. It does not change as time, shown on the horizontal axis with arbitrary units, goes on. The blue curve[44] is a (imagined) plot of sensor measurements over time. We want the blue curve to match the red as time goes on. The arm begins at a position of 0 (on the vertical axis) when the PID controller starts at time 0 (on the horizontal axis).



[45]

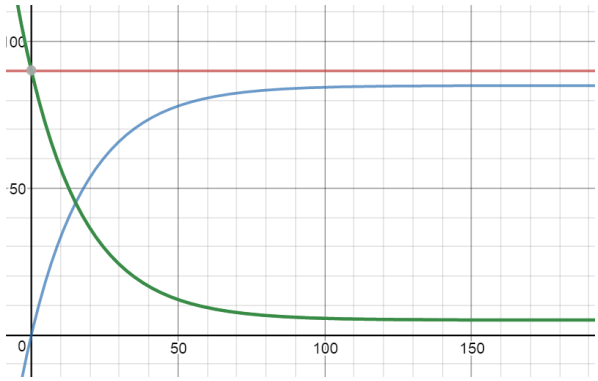
See a problem?

The sensor value approaches 90, the target – but never quite makes it there. Why?

[43] Desmos Graphing Calculator – highly recommended for anything up to single-variable calculus. Also, Wolfram Alpha is a thing.

[45] In reality, the arm would not instantaneously accelerate from a time of zero like this.

The error of this P loop over time can be visualized as the difference (length) between the red (target) and the blue (sensor) curve at any particular instant of time. If a plot of error is made for our graph, it would look like this:



When the arm first begins to move at time zero, the error is large – 90 degrees. However, near 90, the error becomes small – small to the extent where the value of the response (which, after all, is directly proportional to the error) is insufficient to cause the arm to move![46] Thus the sensor value never gets to 90, and some error always exists. This is known as **steady-state error**.

While this might be acceptable in many circumstances[47], the residual error (which some PID authors term **offset**) impairs accuracy.

A robot chassis with a P controller, told to go forward for 5000 encoder counts in autonomous, might stop dead at 4,900.

That is good enough to push a scoring object into a scoring zone. Not enough to hook a scoring object on a narrow pole. Certainly not enough to drive elsewhere and do the same thing again.

As a robot moves about in autonomous, any error in its movement will build up over time. If your robot oversteers in a turn, it will continue to drift off course as it moves about.

So accurate robot movement – accurate PID control – is vital in autonomous.

The precision of a full PID loop is necessary for applications like these. Therefore, the I and D terms are needed. Here we will explain the I term.

---

[46] As a programmer, you will most likely have experienced the effect of what most VEX roboticists call a motor **deadband** or **deadzone** – small motor power values (e.g. below 10 in magnitude) are insufficient to cause the motor to move. We have mentioned this before when discussing the limitations of PID. This deadband increases with motor load; so if your motor encounters large resistance forces (e.g. chassis weight, lifting objects) the deadband may be greater.

The exceedingly small error values result in P-controller-calculated response values that are in the deadband – so the motor never moves to correct that error.

[47] More than you might think. Not all robotics applications need to be incredibly precise. Especially in applications with little load (resistance), you might get away with a P controller like what we have considered.

I TERM INTUITION

In our example, we were encumbered with a perpetual, albeit small, amount of error. How could we "bump" the sensor value up to the target value, eliminating the residual error?

We could make the P term more responsive by increasing the value of k_p. But this will not guarantee no residual error. Furthermore, increasing k_p just to compensate for offset makes the system more unstable, prone to oscillations. So we should not modify the P term.

How could we add a little more kick to our PID (which, so far, is a P controller)?

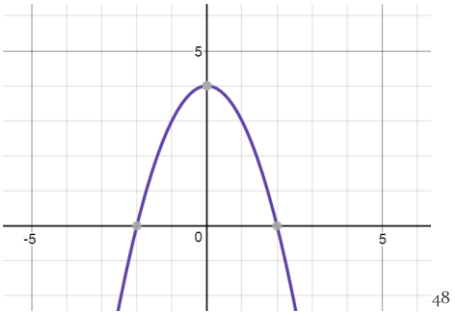
The integral term of PID consists of an **integral**, a mathematical operation that is one of the keystones of calculus. The other major centerpiece is the **derivative**, which is the bulwark of the D term. The integral will provide an additional boost to the system to eliminate the steady-state error.

Integrals Without Calculus

Before we address exactly what the integral term in PID is, we will address what integrals are and provide intuitive visuals for those who have not learned calculus. Instead of considering a definition, let's proceed through an example

From middle school and beyond, we could take the area of basic shapes, and perhaps even some more complicated ones like hexagons. But how would you take the area of a curved shape, like a region of area defined by graphs like parabolas, trig functions, and suchlike? Let's say we were asked the following:

> Find the area of the plane figure that is bounded by the parabola $f(x) = -x^2 + 4$ and the x-axis:



[48]

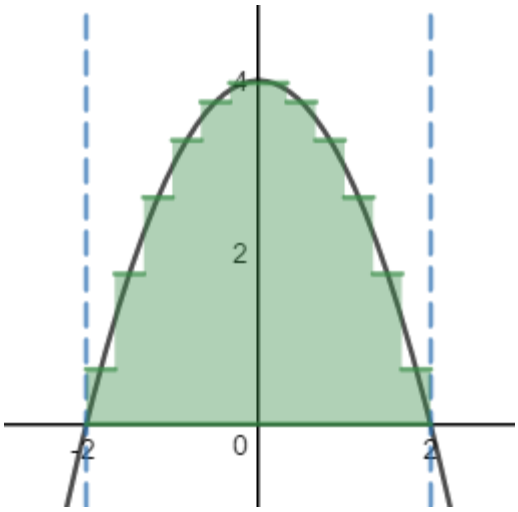
[48] If perchance you are unfamiliar with function notation, "*f(x)*" means that there exists a function *f* that takes in an *x*. We call the output of the function for any x "*f(x)*" – "*f(x)*" is a substitute for "y". For example, if *f* is the function $f(x) = -x^2 + 4$, then $(2) = -(2)^2 + 4 = 0$. This is analogous to stating that, if $y = -x^2 + 4$, then *y = 0* when *x = 2*. "*f(x)*" does NOT mean "f times x". Function notation is used for convenience: an Algebra II textbook might say, instead of "find *y* when *x = 2*", "find $f(2)$".

This is impossible for you if you haven't had calculus.[49] This problem simply does not make sense either.

But let's modify the problem a bit. How can you find an *approximation* for the area under this curve, the area between $f(x) = -x^2 + 4$ and the x-axis?

For starters, since the parabola roughly matches a triangle, we might find the area of the triangle with vertices (-2, 0), (0, 4), and (2, 0). But we soon realize that the approximation is a gross underestimate of the actual area.

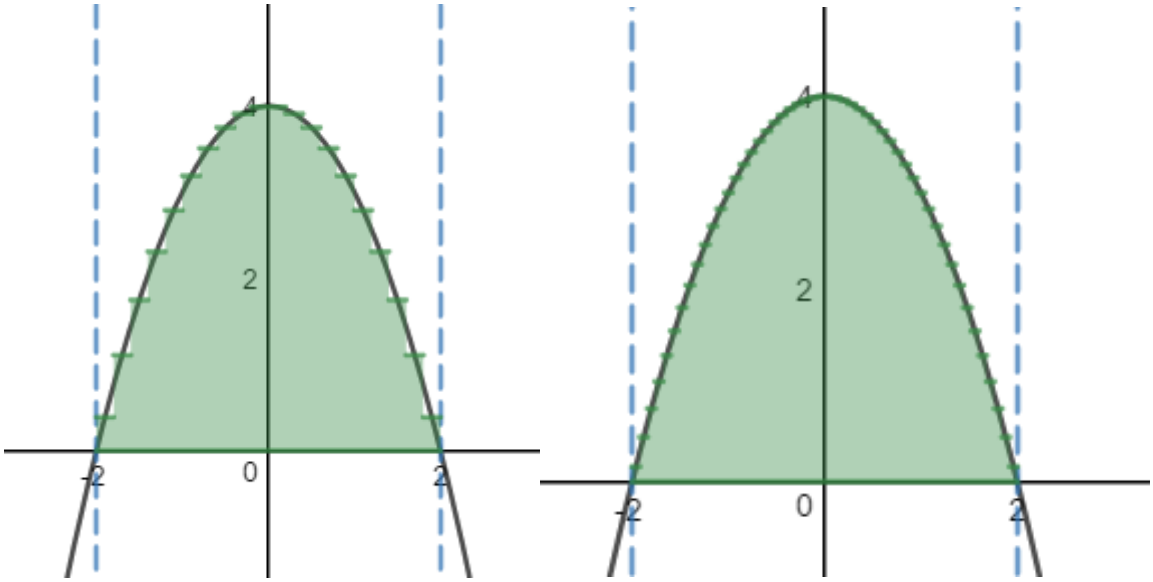
Then we might get creative and imagine doing something like this, approximating the shape of the curve with a bunch of thin rectangles with bases on the x-axis – rectangles whose area you can find easily by multiplying their common width by their height as dictated by the graph:



If you sum the area of each of those rectangles, **you can get a fairly good approximation of the area**! In calculus this rectangle summation is known as a **Riemann sum**.

Notice, however, with more and more rectangles of thinner bases, the approximation gets even better:

[49] Unless you go about deriving calculus by yourself.



What if there were a way to get the area of an infinite number of rectangles – so many that each one is impossibly thin? Wouldn't the area of those be exactly equal to the area under the curve?[50]

There is a mathematical way to do this, known as the **integral** in calculus (Calculus is a branch of mathematics[51]; integral calculus is a branch of that). The process of finding them is called **integration**.

Integrals are denoted and written like this

$$\text{"the integral of } x^2 \text{ with respect to x"} = \int x^2 \, dx$$

This doesn't mean anything useful to us right now. The dx symbolizes an infinitely small change in the value of x - the crux of the integral "area" idea: take rectangles whose bases are infinitely small changes along the x-axis. The expression x^2 here is known as the **integrand**, the expression or function being integrated.

[50] Major pains are taken in real analysis calculus courses to prove this – there are a bunch of technicalities surrounding whether a function is "integrable" – but we don't need to think about the why behind the math.

[51] Calculus as a field of mathematics deals with infinite and such "infinitely small" or "infinitesimal" quantities, like the famous $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \ldots = 1$.

$$\text{"the integral of f(x) with respect to x"} = \int f(x)\,dx$$ [52]

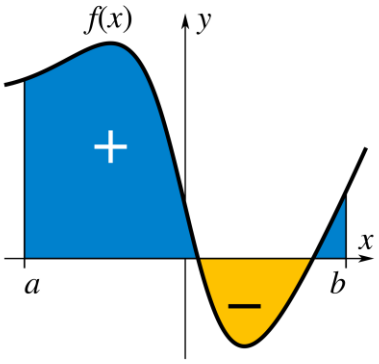
Again, not that useful for PID, but this is how it is written with function notation

$$\text{"the area under the graph of f(x) from x = -1 to x = 1"} = \int_{-1}^{1} f(x)\,dx$$

This sort of integral, a **definite integral** (notice those endpoints) is what we need for PID.

While an **indefinite integral** (the first two, without bounds) has no particular meaning useful for us, the definite integral is the area under the graph of the function being integrated, over the span of the endpoints).

In elementary calculus (and physics with calculus), we interpret the meaning of this definite integral as **"the area under the graph of f(x) from x = -1 to x = 1"** . "Under" here refers to the area between the function and the x-axis. Consider this representation of an arbitrary definite integral.



We see that area above the x-axis (where the function is positive) is counted as positive area, and area below the x-axis is negative.

The S-shaped symbol is the integration symbol, while the dx symbolizes an infinitely small change in the variable x – precisely what those rectangles depict in our area example.

---

[52] Note: These integrals without endpoints – indefinite integrals – are the reverse of the derivative, which I explain later. The result of such an integral is not useful to us now, but note the different results that an indefinite integral gives versus a definite integral (with endpoints)

For example, $\int x^2\,dx$ (indefinite) is actually a function, $\frac{1}{3}x^3$ (that we can get by solving with calculus methods). $\int_0^2 x^2\,dx$ is, however, a number $\frac{8}{3}$, which is the area under the graph of $f(x) = x^2$ from zero to two.

With PID we only concern ourselves with the idea of integrals as "area" – definite integrals

In effect, $\int_{-1}^{1} f(x)\,dx$ can be considered to mean "Consider infinitely thin rectangles that trace the curve of $f(x)$ from -1 to 1. Let them have arbitrary width, which we call dx, on the x-axis. Add up their areas". This infinite addition of infinitesimal rectangles becomes the actual area under the curve!

As impossible as this seems, a numerical solution is possible with calculus techniques.

The answer to our initial problem

> Find the area of the plane figure that is bounded by the parabola $f(x) = -x^2 + 4$ and the x-axis:

is, quite literally

$$"the\ area\ under\ the\ graph\ of\ -x^2 + 4\ from\ -2\ to\ 2 = \int_{-2}^{2} (-x^2 + 4)\,dx$$

which can be solved by calculus techniques, which give an answer of 10.6666666...

Another way to think about what the integral

$$\int_{-2}^{2} (-x^2 + 4)\,dx$$

is actually doing is this:

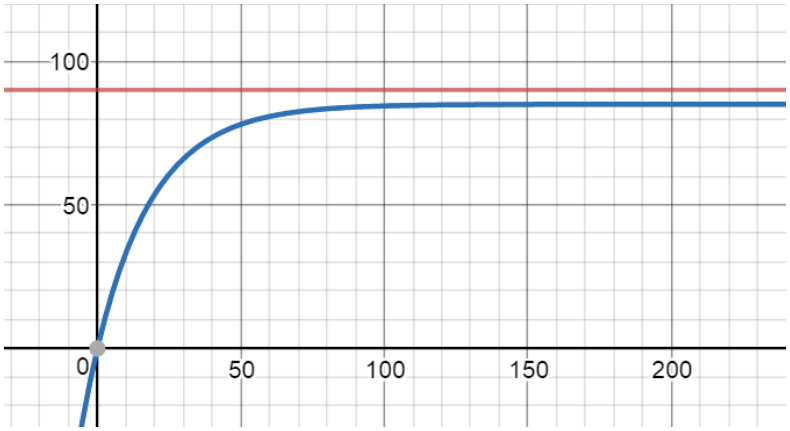
On the graph of $-x^2 + 4$, let's "sweep" along the x-axis starting at -2. Add the area of an infinite number of infinitely thin rectangles with width dx (an infinitely small "value" representing the infinitesimal base of the rectangles), which is $-x^2 + 4 \times dx$, to a running total from -2 to 2. This running total is "accumulated area" – the area accumulated by this "sweep" – which is the integral.

This idea of integrals as a "sweep" accumulating area is how integrals will work within PID.

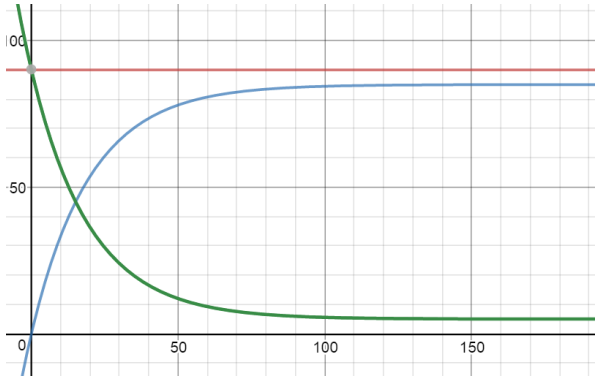
If perhaps you had difficulty understanding this quick explanation of integrals, you can either learn it online (Khan Academy, etc.) or talk to friends or a math teacher. Either way, you should learn calculus. Whether or not you will take a calculus class as a student doesn't matter.

INTEGRATION AND PID

Suppose we took a definite integral of the error in our PID system, our robot arm, over time. This can be interpreted as accumulated area under a plot of error versus time. The result is a positive number that increases when error is positive. Consider this graph of our system target and sensor value from earlier:

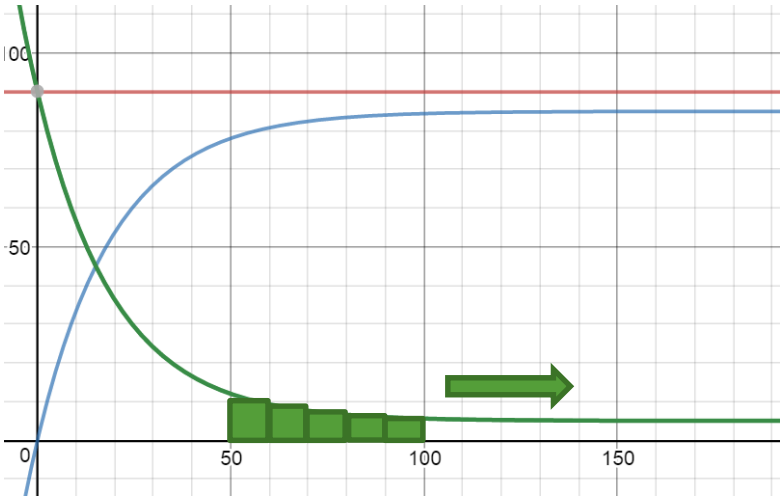


We know that the error is the difference between the target and the sensor value – between the red and the blue curves. If we plot this difference for all times shown on those graphs on a new curve, we obtain this result (green).



Now let's suppose we integrated this error curve over time – we take a definite integral. We are only concerned with this integral when the error is small – let's say less than 10 – I will explain why later.

We see that this accumulation of the area under the error curve – our definite integral – can be visualized like this. Assume we are currently at a time of 100 (units doesn't matter here)

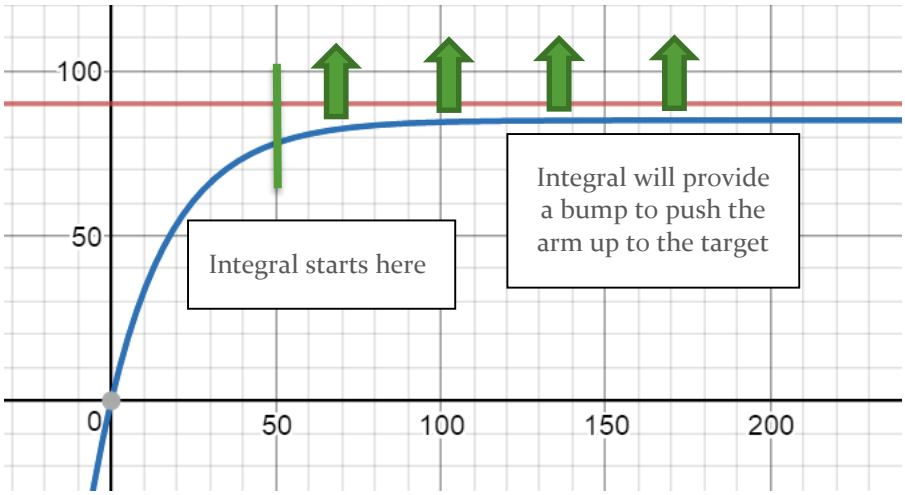


This is a Riemann sum of area under the error curve. Although the Riemann sum is not an integral – it is an approximation of "true" area under the curve – we will consider our integral of error as a Riemann sum to provide intuition and introduce how the integral will be implemented in software.

The sum begins at a time of 50, when the error is less than 10 (thereabouts). We sum the area of the rectangles under the curve as time goes on. Initially, at a time of 50, there are no rectangles so far on the graph – so the integral is zero. At a time of 60, we have one rectangle, whose area is 100 (10 units on the vertical scale times 10 on the horizontal), so our integral has a value of 100 (approximately – remember that a Riemann sum is an approximation) As time goes on, the areas of the successive rectangles are added to our running som. At our current time, 100, we see that our integral is about 300 in value, which is the total of the five rectangles.

This is what an integral of error over time would look like.

Now we can finally address how the integral will help us resolve our residual error problem. The Riemann sum of these rectangles, taken over time, will be added to the P term to give an additional bump to the PID response. This is what will happen as the error approaches the target.



Visually, the area from the Riemann sum is very small – compared to how large the error is initially. But this area – this integral – is enough to add an extra kick to help eliminate the small remaining error that the P term leaves behind. Since the `I term` is added to the P term, this integral will add to the PID motor

`response` for our robot arm and force the motor to move a little more, bringing the arm to its target position of 90 without error.

The same applies if our arm were going down instead, and had negative error remaining. Since the error values over time are negative, the sum of these errors is negative, and the `I term` will provide a negative, downward kick to the motor.

We can represent the **integral term** of PID with mathematics:

$$I = k_i \times \int_0^t error\ dt$$

where dt refers to the fact that this integral of error is taken over time, from zero (when the PID starts out) to some time t (where we are currently at) later. As with the P term, we have a gain constant here, k_i, to scale the integral appropriately.

Although this formula suggests that the integral is a definite integral, the bounds do not matter, and this integral should be interpreted as simply "**the total accumulated area of an error graph over time**". I just use the definite integral here to remind us that the integral is taken from time zero (when the PID starts) to some time in the future (where we are now, after the PID has begun).

Most authors on PID represent the integral as an indefinite integral. However, the bounds we have here remind us that what we are after is a number – not a function (as an indefinite integral would give us a function) – a number that factors into the PID `response`.

PROGRAMMING THE I TERM

However, we have a problem that renders the math useless (and also leaves us without the need to learn how to actually evaluate integrals): while the math is valid, the Cortex – our ulterior objective is to program PID on a robot, yes? – cannot take integrals!

Anyhow, since the math of integrals is not especially intuitive with PID, we can consider the `I term` as this instead:

$$I = k_i \times (the\ accumulated\ area\ under\ the\ error\ curve\ as\ time\ goes\ on)$$

What is *the accumulated area under the error curve as time goes on*? This is a definite integral of error over time – which we can approximate with a Riemann sum. We did this with the Riemann sum from before!

So when we program the `I term`, we can implement a Riemann sum that adds up the area under an (imaginary) error graph. On each repetition of the PID loop, we take the current value of the error and multiply this by the delay time of our loop (the time between each repetition. The result, for one repetition, or iteration, is the area of an imaginary rectangle under the error graph for that instant of time.

However, consider that each of our imaginary "rectangles" will have the same base – the delay time of our loop. We see that

$$\begin{aligned} accumulated\ error &= (delay\ time\ \times error_1) + (delay\ time\ \times error_2) + (delay\ time\ \times error_3) \ldots \\ &= delay\ time(\ error_1 + error_2 + error_3 \ldots) \end{aligned}$$

The delay time is just a constant value that can be factored out. Since there is already a constant in our `I term` – k_i– we can negate this constant value in entirety since we can simply adjust the value of k_i to account for the influence of the delay time. So our formula for programming the I term becomes

$$I = k_i \times (the\ sum\ of\ error\ values\ over\ time)$$

As the PID repeats, we add the current error value – whether positive or negative – to this running sum. We then multiply this running sum by k_i, and use this as our `I term`.

The integral term fulfills two major purposes in PID:

- Correcting for a small error (for which the P term will not provide enough response). This is the principle behind our example of a robot arm and the need of an extra "kick" in addition to the P term.
- Allowing the PID system to tolerate changing load, or changing forces on the motor(s). The integral term will ensure that error will still be as close to zero as possible even if, for example, the amount of weight a PID-controlled arm changes.

To implement this in code, we have several options. We could utilize am `integral` variable to hold the error sum, and multiply this by k_i later when calculating P + I + D:

```
integral = integral + error;
```

Alternatively, we could directly factor k_i into the value of `integral`. If we use this, we will not need to multiply `integral` by k_i when we compute the PID sum.

```
integral = kI * (integral + error);
```

For efficiency, we could write the two statements as.

```
integral += error;
```

```
integral += kI * error;
```

Limiting the Integral

So the inclusion of an integral term in our robot arm P controller will give us a slight "bump" that, with appropriate gain constants, will be enough for the arm to end up at 90 degrees, the target position. This sort of PID control consisting of P and I terms is known as a PI controller[53].

However, we still have a problem. I mentioned earlier that the integral should be taken only when the error is small. Why? Well, …

[53] Various combinations of P, I, and D are possible and frequently utilized in real life: P, PI, PD, and PID

Most PID controllers are actually PI controllers, with no D term. However, these PID controllers are usually used in industrial applications (i.e. controlling industrial equipment) where the need to forcefully counter large disturbances (what the D term provides) is usually not pertinent.

It is rare to find an integral-only controller. If you are interested, research **Take-Back-Half** – a popular controller utilized by many VEX teams for flywheel velocity control during VRC Nothing But Net. This is a controller that revolves around an I term with an additional limit on how quickly the integral can grow so that the response to error is stable. I used Take-Back-Half during that year but found it inferior to PID in controlling motor velocity.

BEWARE: THE I TERM IS UNSTABLE

- Since the I term accumulates error over time, it can get very large, very quickly – especially if error itself is large.
    - Suppose you have an error of 4095 potentiometer counts for 200 milliseconds, and that your PID runs once every 20 seconds. That's a sum of 4095 for 100 times – 40,950. Not at all in the 0-127 motor range. Even if k_i were fairly small, at 0.01 for example
- Compared to the proportional gain constant k_p , the integral gain k_i should be very small, since the raw accumulated error value will be comparatively enormous.
- Too large of a k_i results in perpetual oscillation of the system, around the target value, This is an unstable system.
- The I term must be limited in some way to prevent excessive "build-up" of the integral

The last point is worth expanding. If the integral term is not limited, a large error (caused by, say, a sudden shift in the desired target position of our robot arm) will cause the integral to become very large. However, once the arm does reach the target position, the large integral will cause the arm to overshoot. Since the integral accumulates error as time goes on, it will require a long time until the integral returns to around zero. For example, if our arm's target was increased significantly, the large positive error that persists until the arm begins to pick up speed results in a highly positive integral. This integral would require a long duration of negative error to zero out. Obviously we have a problem if the arm reaches the target, and the build-up, or **saturation**, of the integral tells the arm to keep going up. This increases the time it takes for the system to stabilize and could even destabilize the system entirely, preventing the arm from ever settling down. This unwanted build-up is usually termed **integral windup**.

What we have done in our example – by taking the integral only when the error was small – is one way of limiting the size of the integral. I term this method the **critical error** or **error threshold** approach, since the integral is taken only when the size, or magnitude of the error (absolute value) is less than some critical value, which may be interpreted as the error being within some threshold. If the error is outside of this threshold – if the magnitude is too large – we set the integral to zero. To date all PID controllers in Robotics have used this method.

An alternative method is to limit the actual size of the integral – on every PID loop, we sum the error to the integral as usual, but we restrict the value of the integral to within some threshold. This can be done with a set of if-statements – if the integral is too large, set it to the maximum value; if the integral is too small (negative), set it to the negative of the maximum. I term such a method the **integral limit** approach. However, we must take care to avoid some confusion since the critical error value used in the error threshold method is usually termed the **integral limit** by us in Robotics who know PID.

There are alternative approaches found in other PID literature. Because of the different integral-limiting methods possible, we should take care to add a comment to our PID code stating how the integral is limited.

PROBLEMS

1. (**Introduction to PID**) Do Exercise 1 from the previous section. Using the ROBOTC Sensor debugger window, find the steady-state error. You might want to adjust the proportional gain constant a bit and see if the error improves. Now add an I term to your P controller. By how much can you improve the accuracy of the system (reduce the error)? Try to tune the system (find good gain constant values) as perfectly as possible. Remember that potentiometers are not perfect; zero error is not feasible.

2. (**Introduction to PID**) Do Exercise 1 from the previous section. Add an I term to the P controller and see how much you can improve the steady-state error (how far your robot actually moves, versus the target distance of 3 meters) with this PI controller.

3. (**Introduction to PID**) Describe what happens if the integral gain constant is too small, and too big. Why is the integral gain constant usually much smaller than the proportional gain constant?

4. (*Feedback Control*) Suggest the characteristics of an integral-only controller. (Yes, these do exist).[54] Explain what different values of the integral gain constant (i.e. too small, just right, too big) would cause.

5. (Analysis) Let C = $\{k_i\}$ be a set of positive[55] rational numbers for which a particular PI controller system with that value as the integral gain constant converges to near its target value. Suppose that if the integral gain is zero, the P controller that remains does settle near the target by itself. Prove that C is bounded. (Hint: There is a certain test for convergence of series involving an integral…).

[54] See Take-Back-Half, described in the "Expanding PID" section.

[55] For good reason, the integral gain is never negative. Why?

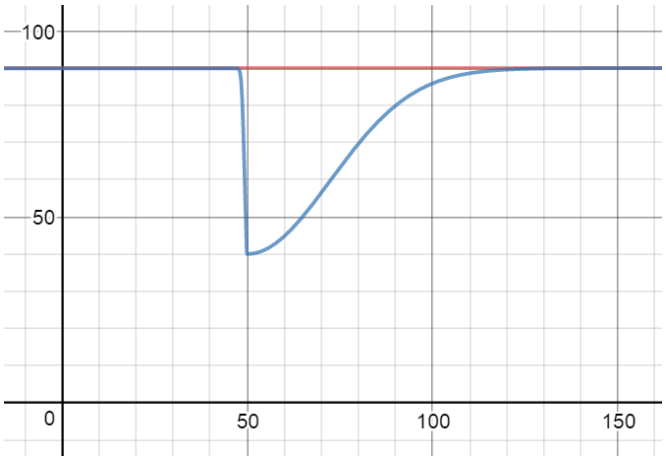
The Derivative Term

RESISTING DISTURBANCES

There is one final problem confronting our arm example, which so far is driven by a PI controller. This is a problem that frequently appears with systems that require feedback control.

Suppose that our arm is sitting at its target value of 90 degrees, stable and not moving. Then someone gives it a forceful push downwards. In feedback controller terminology, this sort of unplanned event is known as a **disturbance**. Obviously, this particular scenario is unlikely to happen in competition – but other disturbances might occur, such as unexpected force on a lift.

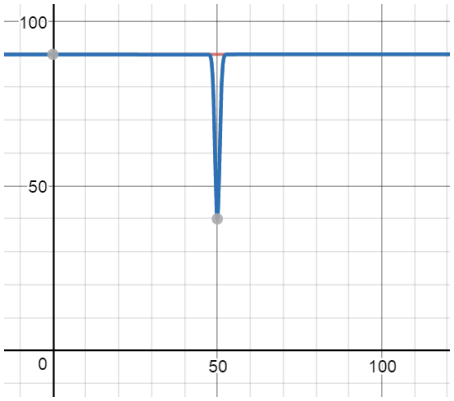
A downward push to our arm might look like this on a graph:



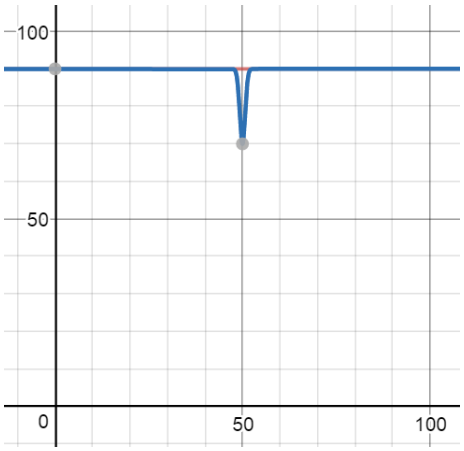
As usual, the red line is our target value (90) and the blue curve represents the sensor value. The graph suggests that the push forced the arm down to a sensor value of about 40. Then the PID (which, so far in our considerations, is a PI controller) recovers, bringing the sensor value back up to the target.

Note that our PI controller does eventually recover from this disturbance. However, this recovery is fairly slow, and the disturbance itself causes error to spike to a great degree.

Perhaps we would like our system to have a greater ability to resist disturbances – being able to return to the target more quickly:



Notice the difference between this graph and the previous – the recovery time is much, much lower. Alternatively, we could do better:



The arm itself is displaced far less from the target. This indicates that the PID actually resisted the disturbance to some extent.

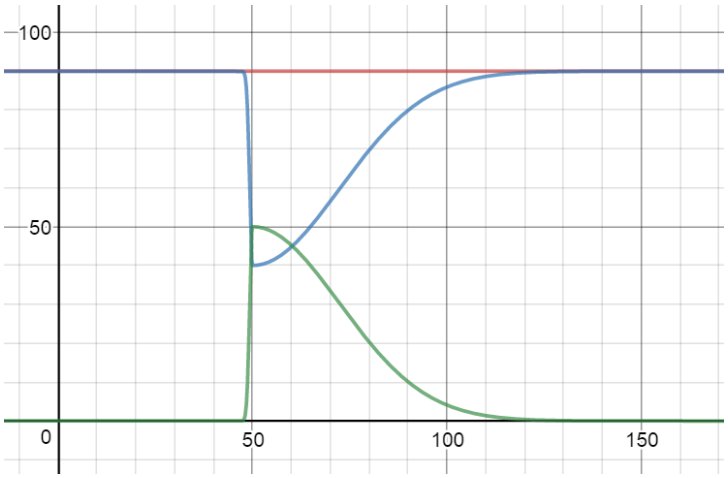
This is ideal behavior is response to a disturbance. However, such behavior is impossible with only the P and I terms.[56] This is where the D term comes in.

D TERM INTUITION

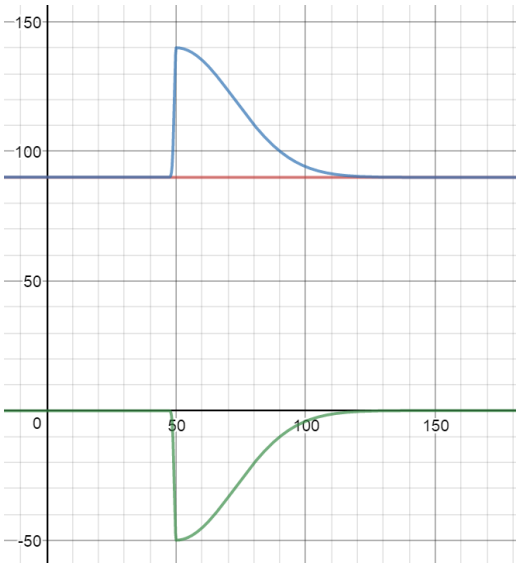
How might we add to the PI control that we have established to better resist disturbances?

From our arm example, we see that the error spikes whenever there is a disturbance. In our case, the error suddenly becomes positive:

[56] Given rather large, sudden disturbances – unless you have an extremely large k_p and k_i and your system somehow does not oscillate, or "bounce", around the target value.



If our disturbance were instead directed upward, the error would spike negatively – since the difference between the target and the sensor value, now greater than 90, is negative:

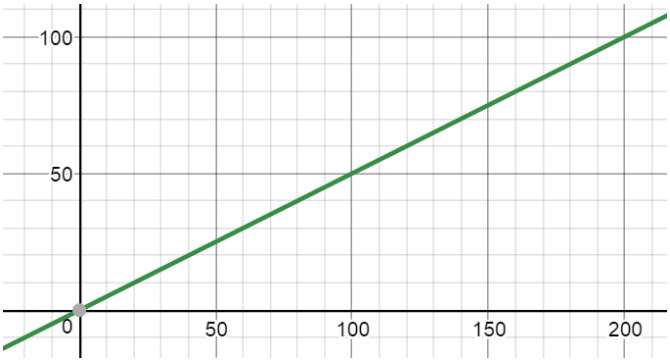


We see that, if a disturbance causes error to rise, we need a positive – upward – "kick" to help drive the system back on target. This is the case in our example. If the disturbance caused error to become negative, the kick should be negative.

What if we based our "kick" response to disturbance on whether such error spikes occur? How might we detect such error spikes (i.e. when a disturbance occurs)?

In any case, we see that the magnitude of the error increases rapidly in a very short time. Perhaps we could detect error spikes by measuring the change of error over time: when error changes rapidly in a very short time (error spike), we know that we have a disturbance.

What is the change of error over time, mathematically? This is but a slope of a graph of error versus time – a graph that is also a straight line.



We know that, to find the slope (commonly denoted m) of this graph, we use the eternally popular formula

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

with any two data points (x_1, y_1) and (x_2, y_2). This formula is more properly known as a **difference quotient**.[57] We may instead write this:

$$\frac{\Delta error}{\Delta t} = \frac{error_2 - error_1}{t_2 - t_1}$$

The Greek letter delta, Δ, symbolizes "change in".[58] Thus the equation indicates the change in error over, or per, change in time (t). In physics, the change of some quantity over time is usually significant – the change of position over change in time is called **velocity**[59], and the change of velocity over change in time is **acceleration**. However, the change in error over time is nothing special.

But, as we have seen, error is almost never a straight line! How can we find the slope, the change of error over time, of a curved graph of error?

[57] This term more commonly refers to the expression for the definition of the derivative – which is technically the slope formula as well – coming up soon

[58] In elementary, algebraic physics, it is common to see delta-quantity over delta-quantity, as we have here.

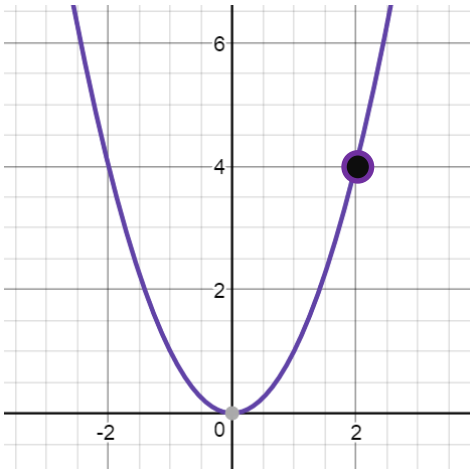
[59] Velocity is not "speed" – velocity can be negative (if the change in position is negative); speed is the magnitude, or absolute value, of the velocity – speed is always positive, or zero.

For our arm, we use the terms "angular" with the position, velocity, and acceleration of the arm in a physics context, since the arm's motion is rotational. If we found the slope of a straight-line graph of our sensor value plotted over time in seconds, we would have the arm's angular velocity in units of (sensor unit) per second.

Derivatives Without Calculus

Let's consider an example:

> Find the slope of the graph $f(x) = x^2$ at the point $x = 2$.



As with curved-shape area problems, such a problem is impossible for you without calculus. This doesn't even make any sense. How can a curved graph have a slope – a slope that depends on only one point?

However, this would make more sense from a physical standpoint.

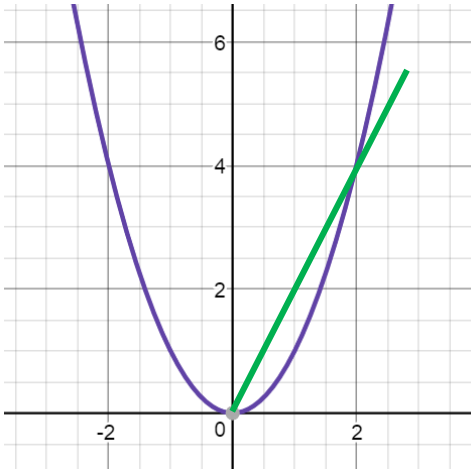
Consider a car that speeds up, from a time of zero ($t = 0$) onward, according to the equation $x(t) = t^2$. This is the same equation from before, but with physical quantities instead.

This car begins from a position that we call $x = 0$ and speeds up in a straight line. At every value of t (in seconds), the car is located at $x(t) = t^2$ meters away from its starting point.[1]

The aforementioned interpretation of slope leads us to conclude that we can easily find the average velocity of the car between any two intervals of time. Suppose that we wished to find the average velocity of this car over the first two seconds of its journey. We can then say

$$velocity = \frac{\Delta x}{\Delta t} = \frac{x_2 - x_1}{t_2 - t_1} = \frac{x(2) - x(0)}{2 - 0} = \frac{4 - 0}{2 - 0} = 2 \frac{\text{meters}}{\text{second}}$$

We have effectively found the slope of this green line, which is clearly two (meters per second, with the units that we have assigned):



Obviously the negative half of the curve does not apply.

The slope, 2 meters per second, means that between time 0 and 2 seconds, the car is moving with an average velocity of 2 meters per second. It covers four meters in these two seconds.

But we know that cars have speedometers that can tell the precise speed of the car at any given moment. If we did this with velocity, this would be known as the **instantaneous velocity** of the car (in contrast to average velocity). The instantaneous velocity is the instantaneous change in position per change in time – the slope – at a single moment in time.

So perhaps instead of saying [60]

> Find the slope of the graph $f(x) = x^2$ at the point $x = 2$.

[60] The following is not a proper statement of the concept of instantaneous change.

We should perhaps state

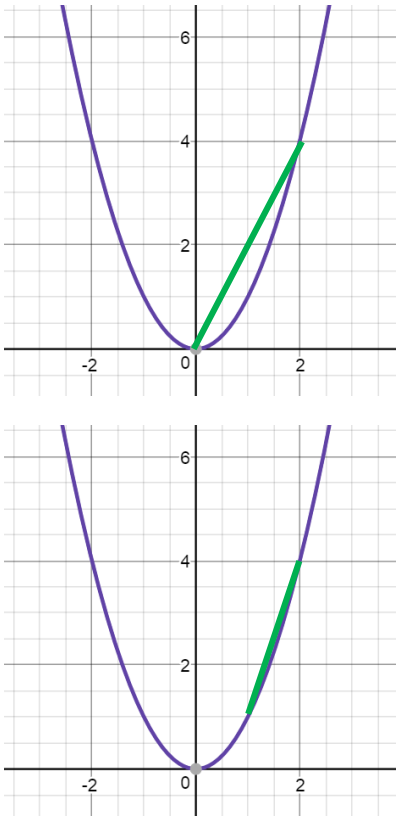
> Find the instantaneous change of $f(x)$ for a very small change in x of the graph $f(x) = x^2$ at the point $x = 2$.

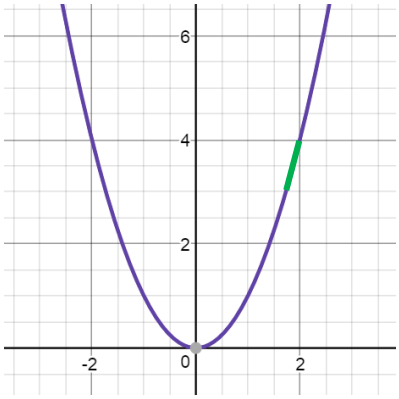
How small? As with integrals, infinitely small.

We know that the difference quotient (the slope formula) works for any two points on a graph.

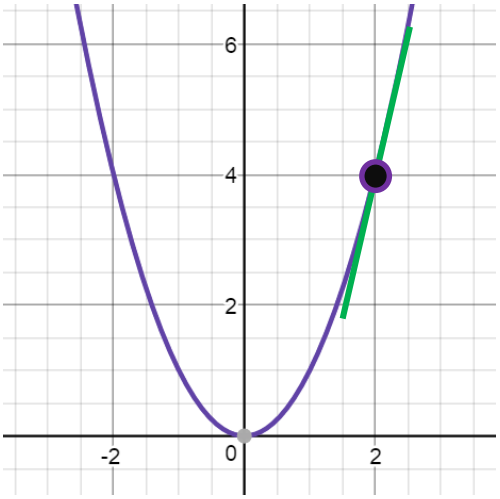
$$\frac{\Delta f(x)}{\Delta x} = \frac{f(x)_2 - f(x)_1}{x_2 - x_1}$$

What happens if both x_2 and x_1 approach the desired point, 2? Obviously they can't both be 2 – then the difference quotient results in division by 0, which is undefined or infinity, whichever you choose. But what happens when the points used in the difference quotient *approach* each other – but are not the same? Let's begin with x = 0 and x = 2:





The line begins to closely resemble the "true" instantaneous change at 2! If we could make the distance between the two points infinitely small (around $x = 2$), we begin to obtain what is known as a tangent line – which, much like the tangent line to a circle, intersects the curve at only one point, $(2, 4)$.



The slope of the tangent line is thus the slope that we are after!

$$\frac{\Delta f(x)}{\Delta x} = \frac{f(x)_2 - f(x)_1}{x_2 - x_1} \; where \; x_2 - x_1 \to 0 \; and \; x_2 \approx x_1 \approx 2$$

We should note that the tangent line intersects the graph at $x = 2$ – since there are infinitely many different tangent lines given infinitely many values of x.

We could express the difference $x_2 - x_1$ as a variable, h.

We can eliminate the subscripts 1 and 2 from the variables entirely and just call x_1 as x – the x where the tangent line that we after intersects the curve – and x_2 as $x + h$.[61]

The difference quotient thus becomes

$$\frac{\Delta f(x)}{\Delta x} = \frac{f(x+h) - f(x)}{h} \; where \; h \to 0 \; and \; x = 2$$

[61] Why do this? This will be evident later on.

This is merely a restatement of "$x_2 - x_1 \to 0$ *and* $x_2 \approx x_1 \approx 2$" into "let $x_1 = 2$ and suppose that $h = x_2 - x_1$ is infinitely small." Since $x = 2$, we can simply plug that in:

$$\frac{\Delta f(x)}{\Delta x} = \frac{f(2+h) - f(2)}{h} \; where \; h \to 0$$

With proper mathematics that we do not need to understand, we can express this more formally as

$$\frac{\Delta f(x)}{\Delta x} = \lim_{h \to 0} \frac{f(2+h) - f(2)}{h} \quad [62]$$

The value of this difference quotient – this is still a slope formula! – is called the **derivative** of $f(x)$ at $x = 2$. The derivative is the **instantaneous rate of change**, or "slope" of the function at the point where $x = 2$. What we have done is that we have found the slope of a tangent line – the tangent line, in fact, as there can only be one – that intersects the graph of the function at $x = 2$. The slope of the tangent line is equivalent to the instantaneous rate of change of the function at $x = 2$.

The proper mathematical language to express our derivative is "*the derivative of* $f(x)$ *with respect to* x *at* 2".

We can solve this with calculus techniques, resulting in a number: 4.[63] This number indicates that at the point where $x = 2$, any minute change in the value of x will "cause" y to change by four times as much. The graph "actually" has a slope of 4 at that point.

This derivative, 4, has a special meaning depending on what we took the derivative of. If $f(x) = x^2$ describes the location of a car $f(x)$ (in meters) along some straight line as time x (in seconds) goes on, the derivative of 4 at $x = 2$ indicates that the car, after two seconds in motion, is instantaneously moving at a speed of 4. 4 what? Look at the units: any slope of this graph will have units of meters divided by seconds. So the derivative has units of meters per second.

With any particular value of x, the value of the derivative of $f(x)$ at that x is

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \quad [64]$$

The formula for the derivative, at its heart, is nothing more than a difference quotient – albeit a special difference quotient because of the infinitesimal h involved. Because of this, instead of representing the derivative with

---

[62] In calculus, this expression is known as a **limit**. The meaning of $\lim_{h \to 0} \frac{f(2+h)-f(2)}{h}$ is nothing more than "the value of $\frac{f(2+h)-f(2)}{h}$ when h gets infinitely close to zero". Unless you are in a real analysis course, or are a freshman at Caltech, where stuff gets ugly. Obviously, you can't divide by 0, but there are ways around that that you will learn in calculus class.

[63] Or we can extrapolate the slope after approximating/fitting a tangent line on the graph. The process of finding derivatives is called **differentiation**.

[64] This is the most common form of the definition of the derivative. Very important in beginning calculus, but hardly ever used due to the presence of shortcuts for taking derivatives (unless you are taking a real analysis course). This is unnecessary for PID.

$$\frac{\Delta f(x)}{\Delta x}$$

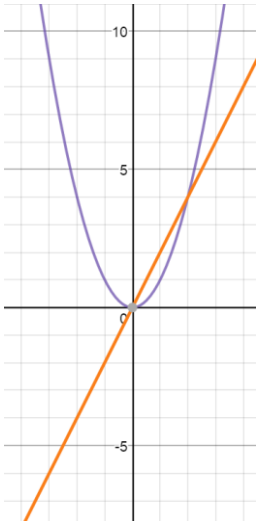
we use the symbols

$$\frac{df(x)}{dx} \quad \text{or} \quad \frac{d}{dx}f(x)$$

The d may be thought to represent "an infinitesimal change in". Therefore, dx is an impossibly small change in x. This is identical to the dx present in integrals from earlier.

We should note that, since derivatives exist at all points on a function,[65] the derivative of a function in general may be considered as a function that gives the value of the derivative at every point on the original function.

For example, we could visualize how the value of the derivative of $f(x) = x^2$ will change as x changes. Initially, when x is far less than zero, $f(x)$ is decreasing – very much so. $f(x)$ then levels out to zero, and then begins to increase – and increase at a rate that is itself increasing. Intuitively, the derivative of $f(x)$ is initially very negative (since the function's instantaneous rate of change is very negative). The derivative then increases to zero, and becomes positive. This suggests how the value of the derivative changes as x changes. The graph of the derivative of $f(x)$ for all x is a straight line, shown in orange here:



At $x = 2.$, the derivative, the value shown by the graph of the derivative at 2, is 4 – as we have stated. This derivative function is in fact the line $2x$.

We previously said that the change of position over change in time of some moving object is called velocity, and the change of velocity over change in time is acceleration. Now we see that the derivative of a position function of time of a moving object is a function – one that gives the instantaneous velocity[66], the

[65] At least, a continuous one, where there are no jarring gaps. There are conditions on when a function is **differentiable** too – whether a derivative can be taken at particular points on the function.

[66] This is distinct compared to average velocity – while average velocity is a difference quotient of position over a time interval, instantaneous velocity is the derivative of position at a particular moment of time. The same concept can be extended to (instantaneous) acceleration, the derivative of velocity.

instantaneous rate of change of position, of the object at any time– and the derivative of that velocity function is another function – one that relates the acceleration of the object to time!

However, in PID, we require only the concept of the derivative as the instantaneous rate of change, a numerical quantity.

DIFFERENTIATION AND PID

The `derivative of error with respect to time` gives us a convenient value for the rate of change of error over time at any particular time that we want.

$$\frac{d}{dt}error \quad \text{or} \quad \frac{d(error)}{dt}$$

With this we can build the D term. Its value is greatest when the rate of change (derivative) of error is greatest (and likewise when error is negative), thereby providing a "kick" to the `response` necessary to counteract disturbances. When the rate of change of error is small, the derivative is negligible and the D term is small.

So what happens in our robot arm example? Suppose that our robot arm is given a large push downwards. Then error rapidly increases (remember, we have defined error to be positive when the arm is below its target value). Initially, $\frac{d(error)}{dt}$ – the rate of change of error– is zero, since the arm is not moving. When the arm is pushed, though, $\frac{d(error)}{dt}$ is positive, since the error increases as time goes on. Since $\frac{d(error)}{dt}$ is positive, the `D term` is positive, which adds to the motor PID `response`. The extra kick helps to drive the arm back to its original position.

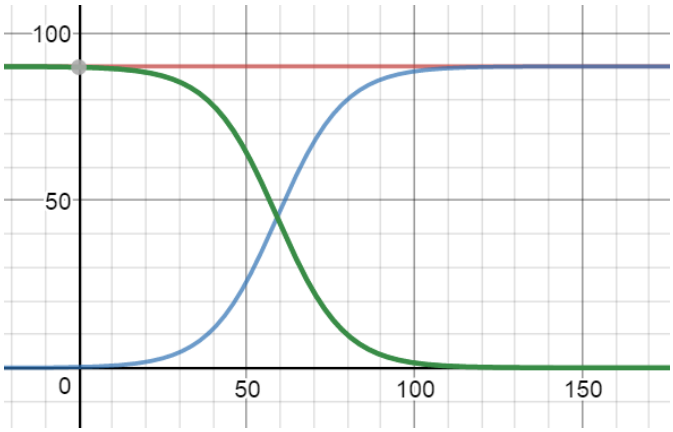
Remember that the `D term` m is simply added into the PID sum with P and I.

Mathematically, the **derivative term** of PID looks like this:

$$D = k_d \times \frac{d}{dt}error \quad \text{or} \quad D = k_d \times \frac{d(error)}{dt}$$

As with P and I, a gain constant – k_d – is also necessary here to scale the value of the `D term` depending on your particular PID and PID-controlled mechanism.

There is one more thing to consider. Suppose that our arm example system is disturbance-free, and is simply approaching a target value. With a perfectly tuned PID, with proper values for the gain constants, the PID quantities – target, sensor value, and error – of such a system might appear like this on a graph:



Notice how, as the PID progresses and the sensor value begins to approach the target and "slow down", the error decreases with time – $\frac{d(error)}{dt}$ is negative (albeit the magnitude of the `derivative` is changing). Therefore, the `D term` is negative, which subtracts from the positive response of the overall PID sum.[67] This helps to make the PID more stable and less likely to oscillate given large gain constants for P and I. If our arm were going downwards instead, the same concept applies – the sign of the D term, in all such cases, would be the opposite of that of the P and I terms.

The derivative term fulfills two major roles in PID control:

- Helping to compensate for sudden changes in error (e.g. A robot arm motor will quickly increase motor power in response to suddenly being pushed away from its target position).
- Making the convergence to zero error – the approach of the motor to the desired target value – more mellow, less prone to overshooting and oscillating above and below the target.

The derivative gain constant, k_d, is usually about as large as the proportional gain k_p and somewhat smaller. However, changing k_d will not affect the system stability as dramatically as changing either k_p or k_i will, so a large range of k_d values will result in good system behavior.

[67] How do we know that the sum is positive? How does the sensor value change over time?

PROGRAMMING THE D TERM

There is but one problem remaining – as with integrals, the Cortex cannot find derivatives exactly. So we must adapt our definition of the derivative term for programming, in which we opt for the next best thing – a difference quotient. After all, this is how we conceptually arrived at the idea of a derivative. This approximates the change in error over a time interval.

$$D = k_d \times \frac{error_2 - error_1}{t_2 - t_1}$$

1 and 2 are simply different error measurements over a certain time interval. We take a measurement of error at time t_1 and again at time t_2, and we calculate the slope between these two points.

However, the denominator $t_2 - t_1$ can be constant (for example, the PID loop fetches data every 50 milliseconds). In this case, the denominator doesn't matter (since its effect can be incorporated into the value of k_d, as with the I term). So we can get rid of it and keep

$$D = k_d \times (error_2 - error_1)$$

Notice that our conditions for removing the denominator are satisfied by a PID loop – small intervals of time between each iteration (as we established with the integral).

We need two distinct error values for this calculation – one taken at a certain time, and one taken some time after. Then because this is PID, we must repeat this calculation with the rest of PID as time goes on.

Notice, however, we can consider $error_2$ as our "current" error value and use a "past" error value – one obtained from the PID loop some time ago – as $error_1$. We refer to this past error as the previous error, usually denoted `lastError` as a variable – since it is the value of the error from the last loop iteration. We have, then

$$D = k_d \times (error - lastError)$$

This is precisely how the D term is implemented in software. As with the integral, can either leave the gain constant k_d to later (to the P + I + D sum calculation), or factor it into the calculation at this time:

```
derivative = error - lastError;
```

```
derivative = kD * (error - lastError);
```

The derivative variable is even unnecessary; we could simply add its value (the difference in errors multiplied by k_d) in the P + I + D sum.

However, we need to add something else. Notice that the error of "this" iteration will become the `lastError` of the "next" iteration. Since the current error becomes the last error of the next iteration, we need to set the value of `lastError` equal to the value of the current error so that the correct `lastError` is used next time around the PID loop. This must be done after the `derivative` calculation, before the `error` is re-calculated on the next PID iteration.

```
lastError = error;
```

The value of `lastError` should be zero when the PID starts, since there is no previous error.

EXERCISES

1. **(Introduction to PID)** Do Exercise 1 from the previous section. Now give your PI-controller-equipped arm a nice wallop.[68] The arm will, of course, fight back (even without the D term – why?). Now add a D term. Play around with the derivative gain constant. How much can you improve the arm's response?

2. **(Introduction to PID)** Explain why a D term might not be necessary with slow, predictable applications (like moving a robot forward for three meters without any obstacles or interference, as in the previous Exercise 2s). Alternatively, explain why a PI controller might be enough to ensure high accuracy in such applications.

3. **(Introduction to PID)** Identify some possible PID-controlled systems that would or might benefit from a D term Use some real-world examples too!

4. **(Introduction to PID)** List characteristics of a well-tuned PID system. (Hint: Think about all that has been discussed Remember that time is ~~money~~ valuable!)

5. (*Feedback Control*) Suppose you have a system where the target value changes (e.g. a robot arm that you wish to control by changing the target position of the arm). Is a derivative-only controller possible? Show why or why not. In general, is a derivative controller possible? Justify.

6. (*Feedback Control*) Propose characteristics of a system where a PD controller (no integral term) might be enough for your get-it-to-the-target needs. Support your answer.(Hint: Why did we need an integral term anyway, in general?)[69]

[68] By "nice", I mean "don't break the gears".

[69] I have found that, unless you really need zero steady-state error and are willing to put up with tuning the value of k_i, which I have found to be more difficult than with the other two gain constants, a PD controller is sufficient for most position-based user-controlled PID systems, like the robot arm.

PID All Together

We know that the math of PID simply states that the response to error should be the proportional term plus the integral term plus the derivative term. And we know that the value of each of those terms is dependent on error – and how error changes over time.

All that's left is to sum together the P, I, and D terms after each term has been calculated, and send the result to our system motor(s).

How this sum will look in programming depends on how we have defined our variables, especially our `integral` and `derivative` variables – whether the gain constants were already included in the variables. Assuming that they were not factored into the calculation of these variables, our sum statement will appear like this:

```
motor[pidMotor] = (kP * error) + (kI * integral) + (kD * derivative);
```

> Note that the sum P + I + D could frequently exceed the threshold for motor power values, [-127, 127]. So we should store the sum in a variable and use a set of if-statements to limit its value to that threshold (like an integral limit). However, ROBOTC will do this for us automatically, in the motor assignment, if we forget. Or nothing bad happens anyway.

There could be some constant in front of the PID sum to scale the sum. One purpose of this might be to correct for error signage issues: if the convention that we have established for our error signage causes the motor response to increase – not decrease – the magnitude of the error, we could fix this by multiplying the sum by -1. Or we could simply reverse the motor direction in ROBOTC's Motor and Sensor Setup. Or physically reverse the connection to the motor.[70]

Another use for such a constant is for personal taste: some programmers like to do all P, I, and D calculations in terms of decimal numbers in the range [-1, 1], and then multiply that sum by 127.[71]

[70] Not recommended. You should make a convention and stick to it.

[71] In Damien Robotics, this is atypical.

PID MATHEMATICS

So the whole of PID can be expressed in mathematics like this (disregarding limits on the integral):

$$error = target - currentValue$$

$$response = (k_P \times error) + \left(k_i \times \int error\ dt\right) + \left(k_d \times \frac{d}{dt}(error)\right)$$ [72]

which is also really this, as we have seen earlier:

$$response = (P) + (I) + (D)$$

This is the complete theoretical representation of PID. This is a "continuous" representation of PID as it changes over time, some authorities prefer to use a "discrete" representation of PID – the state of PID at one particular moment of time.[73]

Different PID authors have different conventions on the variable notations for the PID quantities and how to write the "sum", but we can just leave PID like this, with sloppy notation, since we don't care about the math, but only about how it works.

I prefer to write the PID sum as a function of error and time:

$$u = k_P\mathcal{E} + k_I \int_0^t \mathcal{E}\ dt + k_D \frac{d\mathcal{E}}{dt}$$

where $\mathcal{E}$ denotes error and u denotes the system response ("control" response). The definite integral represents the integral term's summing of error values as time goes on from time "zero", when the system starts.

[72] Such a formula for a feedback controller's action is usually known as a **control law**.

[73] This sort of approach – see where the system is, and what will happen next – is how most academic PID work is done – with systems of **differential equations**, equations that contain derivatives of physical variables like position.

A PID system would be represented with a set of differential equations that relate system inputs and outputs, known as a **state-space representation** or **model**.

PID IS EVERYTHING

PID encompasses everything – **the past, the present, and the future**.

- The P term depends on the present value of the error. The P term corrects for the error found in the system at any single point in time.
- By taking the integral, the I term accounts for the past values of the error. The I term corrects for the error and how it has changed over time.
- The D term is composed of a derivative of the error with respect to time. The value of this derivative indicates whether the error is increasing or decreasing as time goes on. This is a prediction of the future – it indicates how the value of the error will change in the future. For example, if the derivative is positive and very large, the error will increase dramatically as time continues. With this prediction, the D term corrects for future values of the error.

Hence why PID is love, PID is life. PID is everything.

To finish our examination of PID theory, let us return to a familiar example:

The best analogy for what PID is like and does is how we balance on two legs. Before reading about the intricacies of PID, think about how we keep our balance. What algorithm – what sort of rules – do we follow to keep upright? What does our balance intuition depend on?

- How upright we are at any moment – **This is the P term!**
- Whether the ground is tilted and we need to adjust our "equilibrium" balance angle - **This is the I term!**
- Whether we are falling or losing balance even if we are upright (e.g. keeping balanced after being pushed forward by someone) – **This is the D term!**

Although we cannot know for certain that we do indeed perform "biological" PID to keep ourselves balanced as of yet, how we judge how much we should react to keep balance correlates to the response mechanics of PID to a striking degree. In fact, according to some biology researchers, strong evidence indicates that many biological mechanisms – anything from gene expression in cells to animal behavior – depend heavily on feedback control (albeit not necessarily PID).

Now we are ready to examine the actual programming of PID in ROBOTC.

Programming PID

ALL TOGETHER NOW

The PID loop in ROBOTC can be written in just eight lines. No joke. This version demonstrates a PID controller with an error threshold limit on the integral. The PID sensor is named `mySensor`, and the motor is named `myMotor`.

```
while(true)
{
  // Calculate error
  error = targetValue - SensorValue[mySensor];

  // Calculate integral - add to the sum only if the absolute value
  // of the error is less than the integral limit
  if( abs(error) < INTEGRAL_LIMIT)
    integral += error;

  // Calculate derivative (error minus error from previous time around)
  deriv = error - lastError;

  // Set PID sum to motor
  motor[myMotor] = (K_P * error) + (K_I * integral) + (K_D * deriv);

  // Wait and set lastError value to (this time's) error
  lastError = error;

  wait1Msec(PID_WAIT_TIME);
}
```

As we see right away, this is fairly straightforward! The error is calculated, then I and D, and the PID sum is sent to the motor of interest, `myMotor`.

A PID loop could be even simpler than this, but I chose to write everything out clearly[74]. Assuming that all of those variables are defined somewhere (in the function or task where this PID is in, or just globally in the program)[75] and the motor and sensor are set up properly, this will compile and run without any issues.

---

[74] Wikipedia claims five lines at minimum for C code. However, I would rather keep everything clean and readable. It helps when debugging!

[75] In general, minimizing the number of global variables in a given program is desirable. Not using them at all is a precept preached by programmers in general. Why?

Since they exist "everywhere" they can be unintentionally modified simultaneously by two pieces of code, or suffer from other such control flow issues in longer programs. For us, this is relatively unimportant (our code is, after all, not that long), but too many global variables will clog the ROBOTC debug window. Do note that using global variables in robotics is occasionally unavoidable.

Note in this example I don't calculate a "proportional", "integral", or "derivative" variable with the constants in those terms. In that case, the sum statement would look like this: (motor = proportional + integral + derivative). I chose to plug in the constants k_p, k_i, and $k_d$at the end, on the line of the actual PID sum.

Now we will flesh this loop out line-by-line to see exactly how our PID intuition and math is translated into robot code.

The While Loop

At the very beginning there is an infinite loop. OK, it will run forever. Or until your robot is shut off, runs out of power, or takes over the world.

But this is good for PID – PID is supposed to run constantly, anyhow. Unless you need to shut your PID loop off for some reason, in which case doing so is trivial. Most Damien Robotics programmers would just throw that PID loop into its own task, like this

```
task myPID()
{
  while(true)
  {
    // Calculate error
    error = targetValue - SensorValue[mySensor];

    // Calculate integral - add to the sum only if the absolute value
    // of the error is less than the integral limit
    if( abs(error) < INTEGRAL_LIMIT)
      integral += error;

    // Calculate derivative (error minus error from previous time around)
    deriv = error - lastError;

    // Set PID sum to motor
    motor[myMotor] = (K_P * error) + (K_I * integral) + (K_D * deriv);

    // Wait and set lastError value to (this time's) error
    lastError = error;

    wait1Msec(PID_WAIT_TIME);
  }
}
```

and then just call task control commands from the main task or the user-control task, like this:

```
task main()
{
  // Some other stuff...
  // ...

  // If Button 8D is pressed, start PID
  if(vexRT[Btn8D] == 1)
    startTask(myPID);

  // If Button 8U is pressed, stop PID
  if(vexRT[Btn8U] == 1)
    stopTask(myPID);
}
```

This permits us to turn PID control on and off as desired. We could control a motor with and without PID: with PID, the motor would follow the target; without PID, the motor would respond to motor power commands as usual.

Anyhow, it is good that the PID is alone by itself, in its own task. PID depends heavily on that wait time (as we will see later) and cannot be interrupted by other code.

So the loop works. No issues with it running forever.

Calculating Error

Next, we calculate the error by taking the difference between the target value and the current value of our sensor – exactly as defined.

Remember to pay attention to error signage! We see in the PID sum that positive error corresponds to positive motor power. If this does not cause error (the difference between target and current sensor value, mathematically) to fall to zero, we have a problem.

Calculating the Integral

Then I calculate the integral term (without k_i, of course) by adding the error to an "integral" variable. This integral variable will accumulate successive values of the error as the PID repeats.

The error is only added if its size is less than the threshold value, which we call `INTEGRAL_LIMIT`. This is one of the ways that the integral can be capped, as we saw earlier.

Calculating the Derivative

Then I calculate the derivative term (without k_d) by taking the difference between the current error and the error from the last loop iteration – `lastError`. I call this value `deriv`.

Obviously, the value of this needs updating as the loop repeats: the error of "this" iteration will become the last error of "next" iteration. I do this after the PID sum, but it can technically be done anywhere after the calculation of the derivative in the loop, or before the error is re-calculated.

The Sum

Finally, I set my motor power equal to the sum of my three terms, k_p times error, k_i times `integral`, and k_d times `deriv`, and wait to repeat.

Technically, the PID sum should be limited to [-127, 127]. We may accomplish this with a set of if-statements, but, as aforementioned, ROBOTC will do it for us when you assign the PID sum to the motor if we don't.

The Loop Delay

Why is that wait statement at the end necessary?

Wouldn't it be best if the PID ran as fast as possible, so that updating the motor power will be as precise as possible? Well, no…

1. The Cortex can only handle so much. In terms of tasks, it does not run multiple tasks at the same time (as it appears to do). The Cortex actually runs tasks in a "round-robin" fashion where each task gets its own slice of time to run, and then the Cortex moves on to run other tasks. Well… it wouldn't be good for your robot if your PID is incessantly hogging all that processing time.

   Want to see the effects of that? Just call this inside any task. You have been warned.

   ```
   hogCPU();
   ```

2. In the short run, many of the sensors usually used with PID, especially if they depend on a sort of calculation (let's say, for example, that your system variable was velocity and you were calculating that based on encoder counts traveled divided by time) fluctuate a lot. This is because
   a. sensors aren't that accurate anyway, and
   b. we, Damien Robotics, aren't SpaceX, with the ability to autonomously land a flyback rocket booster with accuracy down to the meter on a tiny platform floating in the ocean[76] by absurdly accurate sensors and feedback control[77].

[76] See SpaceX CRS-8. The inquisitive reader should research the history of what has been termed the Damien Space Program with said space mission. I happen to have done research with the Damien Space Program and would gladly tout its merits, and its conspicuous lack of launch services.

[77] May have improved by the time this is read.

By making the PID repeat with very little time in between iterations (like 1 millisecond), the fluctuations will be comparatively worse versus a more reasonable rate (say 30 milliseconds) where the "**noise**" will not bleed into the PID calculations as much.

Optimal PID loop delay times for VEX, I have found, range from **20 milliseconds** (a useful baseline value) to 50 milliseconds. These are still very small delays (repeats of up to fifty times per second!).

A NOTE ON CODE STYLE

In this PID loop, I have opted to nest the PID in its own task, with variables that are defined locally. This is a clear way to implement a PID controller in software.

We should be aware that there are other methods to code this exact same algorithm, depending on the personal taste of the programmer. For example, some may choose to implement a PID function that takes arguments of error and the other PID variables and constants, and returns the motor response value.[78] Such programmers may then simply set the PID motor(s) equal to the this PID response function inside a loop with delay time.

This can yield extremely concise code[79]; however, I do not present such approaches here since the task-centric loop approach is easier to understand intuitively.

[78] Mr. Enright, among others. I prefer to do this too, but the simplicity of a dedicated PID task not bundled in function definitions is nice and easy to understand.

[79] In personal projects, I have written controllers for PID controllers that take up three lines.

A NOTE ON SENSORS

PID for VEX Robotics motors can be utilized with three sensors – the potentiometer, the quadrature encoder, and the integrated motor encoder (IME). Only these three sensors can measure motor movement in terms of rotation.[80] It is important to note several properties of these sensors before we proceed with our discussion on PID programming.

1. Potentiometers provide absolute position; encoders do not: A potentiometer has a fixed reading scale – a value of 3000 always indicates the same position. However, since encoders (quads and IMEs) can be cleared, they only indicate a position relative to the cleared position, which is "zero". If we wish to use an encoder for a positional PID controller, we must keep this in mind, assigning "zero" to a known physical position so that our target angles always refer to the same physical angle. Because of the fact that the same potentiometer reading always corresponds to the same physical position, we in Robotics prefer potentiometers for PID when they can be used.

2. Potentiometers have a rotation limit: Potentiometers are physically limited to about 270 degrees of rotation. Therefore, we cannot use potentiometers in systems where we need continuous rotation (e.g. chassis, flywheel, intake roller, etc.) – unless one perchance designs a slip gear that engages and disengages the potentiometer at regular intervals. If we desire to use a potentiometer with a range of motion larger than the limit, we would need to use a gear train to ensure that the potentiometer is never forced beyond that limit.

3. Potentiometers have greater precision than encoders: A potentiometer provides 4096 distinct readings (zero through 4095) in about 270 degrees of rotation. Thus, in one degree, we have approximately 15 distinct positions that a potentiometer can read. Compare this to a quadrature encoder (which has 360 counts per rotation, or 1 distinct position per degree) and IMEs (for regular torque motors, 627.2 counts per rotation, or about 2 positions per degree). The higher precision of potentiometers is evident. Since this is beneficial for PID, use them if possible.

4. Encoders have a maximum reading: Beyond a certain integer range limit, encoders will not function properly. This is usually of no concern unless the encoder is used for velocity control, where the encoder should be zeroed regularly. Encoders also have a speed limit above which erroneous positional data may be recorded: VEX claims that the quad encoder can tolerate up to 18.9 revolutions per second, or 1,133 rpm. Thus, an encoder for PID velocity control should never be placed directly on, say, the shaft of a flywheel

An important point to see here is that positional PID cannot be used with servos (since servos are already controlled with input that corresponds to rotational position) or pneumatic pistons (we have yet to see anyone try).

[80] An important note is that these sensors are not "digital" – they are not simply on, "reading", or off, "not reading" – like a button or limit switch. We need a continuous range of sensor values for PID to operate.

Also note that we can only use PID in VEX Robotics with motors. It's theoretically possible to control a pneumatic piston with a PID loop, but why would you ever need to?

PID IN USER-CONTROL

The first step in adapting the PID loop to control a robotics system in user-control code is realizing that this PID loop is essentially taking the target value and moving the motor there.

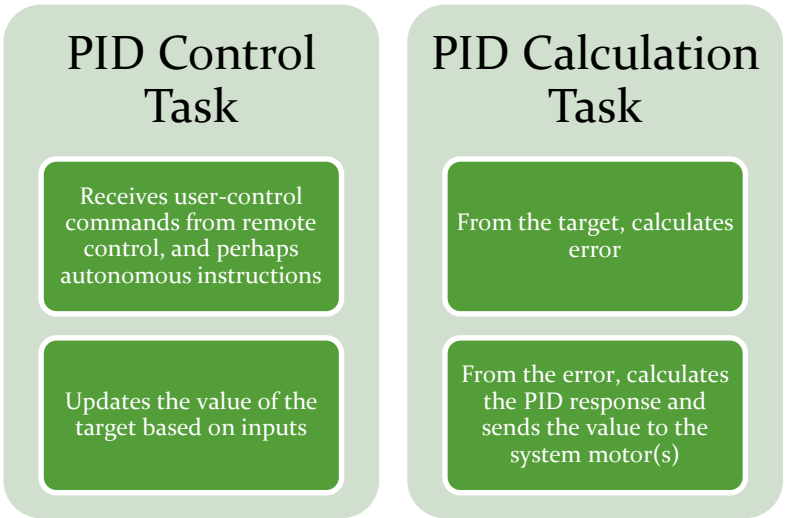
So to control a system with a PID controller, *we must change the target value with user-control code*. The system will follow the target value when it changes, and resist attempt to move the system away from the target. Instead of programming in terms of motor power, we now must program in terms of "where' we want our system to be – in terms of position (most common)[81], velocity, or something else.

But the target value needs to be set and changed somehow – and this is what user-control PID code must do outside of the PID loop.

Before we continue we should be aware that there are different approaches of performing user-control PID, just as the PID loop itself can be implemented in many ways. I merely present my own, which establishes a "target-centric" view – the target value connects the user-control part to the automated PID calculation part.

Because the target value must be changed somehow, we need two loops – the PID sum loop, which calculates error and the PID sum, and sends the result to a motor, and a loop for user-control input that change the target value. I like to term the former the **calculation loop** (or **calculation task** since the PID sum loop is usually in its own task) and the latter the **controlling loop**.

In theory we could have one loop – read changes in the target commanded by the user, and run the PID – but I find the flexibility and simplicity of two loops, in separate tasks, preferable.



[81] By now you can probably tell that most PID loops that we use in Robotics are positional PID controllers. Why? Perhaps it's because of the irresistible awesomeness of using a motor like a servo. Or perhaps the applications of positional PID are simply more common.

Note that this approach requires a global variable for the target to be accessible to both tasks.[82]

Others handle user-control differently from this method. I advocate this particular implementation of PID in user-control – two tasks united by a target value variable – owing to its clear division of objectives (one task handles user input, the other the automated PID calculations) and the fact that the target directs the PID system (PID is, after all, a means for the robot system to reach the target).

Before we dive deeper into user-control PID, I will explain the concept of **presets**. Since PID permits control based on a simple "go here" command (i.e. the PID follows the target value, so the system can be controlled by changing the target value), we can code remote control button presses to set the target value to various pre-set values. So, at the touch of a button, a lift could be commanded to go to a certain position, flywheels can be set to a certain speed, and so on. Even though the code is deceptively simple – if button pressed, target equals constant number –presets have become invaluable in Robotics.[83]

Now we will explore how specific robot system movement correlates to changes in the target value that the controlling task can make.

As with other user-control tasks, commands in the PID controlling task must reside in an infinite loop. These commands might adjust the target value in different ways, as suitable for different control schemes.

In the following table I outline various types of control input that may be desired, with examples, and show how the target value should be changed.

[82] Although it is good programming practice to avoid global variables in our program, in this case, the global variable is unavoidable. This is because the target must be accessed by two tasks, and variables in any task are invisible to others. If you know pointers, you could use one instead. (Protip: Don't take a computer science class that presupposes C knowledge without an understanding of exactly how pointers work. That happens frequently at Caltech.) Memory is memory. Memory is like a box of chocolates – ordered yet random, and you can't ever have enough.

[83] In Nothing But Net, the PID-equipped teams utilized presets to position their flywheels at a particular angle value that corresponded to a physical angle that guaranteed accuracy from a certain location on the field. Thus the driver could simply drive to the proper field location, hit the preset, and score balls. I used presets with both positional and velocity PID for my flywheels; at a single button press, both the angle and the speed of my flywheels were set with PID.

The code is really that simple.

CONTROL INPUT	ACTION ON TARGET
Buttons or joystick – "incremental adjustment" - e.g. up/down buttons for a lift (position) - e.g. step up/step down flywheel speed (velocity) with button presses	Add to or subtract from target in increments, once per task loop. Add a debouncer (prevents multiple readings for one press of a button) if desired.
Joystick – "fixed adjustment" - joystick position directly corresponds to target - e.g. joystick position correlates with flywheel speed directly (velocity), a la RC helicopter throttle - e.g. Ackerman steering[84] based on a joystick (position)	Scale the joystick values from [-127, 127] to desired target value range, and set the target value equal to that scaled value
Preset - e.g. a single button press moves a lift to a certain height (position) - e.g. an autonomous behavior, like raising a lift to a certain height in response to a predetermined behavior, such as a full tray of scoring objects (position)	Directly set the target value equal to a predetermined preset value

The reasoning behind why the target should be modified as it is in each case is not difficult to see. Of course, our PID control task must include code to detect when we have control input – if-statements.

For example, a barebones PID control task for a lift system controlled by `Btn6U` (up) and `Btn6D` (down), might look like this.

```
task PidControl
{
  while(true)
  {
    if( vexRT[Btn6U] == 1 )
      target += 5;
    else if( vexRT[Btn6D] == 1 )
      target -= 5;

    wait1Msec(20);
  }
}
```

Notice that this control task and the PID calculation task (which we are familiar with by now) must be running simultaneously.

[84] Steering by rotating the alignment of the front wheels of a vehicle. Drive systems, or chassis, in robotics that implement Ackerman steering are also known as car drives, for obvious reasons. They are usually not used in competition, for obvious reasons.

Of course, the increment amount (which I like to call **delta-target**, the change in the target per loop) and the loop delay can vary. The values of these determine how quickly the target can change over time – or how quickly our PID tries to move. The larger the delta-target, the larger the change in the target value per given time. The smaller the delay time, the more loop iterations can run in some given time, and the target can change more.

We are, however, missing one critical element before we have properly functioning PID. Notice that we have no idea what the value of `target` is when this code begins to run (presumably with the PID calculation task as well). We cannot initialize it to zero – otherwise, our lift will immediately go to zero – wherever "zero" to the potentiometer is – on start-up or whenever the robot goes into user-control mode. Not good.

To solve this, we may add a statement before the infinite loop that immediately initializes `target` to the current sensor value. Thus the error when PID begins to run is zero, and our PID response is zero – the system does not move. This prevents any unintentional surprises when the PID starts.

Suppose that our PID potentiometer is in analog port 1; we have, then

```
task PidControl
{
  target = SensorValue[in1];

  while(true)
  {
    if( vexRT[Btn6U] == 1 )
      target += 5;
    else if( vexRT[Btn6D] == 1 )
      target -= 5;

    wait1Msec(20);
  }
}
```

Assuming that `target` is passed to the PID calculation task and the system signage is correct, this system will be controllable by the remote buttons. Pressing `Btn6U` increments the target as the loop repeats. In response, the PID follows the target, raising the lift up. The opposite occurs if `Btn6D` is pressed. We can add additional if-statements to our code to add presets and other functions if we wish. The value of delta-target (5), in addition to the loop delay (20 ms), determine how quickly the target is permitted to change.

There is an additional concern with delta-target. It is evident that with this incrementing of the target, the target is basically permitted to approach any value. If the user holds down `Btn6U`, the target value can become infinitely positive.[85] This is frequently undesirable for positional PID since arms and other mechanisms usually have limited freedom of movement, and also for velocity PID, since motor speed cannot exceed the speed of light.[86] If this occurs, the PID will be stuck attempting to reach an unapproachable target. We can prevent the target from growing to such unmanageable levels with a simple if-statement threshold limit (as we have done with the PID integral and response) after whatever changes we make to the target in the loop. Simply check for excessively large and small target values, and set these to the max and min permissible.

[85] Or as large as the data type of the target variable in ROBOTC permits.

[86] Or somewhere around 100 rpm, whichever speed limit you prefer.

This example setup would be different if an encoder were used. Since encoders do not track absolute position, we can start the lift at any position we wish, set the encoder to zero, and then start the PID with a target value of zero. Since the error is zero, the lift does not move. However, with such a position-based system it is often preferable to add some method of ensuring that "zero" always corresponds to the same physical lift position e.g. always starting the lift in the same physical place.

PID IN AUTONOMOUS

Simply start the task containing the PID loop, set the target value in your autonomous routine (or some other task), and enjoy. Don't forget to account for the time it takes for the PID to reach the desired target when doing other things like opening and closing claws, reeling roller intakes, etc.

However, there is something besides this to note. In autonomous, there is a clearly defined flow of control – while a user-control routine involves code for all the different robot systems (e.g. drive, lift, manipulator) running simultaneously, an autonomous routine is linear.

Frequently it becomes complicated to maintain control of both the PID task and the controlling task (i.e. the main set of instructions in the autonomous that modifies the PID) when a PID controller must be run intermittently. It can become quite a hassle to keep track of when the PID task should be turned on and off, especially if multiple sets of PID systems are running at once. This is not a problem with PID controllers that should be "always on" (e.g. flywheel PID, lift position PID), but may be a formidable one for complicated systems like a chassis (which often requires control by multiple PID routines, one after the other, for different types of movements in autonomous).

To ease this problem, let us consider if we can fit the PID loop in the actual flow of commands that forms the backbone of autonomous, instead of a separate task that runs along with autonomous. It turns out we can:

```
task autonomous()
{
  // ... Other commands ...


  // PID begins here

  int target = 3000;

  /* Loop and carry out PID calculation while the magnitude
  of the error (absolute value of target - sensor value)
  is > some threshold value (50)

  Once the error is smaller than the threshold, the PID exits */
  while( abs(target - SensorValue[dgtl1]) > 50 )
  {
    // PID here
  }


  // ... Other commands ...
}
```

This sort of while loop is termed a **sentinel loop**, since its condition behaves as a sentinel. Once the condition becomes false (the error is within the threshold), the PID loop exits and autonomous code following the PID executes.

We need to take the absolute value of error since the error might be approaching zero from the negative direction.

Of course, we could use an `error` variable directly in the sentinel loop comparison. The vital aspect of this sentinel loop is that the condition must eventually become false (for the loop to exit). Therefore, we need some threshold to guarantee that our PID exits – rather than exiting when error is zero (since error might not always reach zero exactly), exit when the absolute value of the error (the size of the error) is less than some threshold value. We could tune this threshold value later.

The threshold value is critical – we must choose its value carefully to make sure that the loop will terminate even in the worst case – perhaps something blocks the robot so that it cannot ever reach near-zero error. We must also update the error, or some other "sentinel" quantity, within the loop. Otherwise we are stuck in an infinite loop.[87]

We could base our sentinel loop on time as well, commanding the loop to terminate once a certain amount of time has passed. This is possible with the timers in ROBOTC.

[87] I learned this the hard way. To tackle Programming Skills for Nothing But Net, I programmed my robot (one of the 6526D-57 Series) to execute a side-strafe maneuver across the field with precision so that our flywheels, having exhausted the match-loads on one side of the field, could position itself to fire the remainder on the other side. 6526F did this by dead-reckoning trial and error.

I decided to use sentinel-loop PID autonomous functions to accomplish this (they may be found in the 57-series code, from 5702 onwards). However, I did not tune the threshold perfectly. This, combined with less-than-optimal P controllers, caused my autonomous routine to stall out, stuck in the infinite loop, before it could complete the maneuver on several occasions. Why? The error was too small for the motors to push the error even smaller, so that the loop could exit.

There are ways to avoid this, however, such as an escape condition based on a timer (terminate the loop after a long time has passed regardless of the error value), or an escape condition based on when the PID response becomes smaller than the motor movement threshold (around 11 to 15, in terms of raw motor power for a single motor, depending on the load that is driven).

Multiple Motors and PID Controllers

Frequently (almost always) you will need to use PID with multiple motors. How exactly you deal with multiple-motor systems depends on the physical qualities of the system.

MOTORS GEARED TOGETHER

Simple. Just assign the same PID response to all motors. Why wouldn't you? Just make sure they're all spinning so that they won't fight each other.

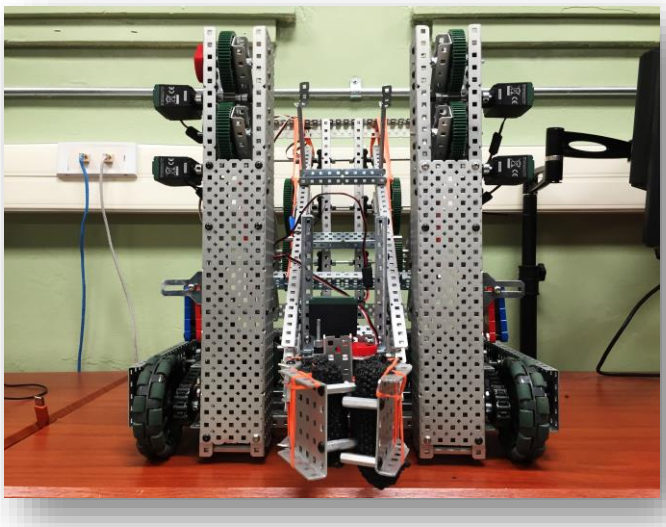
This could also be done by a handy ROBOTC command that "slaves" a motor to a "master" motor. Whatever power is assigned to the master motor will be assigned to the slave motor as well. The syntax is, quite simply,

```
slaveMotor(slave, master);
```

INDEPENDENT SYSTEMS

In many cases we will have independent robotics systems that need to be synchronized. The most frequent case of this is symmetrical mechanisms: chassis and lifts. These systems are either wholly independent or somewhat independent.

An example of the latter is a robot lift comprising two "lift tower" motor assemblies, such as 6526D-5603A:



Notice how even though both sides of the 5603A linkage are connected in various locations near the rear, the two are not linked in the same gear-train. Because of this, the two sides of the lift might move to different positions, causing the lift to tilt. This issue is more problematic with wide lifts that aren't sturdily built and supported. It is also especially detrimental to long, narrow arms – such as this one.

It is desirable to synchronize position control of both sides of such a lift system so that swaying does not occur.[88] The two motors on each side of the chassis can be given the same PID response. But how do we ensure that the target position on both sides is the same? In other words, how do we synchronize the system?

Such synchronization could be accomplished in a variety of ways, among which are

- Physically connect the systems securely (when practical, of course). Notice how this was not possible in 5603A, as any cross-bars on the linkage would prevent the lift from retracting further.
- Use two completely independent PID loops, with separate target values that are given to the PID controllers of each individual sub-system.
- Make the PID controller one side a "slave" to the other by having its target value be the sensor value of the other side.

In either of the latter two options, multiple PID sensors are required, one for each sub-system. (Why?)

Of course, in some systems (such as a chassis) it is not desirable to always have the two sub-systems synchronized. You must then use independent PID loops; one per sub-system.[89]

[88] As evidenced by the single potentiometer, this did not happen for 5603A.

[89] Simple yet complete autonomous control of a chassis by PID is a problem that remains yet unsolved by Damien Robotics. There have been different approaches (mine, Mr. Enright's, etc.), but there is no standard algorithm to, say, get your robot to coordinate positions (x, y) (perhaps in meters) at an angle (orientation) of θ (perhaps in degrees relative to the starting orientation) autonomously.

PROBLEMS

1. **(Introduction to PID)** Write an autonomous PID routine to move a robot forward for ten meters, fulfilling the dilemma that we started with. (You may scale down the distance if you wish). **Use a sentinel loop – that is, your PID loop must terminate successfully after the robot has finished moving (in contrast with previous exercises). Do not finish inside an infinite loop.** How consistent can you make the loop?

2. **(Introduction to PID)** Most likely (unless you are an apt programmer, or went ahead) you will have unintentionally forced the robot into an infinite loop at some point during Exercise 1 by mistake. Implement some failsafes for Exercise 1. Then try to make it fail.

3. (*Robot Control*) Build a robot arm with an attached potentiometer. Program manual raise/lower controls on a pair of remote buttons. Write an algorithm that will, upon the press of a remote button, record how the arm is moved by user-control (or even just by moving it around by hand) for ten seconds. At the end of the ten seconds, pressing another button will cause the robot to re-play the motion. (Hint: Take sensor data at equal time intervals of, say, 20 milliseconds. Now use PID for each time where data was taken. This is easier to do with a servo, since no PID is required.).

4. (*Robot Control*) Using a gyroscopic sensor, write an autonomous function that turns a robot chassis by 180 degrees precisely. Use function parameters to specify the maximum allowed motor power, in addition to the direction.

Sample PID Programs

If you are reading this, you are perhaps aware of the existence of a large repository of robotics code that I compiled in the summer of 2016, available here.

https://www.dropbox.com/sh/ok6torfk6chelei/AAB4UcfPSXBuP8a5p8NTIcyJa?dl=0

It is a Dropbox folder. The directory of note for us is `\Code\PID Programs`. Please request access by contacting me or the incumbent Robotics president.

A VERY SIMPLE SAMPLE PID

I wrote this in the summer of 2015 as a demonstration code for basic PID. It's simple, clear, and uncluttered. This sample program will move an arm (motor with encoder attached) for 400 encoder counts (since our encoder should be zeroed on program start, our target of 400 implies desired movement by 400 encoder counts in the positive direction).

The Motor and Sensor Setup `#pragma` directives are not shown for brevity. The full program may be found in the robotics repository. Here, a PD controller is in action (since the integral gain is zero). The PID constants here are typical values

Note: Here, `task main` is the PID task. Normally – if you want your robot to do anything other than PID, including basic driving or even modifying the target value of its own PID – the PID loop should not be in the main task, as we have seen. Otherwise your robot can't do anything else but PID!

```
// Constants
const float kP = 0.3;               // Proportional constant
const float kI = 0.0;               // Integral constant
const float kD = 0.3;               // Derivative constant

float targetValue = 400;            // Target value
// The target value doesn't change in this sample code. To control a PID
// system, simply change the target value in a different task.

float errorValue;                   // Error value (target value - current value)
float lastErrorValue;               // The previous error value (one calculation ago)

float PIDIntegral;                  // Variables to store the integral and the derivative
float PIDDerivative;

#define PID_INTEGRAL_LIMIT  (50)    // Integral limit. The I term is only added when
//                                      the error is less than this

#define T_PID               (10)    // The period (time for refresh) of the speed
//                                     calculation, in milliseconds
```

```
task main()
{
  lastErrorValue = 0;             // Initialize the previous error value at 0, since there is
  //                                 no previous error value at startup

  while(true)
  {
    errorValue = targetValue - SensorValue(ENC);    // Calculate the error value

    if(abs(errorValue) < PID_INTEGRAL_LIMIT)        // If the error is less than the integral limit
      PIDIntegral = PIDIntegral + errorValue;       // The integral is the sum of the errors
    else
      PIDIntegral = 0;                              // Otherwise the integral is zero

    PIDDerivative = errorValue - lastErrorValue;    // Find the derivative (difference)

    // Calculate and assign motor speed
    motor[Arm] = (kP * errorValue) + (kI * PIDIntegral) + (kD * PIDDerivative);

    lastErrorValue = errorValue;                    // Set the last error equal to the error, since the
    //                                              // calculation will refresh

    wait1Msec(T_PID);                               // Wait to refresh PID calculation
  }
}
```

A VERY SIMPLE SAMPLE PID: USER-CONTROL

Modifying this program for user-control is surprisingly simple. All that's needed is a second task that controls the value of the target based on remote control, like what we have covered in the user-control PID section. I have also moved the PID itself to its own task.

I have included provisions for remote control (up/down buttons) in addition to buttons that command the PID to two different presets. As before, the Motor and Sensor Setup is not shown.

```
// Tasks
task PIDControl();                  // Target value control task
task PID();                         // The PID task


// Definitions (examples)
// The keyword "#define" can be used to declare CONSTANTS or anything
// that can be substituted into the code.

#define PRESET_1            1000
#define PRESET_2            2000

// ##### Remote controls #####

// Manual up/down controls
#define UP_BTN              Btn5U
#define DOWN_BTN            Btn5D

// Preset buttons
#define PRESET_1_BTN        Btn7U
#define PRESET_2_BTN        Btn7D
```

```
// ##### Constants #####

const float kP = 0.3;               // Proportional constant
const float kI = 0.0;               // Integral constant
const float kD = 0.3;               // Derivative constant

#define PID_INTEGRAL_LIMIT  (50)    // Integral limit. The I term is only added when
//                                     the error is less than this

#define T_PID               (10)    // The period (time for refresh) of the speed
//                                     calculation, in milliseconds


// ##### Variables #####

float targetValue;                  // Target value

float errorValue;                   // Error value (target value - current value)
float lastErrorValue;               // The previous error value (one calculation ago)

float PIDIntegral;                  // Variables to store the integral and the derivative
float PIDDerivative;

task main()
{
  // Start the PID control task
  startTask(PIDControl);
}


task PIDControl()
{
  // Initialize the target value at the current pot reading so that the
  // arm doesn't go anywhere when the robot turns on (error will be zero)
  targetValue = SensorValue[ArmPot];

  // Start the PID task
  startTask(PID);

  while(true)
  {
    // ***** MANUAL CONTROL *****
    // Change the target value if the manual controls are pressed. The PID
    // will follow the target value.
```

```
    // UP - ADD to target value (if upwards motion increases pot readings):
    if(vexRT[UP_BTN] == 1)
      targetValue = targetValue + 5;      // or some other constant
    // DOWN - SUBTRACT from target value
    else if(vexRT[DOWN_BTN] == 1)
      targetValue = targetValue - 5;

    // ***** PRESET CONTROL *****
    // If a preset button is pressed, set the target value equal to the
    // angle value of the preset (pot reading). The PID will move the arm there.
    else if(vexRT[PRESET_1_BTN] == 1)
      targetValue = PRESET_1;
    else if(vexRT[PRESET_2_BTN] == 1)
      targetValue = PRESET_2;

    // The else-ifs ensure that the PID can't receive two conflicting
    // commands at the same time.


    // Some brief wait time is optional here
  }
}


task PID()
{
  lastErrorValue = 0;               // Initialize the previous error value at 0, since there is
  //                                   no previous error value at startup

  while(true)
  {
    errorValue = targetValue - SensorValue[ArmPot]; // Calculate the error value

    if(abs(errorValue) < PID_INTEGRAL_LIMIT)        // If the error is less than the integral limit
      PIDIntegral = PIDIntegral + errorValue;       // The integral is the sum of errors
    else
      PIDIntegral = 0;                              // Otherwise the integral is zero

    PIDDerivative = errorValue - lastErrorValue;    // Find the derivative (difference)

    // Calculate and assign motor speed ( P + I + D )
    motor[Arm] = (kP * errorValue) + (kI * PIDIntegral) + (kD * PIDDerivative);

    lastErrorValue = errorValue;                    // Set the last error equal to the error, since the
    //                                              // calculation will refresh

    wait1Msec(T_PID);                               // Wait to refresh PID calculation
  }
}
```

Note that the target value should be limited by means of a maximum/minimum threshold (as we have mentioned earlier). This is critical to safe PID operation.

OTHER SAMPLES

Other sample programs exist in the robotics repository. Please work through the logic in the programs and reverse-engineer the PID code in them.

Tuning PID

Now the last bit of putting PID into practice: how do you find the correct values for the constants k_p, k_i, and k_d for your particular PID system? Before, we have assumed that the values of the constants should be just "right" so that the PID behaves perfectly. But how do we actually determine the numbers to plug in?

BY TRIAL AND ERROR

The most common way (in VEX, at least) of finding the gain constant values is pure trial and error. Tuning the constants of a PID loop decently will take at least an hour. I remember spending a month tuning a modified PID for the flywheel[90] on my Nothing But Net robot during my last season. Much frustration will be vented on a certain three constants, and my words cannot do much to minimize your sufferings.

There are some common, more intuitive ways to tune PID that can easily be found online. One popular method is:

1. Start with all constants equal to zero. Increase k_p as fit, to the point where the PID is about to overshoot and oscillate back and forth without end.
2. Increase k_d as fit.
3. Increase k_i if necessary.

Look online for more.

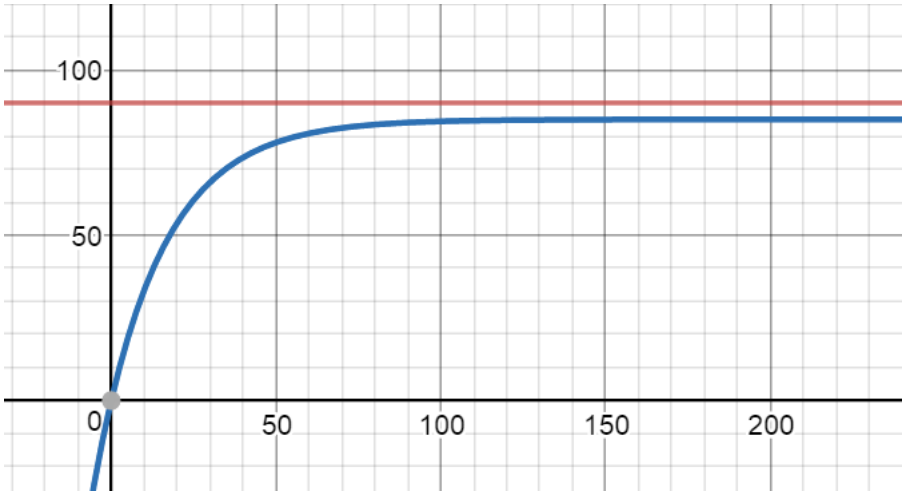
After some experience with PID you will gain a certain intuition for tuning constants. For example, suppose your PID-controlled arm needs some more oomph to resist external forces. Then you know immediately to increase k_d. You will find other, less intuitive "telltale signs" as you gain experience.

> *There is no such thing as a perfectly tuned PID controller.* It all depends on what sort of performance you desire from your PID loop.
>
> PID can be tuned to minimize time to target (ideal for arms, flywheels, and most mechanisms), minimize overshoot (error is never negative, ideal for robots that must drive precisely with PID in autonomous), some combination between the two, and so on.

[90] PID adapted for velocity control with the addition of a constant – described later – and some astonishing sensor noise reducing calculations.

BY TRIAL AND ERROR, WITH THE HELP OF A GRAPH

With some intuition on how to tune PID, you may also tune PID with the guidance of a graph of target and sensor value versus time. Consider our previous example:



These problems with the PID are apparent from the graph

1. There is considerable steady-state error. The sensor value settles at 85 where the target is 90
2. We have no information on how effective this PID resists disturbances.

The intuitive solutions:

1. Increase k_i, or loosen the limitations on the size of the integral. If there is no I term, add one.
2. Experiment with disturbing the system (giving the robot arm a good shove), see the result, and adjust k_d as fit.

But how might we get graphs like this?

One way is to record values of the target and the sensor value (and perhaps error, too) every loop iteration in the ROBOTC debug stream. This can be done with the command `writeDebugStreamLine()`. Simply write values to the stream with that command, and run your code. These values may then be pasted in a program like Microsoft Excel and graphed. Tedious.

The best method is to use the datalogger built into ROBOTC to graph the PID quantities.[91]

[91] At the time of writing, the ROBOT datalogger graph is a relatively new feature, one that Damien Robotics has yet to take much advantage of.

THE ZIEGLER-NICHOLS METHOD

There is an alternative to tedious tuning of PID by hand (Yay!). Actually… it's not perfect. But in some situations fairly useful values for k_p, k_i, and k_d can be calculated – yes, calculated. This technique was pioneered by J. G. Ziegler and N. B. Nichols during World War II.

First, we set k_i and k_d to zero (so our controller is just a P controller). Now increase k_p until your system starts "bouncing" or oscillating back and forth (without leveling out after a while). Now decrease k_p from that value until our k_p value is the smallest possible one where regular, perpetual oscillations will occur.

This value of k_p is known as the **ultimate gain** (k_u) of your system. Write it down!

Next, we need to find the **period** (denoted T) of those oscillations. The period of some repeating event is the number of seconds it takes for the event to repeat. It's also used to describe waves – the period of a wave is the time between successive peaks of a wave.[92] Well, it's likely that our system is bouncing too fast for us to obtain accurate period measurements. We can now either

1. wing it, and estimate, or
2. get something to measure the period more accurately (hmmmm, a testbed with a limit switch, a Cortex, and the ROBOTC debug stream…; If we have an arm, we might find out how fast our arm changes direction by using the derivative…; or the ROBOTC datalogger graph…).

The number we get is called the **ultimate period** (T_u) of our system.

Now we just calculate our gain constants with the formulas from this table, depending on what type of controller we desire. Use seconds for the period:

Control Type	K_p	K_i	K_d
P	$0.5K_u$	-	-
PI	$0.45K_u$	$1.2K_p / T_u$	-
PD	$0.8K_u$	-	$K_p T_u / 8$
PID	$0.6K_u$	$2K_p / T_u$	$K_p T_u / 8$

[93]

[92] Closely related to the period of something is the frequency – the rate at which something occurs. In fact, it's the reciprocal of the period ($1/T$). Frequency technically has no unit (it's the number of times that something happens in one second) but is usually expressed in Hertz (Hz), which is a unit for such. There is also the inverse second…

[93] From Ziegler, J.G and Nichols, N. B. "Optimum settings for automatic controllers" (1942)

Then we plug in the constants into our code, and we're done!

Now, as I mentioned before, Ziegler-Nichols isn't perfect. The resulting PID is aggressively tuned; it will overshoot before reaching its target value (in essence, it sacrifices precision – no overshoot – for speed – reaching the target quicker). It might not be suitable for your application. But the method is useful when you need such a controller.

There are other tuning tables online that aim to provide PID tuning for different system characteristics based on k_u and T_u. Use these if you wish.

OTHER TUNING METHODS

There are other formula-type methods beyond Ziegler-Nichols. There is even software online that could do much of the tuning for you. But these are probably all proprietary ($$$$$, not just $, too, since such software is usually made for industry use) and not suited for VEX or robotics in general.

I have used an automatic PID tuner in Simulink, a simulation program compatible with (and made by MathWorks, the makers of) MATLAB, a programming language commonly used in industrial and scientific applications. To my knowledge MATLAB and Simulink are offered for free to VEX Robotics teams as a programming resource. In the future we should explore this resource.

To date, Robotics has tuned its PID loops by hand and endless repetition. I wish you luck in your first endeavor. I wish you success in learning the arcane art of tuning PID based on observation and sheer intuition.

A last key point to remember is that PID tuning is all about tradeoffs (as you will discover). Suppose that we were trying to tune our autonomous robot arm PID so that it will rotate 90 degrees. If we used large values for k_p and k_d, we would end up with a very aggressive controller that reaches the target very quickly – but also overshoots and spends some time bouncing back and forth, settling down. If we used small values for k_p, we would get a slow controller that gradually approaches the target – but doesn't overshoot. If we tried to find a happy medium, we would obtain a controller that approaches the target in less time, but overshoots somewhat. No PID will get you instant response with no overshoot – and neither will anything else, because of the laws of physics. So in tuning PID we seek to balance response time with overshoot in order to get qualities that are best for our system.

The Wikipedia article on PID[94] provides a summary of what we at Robotics have found from experience.

Effects of *increasing* a parameter independently[18]

Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability[14]
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

[94] The table is from Zhong, Jinghua. "PID Controller Tuning: A Short Tutorial" (2006)

Expanding PID

PID in its base form is great. However, PID can easily be improved, and some applications of PID require the P + I + D sum to be modified slightly, or even the whole algorithm to be changed.

Most notable among such applications is utilizing PID to control the velocity of a motor, instead of its position. Before we have considered examples of velocity control, but here we will proceed with a method of adapting positional PID for velocity control.

PID FOR VELOCITY CONTROL

In Robotics, PID is most frequently employed to control motor position. However, PID is also useful for regulating the velocity of a motor.

In adapting PID to control motor velocity, we must first note a critical discrepancy between position and velocity control.

Adding an Offset Constant

The PID sum suggests that, when error is zero, the PID response is zero.[95] This works for positional control, since when the error is zero, the system should not be moving, so the response should be zero so that the system doesn't move[96] – but *notice that motor power should **not** be zero when there is no error in velocity!* (Why?)

So we must add another term to the PID sum, one that, when the error is zero, sends the correct power to the motor to keep the error at zero. What is the value of this term? We see that it should be a prediction for the correct power to assign to a motor to keep it at a given velocity. Therefore, it must depend on the value of the target velocity – and not on error.

We know from our earlier assessment of motor limitations that there exists a correlation between motor power and motor velocity (true "motor speed"). This relation is, sadly, not linear.[97] But if we can quantify this relationship in an equation, we can use our target velocity to calculate a prediction for the power necessary to keep the motor running at that target velocity, and add this to the PID sum as an **offset constant**.

---

[95] Not exactly zero due to the I and D terms, but thereabouts.

[96] The I term may output some small response to keep the system from moving, resisting an external force of some sort (e.g. gravity, differences in robotic system mass from when it was tuned, etc.).

[97] It is actually somewhat logarithmic, as previously mentioned. This means that the inverse – motor power as a function of desired motor velocity – is exponential.

If there is error, the PID will correct for it with the usual three terms. If error is small, the system response will settle around the offset constant, providing the backbone necessary to keep the system running at the desired velocity.

> Some authorities prefer to not add such an offset constant and adapt PID so that the integral provides the response necessary to keep the motor moving at zero error. I opine that this is unwise, as it defeats the purpose of the integral term entirely by saturating the integral.

[98]

Remember that the value of the offset constant must be negative if the desired velocity is negative ("backwards" motion).

Calculating Motor Velocity

There is another issue at hand in adapting PID for velocity control – no VEX sensor directly measures the velocity of a motor! We must therefore calculate velocity and use such as our PID process variable. We can calculate this velocity from any encoder[99] we wish – it does not have to be attached directly to the PID system motor(s), only to the system that the motor(s) drive.

There may be an exception to the need to calculate velocity – in modern ROBOTC, there exists a function for IMEs called `getMotorVelocity()` … hmmmmm…[100]

Without this function, our only recourse is positional data from encoders. We know that the change of (rotational) position over a change in time is (rotational) velocity – or, alternatively, that the derivative of rotational position with respect to time is rotational velocity – and that we can approximate the derivative with a difference quotient (ye olde slope formula). So we can do something similar to the D term with our encoder readings over time:

$$\frac{d(readings)}{dt} \approx \frac{\Delta readings}{\Delta t} = \frac{reading_2 - reading_1}{t_2 - t_1} \rightarrow \frac{reading - lastReading}{\Delta t}$$

If we implement this in a loop with a set delay time Δt, the result can be stored as a variable whose value approximates the angular, or rotational, velocity.

Unlike the D term, the time interval Δt in the denominator is now ***not*** negligible, since we are concerned with the actual value of the difference quotient (and not just an arbitrary value that the derivative gain can scale easily). We cannot remove the denominator and preserve the meaning of the calculation. The quantity

---

[98] What we are developing here is not "the" velocity PID controller. There is no definitive velocity controller commonly used in VRC.

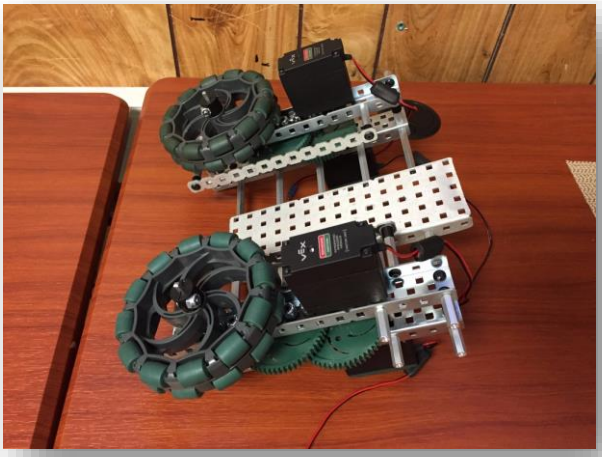
[99] We are concerned with encoders only since only encoders can be used to track continuous rotation.

[100] This did not exist for the majority of the VRC Nothing But Net season. Therefore, I wrote my own velocity calculation code, utilizing encoders.

$$\frac{reading - lastReading}{\Delta t}$$

will have a physical meaning: rotational velocity in *encoder counts* (unit of the readings) *per microsecond* (or whatever unit the delay time is in). With appropriate conversion factors, we can convert this to any units we wish, and then use it as our process value. I prefer to use RPM since the maximum velocity of a motor is nominally 100 RPM.[101]

In conclusion, we should note that this calculation will be incredibly noise-prone (i.e. much fluctuation in the calculation around the true motor velocity). Some algorithms exist that can reduce the noise significantly and thus yield a much better PID loop.[102]



A VRC Nothing But Net flywheel system prototype that I built in 2015, comprising two independent gear-trains of two motors driving two wheels with a 1 : 25 (12 : 60 compounded with 12 : 60) gear ratio. Ideally, the system is capable of more than 2,500 RPM. Flywheels are the archetypal velocity PID example. Notice the IMEs on both sides of the system, which would have been used for velocity PID control.[103]

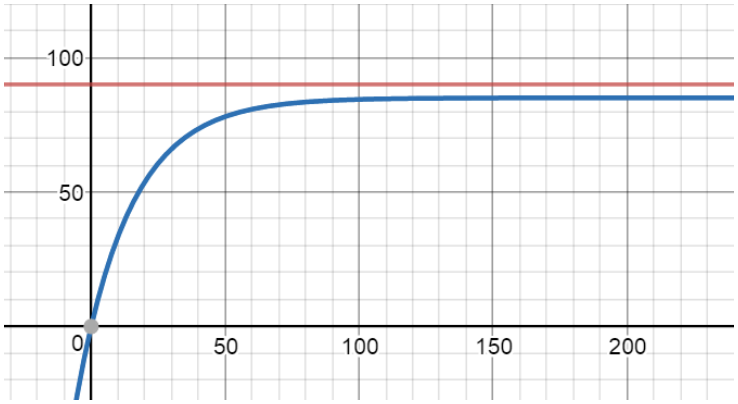
[101] VEX 393s are actually capable of about 110 RPM without any load attached.

[102] Averages of the velocity over time; weighted averaging, exponential weighted averages; etc. I used an exponential weighted average in my VRC Nothing But Net code.

[103] One particular concern about rotary encoders pertinent to flywheels that I would like to address is that they have a maximum rotational velocity beyond which they cannot measure position accurately any more. This limit is approximately 1000 RPM. Thus an encoder should never be mounted directly on a flywheel. (Also, the friction in the encoder will be amplified by the gear ratio back into the motor(s), slowing them down), I have had success running quadrature encoders at 500 RPM or so (see 6526D-5703 and beyond).

OFFSET CONSTANT APPLICATIONS

Occasionally other PID loops beyond velocity control require an offset constant. Consider the residual error problem presented when we were introducing the integral term:



The PID (in that case, it was a P controller) levels out, unable to reach the target, because the motor power response is too low to move the motor. Rather than fixing this with the rest of PID (namely, the integral term), we could add an offset constant that ensures no matter how small the error, the PID response can move the motor.

That is,

- If the error is positive, add the maximum motor power such that our system will not move to the PID sum (which is really only the P term in this case).
- If the error is negative, subtract the maximum motor power such that our system will not move from the PID sum.

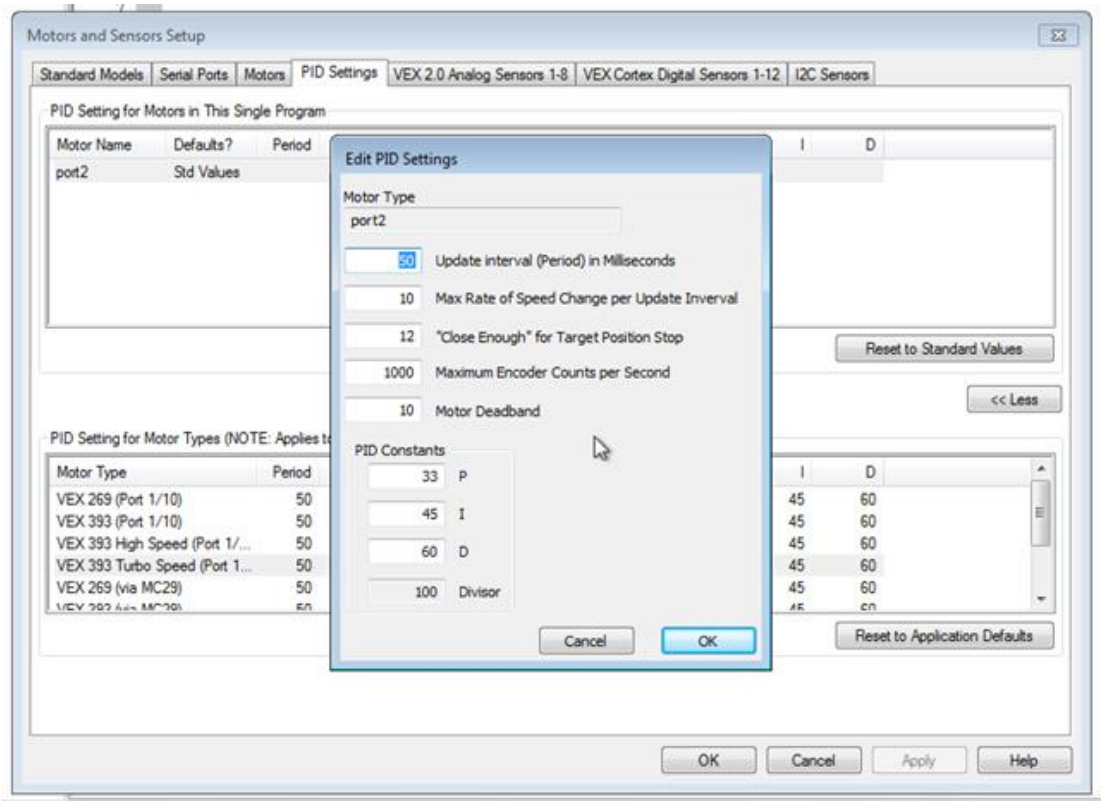
With this offset constant, the P controller will no longer peter out before reaching the target. Such application of an offset constant could be useful in simple autonomous chassis P controllers.[104]

[104] I utilized this in my VRC Nothing But Net code. The autonomous program had about six different PID controllers – sentinel loops – for different maneuvers, such as "go forward for some distance at some maximum speed", "side-strafe for some distance at some maximum speed", or "turn for some number of degrees at some maximum speed". A different approach is to have different positional PID loops – one for each wheel (for a tank drive, only two are necessary, one for each side of the chassis; but for an x-drive, four independent loops are needed) – and assign target values to each wheel according to the maneuver needed.

THE BUILT-IN PID IN ROBOTC

There is PID controller functionality native in ROBOTC. It requires the use of an encoder, and can be enabled in the Motor and Sensor Setup Dialog with a motor that is linked to an encoder.

There exists a menu in the dialog particular to the PID:



However, since the particular PID algorithm used is proprietary (and Robomatter Inc.[105] does not provide us with their source code, obviously), I discourage use of the ROBOTC PID, since we cannot even modify this PID algorithm directly to suit our own needs. Also, the intuition we gain from tuning PID loops over time – gut feelings on about how large the values of the gain constants should be – will be incompatible with the custom PID that we have developed.

The built-in PID is reportedly capable of velocity control.[106] If utilized, this may be a far more efficient (i.e. space-saving, and easier to wield) method for velocity control than custom PID loops.

[105] The makers of ROBOTC.

[106] To me it appears that the built-in PID is only intended to be a velocity controller.

Limitations of PID

As great PID appears (and really is), PID can suffer from some serious drawbacks based on what you want to do with it.

> The bottom line or "disclaimer" behind all PID in VEX Robotics is, of course, that any PID with VEX equipment is limited by the quality of VEX equipment.
>
> Positional PID can hold an arm up against gravity, but it won't make it magically overheat-proof. Velocity PID can get a flywheel spinning to within half an RPM away from its target speed or better, but the accuracy is inevitably limited by sensor noise (fluctuations in the accuracy of the sensor) and the reality of physics (friction, air resistance…).

The PID algorithm itself also has its own limitations. PID is not great for controlling systems that receive large, sudden disturbances regularly. PID is also fundamentally limited in that it is a feedback controller – it observes its system (whether there is error) and then responds to it. There is an inherent lag in PID because of this – PID can only respond to disturbances, or commands to move, "after the fact" – after there has already been error.

PID alone is also never a magical solution to any robot control problem. You need to understand what you are doing beyond PID: you must comprehend how your entire code works in concert with the PID loop to get your mechanism where you want it. There might also be other factors at play that need to be accounted for. Suppose that you must deal with noisy (i.e. inaccurate, fluctuating) data on motor speed. You might need to code some sort of filter, something to average out the data, in order to ensure that PID runs smoothly.

ALTERNATIVE FEEDBACK CONTROLLERS

There are various improvements on PID[107] out there, and a large body of technical literature on designing custom control systems. There are a variety of feedback controllers that have been adapted for VEX Robotics (as seen in the VEX Forums); the following describe some of the most notable from the realm of VRC and when they might be superior to a PID loop

Take-Back-Half

As mentioned previously, Take-Back-Half is an integration-based controller that was popular in the realm of flywheel velocity control for VRC Nothing But Net. Besides the integration gain constant, Take-Back-Half limits the response size by averaging out response values over time by halving the sum of the current response value and the average of all previous response values, (and then using this halved sum as the motor power) hence the name. This controller is commonly used in air conditioning applications. Take-Back-Half, to my knowledge, has not been applied to motor position control.

The particular algorithm may be found in old VEX Forum posts; my 57-series competition code (5701 v.5.X.X.X) contains some modified variants.

[107] Some of the most notable include **feedforward**– adding a behavior to PID responding to disturbances with a mathematical prediction or model before the disturbances can affect the system, rather than with the PID feedback-based sum alone – and **cascade control** – PID loops nested inside PID loops, where the response of outer loops become the target value of inner loops.

"Bang-Bang"

```
while(true)
{
  if( error > 0 )
    motor[myMotor] = 127;
  else
    motor[myMotor] = 0;
}
```

Need I say more?[108]

Bang-bang control is more properly termed a **hysteresis controller** (a controller that oscillates, or switches, between two states – as we have here). This controller was also a fairly popular option during the Nothing But Net season, particularly for single-flywheel machines. Although this is evidently a crude form of control in contrast to PID (the system will always "bounce" around the target), in certain circumstances bang-bang can be incredibly effective: if the system is relatively powerful torque-wise and has little mechanical "slop"[109]. It is also possible to implement corrections for some of the bad effects.[110]

Observe that the above code implements a *velocity* controller, and not a position controller. It also seems that the motor cannot run backwards…

Composite algorithms

I have had a fair amount of success by combining the principles of these common feedback controllers – bang-bang, Take-Back-Half, and PID. This is not easy to do, and whatever you choose to do should depend on our application in question. Or make your own algorithm.

---

[108] This is actually a velocity controller, and not a position controller.

[109] I mean the unavoidable play in the gears of a gear train – that some motion of the gears is possible even when the driving motor isn't moving. The less gears you have, the better a hysteresis controller might work. You should still avoid them in general. PID is patently superior.

[110] I attempted this, and succeeded, during the VRC Nothing But Net season with my flywheel controller.

One way that the stability of a bang-bang controller can be improved is implementing **hysteresis** (not to be confused with the name of bang-bang), where the switching is delayed until the error grows larger than a set threshold value. For example, a bang-bang velocity controller could be made that would apply full power until the motor overshot the target velocity by 5 RPM, then apply zero power until the velocity falls under 5 RPM below the target, then switch back to full power, and so on.

THE KALMAN FILTER

In competitive robotics, the next advancement that many teams take beyond PID is the **Kalman filter**. This is a form of feedback control[111] usually used to control robot position.

Rather than driving individual motors, a Kalman filter acts on an entire robot (the whole robot is the "system"). Kalman filters are custom-built for any particular robot, with particular sensors and mechanisms. The principle behind a Kalman filter is estimation – it allows a robot to take input from multiple, different sensors and predict its position with a higher degree of accuracy than with one sensor alone.

I will not discuss the Kalman filter because it depends heavily on linear algebra, which is beyond the scope of most high schools. But it's there, and it may be the next step for robotics control at Damien after PID.

[111] Not really – not in the sense of correcting for error.

Concluding Problems

1. (*Robot Control*) Attach a position sensor to the arm or lift mechanism of any robot (but use a competition robot if you have one, as this exercise will benefit your robot's performance greatly). Implement a PID controller system for the lift that allows it to be controlled by user-control. Utilize two remote control buttons as up/down controls, and two others as presets – pressing one sends the arm to its fully lowered position, and pressing the other sends it to its fully raised position. Ensure that the arm cannot be broken by poor driving (e.g. forcing beyond physical limits, etc.).

2. (*Robot Control*) Using three line followers, program a robot to follow a path on the floor. (Hint: use black tape on a brightly-colored floor, or vice versa). Do not use PID. See if you can improve the performance of your algorithm by taking advantage of the line follower's analog reading and PID.

3. (*Robot Control*) Using any combination of sensors and PID, program a "slave" robot to follow a user-controlled "master" around. (Hint: There is no reason to make the following robot gigantic. You may use a VEX flashlight if you wish. You may find it helpful to attach a large sheet of solid, or brightly-colored, material to the rear of the master robot).

4. (*Robot Control*) A simple maze-solving algorithm is "go forward while turning only in the same direction". If a maze is available[112], use PID to keep a robot moving at a constant distance from the maze wall while following this algorithm. (Hint: You need a non-contact distance sensor that works at range. You also need more sensors than just the one involved in the PID). If a maze isn't available, use any wall. For a slightly more exciting exercise, program a don't-fall-off-table robot with this algorithm.

5. (*Robot Control*) Make a PID velocity controller for motors. Employ it to control a robot chassis via "tank drive" dual-joystick control. (Hint: You need a fairly accurate method of finding motor velocity. You also need two encoders, one for each side.) How much difference does the presence of PID make?

6. (*Robot Control*) Research and build a "choo-choo" catapult with two VEX push-buttons attached on the side. Use PID to control the movement of the cam so that pressing one button engages the cam, cocking the arm back, and pressing another releases the catapult. (Hint: You will need to use an encoder. You might think about using some external sensor like a limit switch to clear the encoder at regular intervals).

7. (*Robot Control*) Use the ROBOTC datalogger to tune an autonomous robot arm like the example that was used in the text. (Hint: Use the graph feature and graph target and sensor value for, say, 15 seconds. Program the system so that at a time of 5 seconds after the program begins, the target value changes from zero to a set value. You will see a "step" on the target-versus-time graph, and you will see how the arm responds to this step on the sensor value graph. Tune accordingly. It's actually quite easy to do!)[113]

---

[112] If Damien hosts its annual Boe-Bot challenge, then likely yes. The first of these was a junior-high robotics competition sponsored by our very own Damien Robotics, in which Boe-Bots – small autonomous robots with a variety of sensor fittings that could be attached – competed in three events: line following, maze-solving, and "dancing" to music.

[113] The sort of system behavior that results is known as a **step response**, and this sort of tuning method is often called **bump testing.** There are actually algorithms for finding gain constant values from bump test results.

8. (Analysis) Show that any feedback controller with an integral term has zero steady-state error, no matter how small the integral gain constant is. Assume that the integral gain is not so large so that the system never settles down.

9. (*Feedback Control*) Write a bang-bang controller *for position*, for a robot arm. Try it out. Sure, it's sucky. Now you know why.

10. (*Feedback Control*) Research Take-Back-Half and compare it to velocity PID control (see "Expanding PID"). Which controller do you expect to be more "aggressive"?

11. (*Feedback Control*) Many PID implementations use the derivative of the *process variable*, and not the error, with respect to time. Is this a valid approach? Is it any better than taking the derivative of the error?

12. (*Feedback Control*) Is there any point in using a velocity controller (i.e. any kind of feedback controller) to set motor velocity to the maximum possible (a power of 127, or about 110 RPM)? Explain why or why not.

13. (*Robot Control*) Build and program a two-wheeled, self-balancing robot that can be driven by remote control.

Going Further

Many PID resources exist online, but many of these are too technical to be of much benefit for VEX Robotics. Additionally, most PID literature deals with its industrial plant control application – and not robotics.

But there are many sources online that deal with the particulars of adapting PID in robotics, especially in terms of specific applications. For example, some DIY roboticists have written on the techniques of utilizing PID to balance a two-wheeled robot (a la Segway).[114] Additionally, a great deal of insight into PID particulars has been disseminated in old VEX Forum posts, especially in the subject of designing an accurate and responsive velocity feedback controller (not just PID) for VRC Nothing But Net.

The Kalman filter as well as other, more advanced feedback controllers (i.e. design your own!) are, of course, open for those who have mastered PID.

[114] This is actually a very common problem known as the "inverted pendulum" in feedback control applications for engineering. It is related to anything from the Segway to hoverboards to keeping a rocket stable while it is flying through the air. It has been performed in Robotics with gyroscope-based PID – the code is in my repository.

I highly recommend reading <http://www.cds.caltech.edu/~murray/books/AM05/pdf/am08-complete_22Feb09.pdf> for any feedback controller needs that I cannot elucidate. This is an advanced text meant for students who have had college-level math (and no, not just AP Calculus or even differential equations), however.

About the Author

I co-founded the Damien Robotics club/program in 2013 and served as the first official program president in the 2015-2016 school year, my senior year. I have been an informal mentor to the team since graduating from Damien in 2016. I am currently an undergraduate student at Caltech studying electrical engineering and control and dynamical systems. Aside from classes, I am a 3D printer technician and an aerospace researcher, working on autonomous spacecraft simulators at the Graduate Aerospace Laboratories of Caltech (GALCIT).

I can be found at raysun@caltech.edu

1200 E. California Blvd. Pasadena, CA : MSC 919

Visits and mail will be much appreciated. Don't play The Ride.[115]

[115] **Ride, The**. (n). Richard Wagner's *Ride of the Valkyries*, which is played at extreme volume at 7:00:00 A.M. sharp on the mornings of finals week at Caltech, at least in one Hovse (sic). Also never played during any other time. While on campus, listen to at your own risk. At least use headphones.

Alternative Resources

There are other excellent VEX Robotics PID resources online, including the following:

- "Introduction to PID Controllers" by George Gilliard – a much shorter version of what this paper presented. Gilliard also explicates the concept of presets in another document on his website. If you found this text rather confusing, Gilliard takes a more intuitive approach than I do, which may be helpful for some.
    - http://georgegillard.com/documents/2-introduction-to-pid-controllers
- A sample program:
    - http://www.vexforum.com/index.php/6465-a-pid-controller-in-robotc/0
- Some helpful videos:
    - https://www.youtube.com/watch?v=7BDjZYGHupE
    - https://www.youtube.com/watch?v=DoH4t4n5J6k
- The sample code in my robotics repository
- Various VEX Forum posts. Beware that few people bother to explain PID in entirety, but rather provide links to resources.

SPONSORED BY

KERBAL PROPULSION LABORATORY

A 6526D PUBLICATION