WPID: A Simple PID Library for VEX

A Major Qualifying Project submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the Degree of Bachelor of Science

Submitted to Worcester Polytechnic Institute

Submitted by

Jair Meza

Austin Rebello

Brianna Sahagian

Advisor

Professor George T. Heineman

Date: February 29, 2024

Abstract

While VEXCode is the officially supported API for programming the V5 Brain, there is a distinct lack of beginner-friendly third-party libraries. The WPID Lib and corresponding website aims to fill this void by introducing novice teams to Proportional Integral Derivative (PID) *concepts, accompanied by a simple-to-use library written for VEXCode. Beginners can use WPID Lib to help narrow the gap with experts. With thorough documentation and text tutorials hosted on the website, students are able to quickly learn PID concepts and how to use PID on their robot, granting them the ability to increase the quality of their autonomous driving sections of the competition. After conducting our study on VEX teams who tested out our library, we confirmed that WPID Lib is an excellent resource for beginner and intermediate level VEX teams in competition.*

Acknowledgements

We would like to thank the people who made this project possible.

From the VEX Community, we would like to recognize all the teams that took the time to try out our library. We understand that the competition season is busy, but your feedback on our WPID Library was crucial to building out a robust and powerful PID tool for beginner and intermediate teams.

We would also like to thank Sarah Stanlick and Ruth McKeogh from WPI's Global School and Institutional Review Board for assisting us with taking the proper steps to conduct ethical human trials of our software.

Finally, we would also like to thank our WPI advisor, Professor George Heineman. His support and feedback was invaluable throughout our project, and helped push us to create an excellent resource for VEX competitors.

Authorship

Sections	Primary Author(s)
Abstract	Austin Rebello
Acknowledgements	Austin Rebello
1.0 Introduction	Brianna Sahagian
2.0 Background	Brianna Sahagian
2.1 The VEX Robotics Competition	Brianna Sahagian
2.2 PID Control	Brianna Sahagian
2.2.1 Open-loop and Closed-loop Control Systems	Brianna Sahagian
2.2.2 P, I, and D Constants and Configurations	Brianna Sahagian
2.2.3 PID Example	Brianna Sahagian
2.2.4 PID Areas for Improvement	Brianna Sahagian
2.3 The Current VEX Landscape	Jair Meza
2.3.1 Issues in Implementing PID	Jair Meza
2.3.2 Third Party Solutions	Jair Meza
3.0 Methodology	Jair Meza
3.1 Coding the WPID Library	Jair Meza
3.1.1 Structure and Organization	Jair Meza
3.1.2 Implementing the PID Class	Jair Meza
3.1.3 Implementing the Mechanism Class	Jair Meza
3.1.4 Implementing the Chassis Classes	Brianna Sahagian
3.1.5 Types of Motion	Jair Meza
3.1.6 Logging Capabilities	Austin Rebello

3.2 Conducting the Study on VEX Robotics Students	Jair Meza
3.2.1 Exploration	Jair Meza
3.2.2 Testing	Jair Meza
3.2.3 Interview	Jair Meza
3.2.4 Research Conclusion	Austin Rebello
3.3 Creating the Website Documentation and Tutorials	Austin Rebello
3.3.1 Website Design and Implementation	Austin Rebello
3.3.2 Documentation	Austin Rebello
3.3.3 Tutorials	Austin Rebello
3.4 Measuring Success	Austin Rebello
3.4.1 Autonomous Route Analysis	Brianna Sahagian
3.4.2 Interview Responses	Brianna Sahagian
3.4.3 Donation of Materials	Austin Rebello
4.0 Results	Austin Rebello
4.1 Robot Testing	Brianna Sahagian
4.2 VEX Student Trials	Austin Rebello
4.2.1 Successes	Austin Rebello
4.2.2 Drawbacks	Austin Rebello
4.3 Future Project Opportunities	Jair Meza
Appendix A: Informed Consent Form for Over-Eighteen	Brianna Sahagian
Appendix B: Informed Consent Form for Under-Eighteen Participants	Brianna Sahagian
Appendix C: VEX Team Interview Questions	All
Appendix D: Roughly Coded Interviews for Pro vs Cons	Austin Rebello

Table of Contents

1.0 Introduction

The Robotics Education & Competition (REC) Foundation is an organization dedicated to supporting STEM-based education through their global competitive robotics movement. In partnership with VEX Robotics Inc, the REC Foundation hosts multiple yearly robotics programs including the VEX IQ Competition, VEX Robotics Competition, and VEX U (REC Foundation, n.d.). Many VEX Robotics teams are encouraged to use the Proportional Integral Derivative (PID) control algorithm to improve the accuracy and consistency of their robot's movement. PID is a closed-loop algorithm that incorporates feedback into its execution, making this algorithm a desirable choice for VEX teams that want smooth motion to a target position (Barr, 2002). This algorithm can be extended to any motor on the chassis or mechanism, making PID reusable across robotic systems. While PID is a flexible and useful algorithm for competition, the algorithm involves complex calculus-based concepts that can be difficult to understand without dedicated educational resources. However, VEX teams that are novice programmers find it difficult to implement PID over different robotic systems, as is evident by the hundreds of forum posts regarding issues with implementing PID.

This project aims to create a simple VEX library using the VEXCode API for beginner to intermediate teams that allows them to quickly and accurately move their robot with PID control during the autonomous stages of competition. We also will create standalone tutorials with videos on a supplemental website to document the library API. The library will serve as the intermediate stage between pure VEXCode V5 and established third-party APIs; it is meant to help teams take the step from the beginner to advanced level ("VEXcode API Reference", n.d.). Another goal is to successfully donate the MQP materials to one of the teams in the VEX community after the project concludes. Finally, the team intends to post the library as an open-source project so that it can continue to benefit teams in future seasons.

2.0 Background

VEX Robotics programs invite students at different education levels to learn hardware, electrical, and software skills in a collaborative team environment (VEX, n.d.). In total, the VEX community consists of over 24,000 teams and 1,100,000 student competitors. The work on this Major Qualifying Project is focused on the subset of high school level teams within the 11,500 VEX teams that compete in the VEX Robotics Competition (VRC) (VEX Robotics, 2024).

2.1 The VEX Robotics Competition

The VEX Robotics Competition (VRC) incorporates a *versus* style of competition between pairs of opposing red and blue robots (VEX Robotics, 2024). Each competition has a fixed 15-second Autonomous Period where the robot executes instructions independent of human interference and a 1 minute and 45 second Driver Controlled Period. While every VRC follows these standards, each yearly challenge involves different mechanics that fundamentally alter the way VEX teams build and program their robots. The 2024 VRC, *Over Under*, features unique field elements called *Triballs* and two *Elevation Bars* for robots to interact with to score points. The *Triballs* are meant to be scored in the team's alliance net, and the *Elevation Bar* presents a final challenge for the alliance robots to ascend at the end of the Driver Controlled Period (VEX Robotics, 2024). The nature of these tasks determine the chassis type and mechanism design of robots participating in the *Over Under* VRC.

2.2 PID Control

PID control is the most common method of control used in industry applications (Emerson, 2023). PID is preferred professionally because the algorithm has the ability to adapt to different environments and allows for generally straightforward implementation. The PID controller market is predicted to reach $1.6 billion by the year 2026, with applications in the food and beverage, oil and gas, temperature, motion, flow, pressure, and power sectors (Global Industry Analysts, 2022). In the VEX Robotics setting, PID algorithms are used as motion controllers for individual motors and motor groups on a robot.

2.2.1 Open-loop and Closed-loop Control Systems

When considering types of algorithms to control motion, there are two general approaches: *open-loop* or *closed-loop* control. In open-loop control, the algorithm takes in a set of input parameters and uses the values of these fields to compute a response. During the execution of the algorithm, no new internal or external data is introduced into the calculations. This approach allows for efficient and cost-effective implementation of an algorithm. Examples of open-loop control systems include the software behind washing machines. While a washing machine takes in a user-specified set of parameters including load-size, spin cycle, and water temperature, these inputs remain static while the machine is running (Collimator, 2023).

In closed-loop control, the algorithm takes input from the initial set of parameters and factors the system output into its calculations. These systems are typically more complex than open-loop systems because the hardware must include a component to collect internal or external information. Different types of sensors can be used to collect external output; the type of information collected varies depending on the application. An example of a closed-loop control system is a smart temperature system. The software that operates this system takes in an initial temperature input and the external room temperature data to determine the heating or cooling state of the temperature system. Closed-loop systems are typically more expensive due to the need for sensing components, but algorithms using this approach are more precise and accurate than open-loop control systems.

PID control is a type of closed-loop control system. In the VEX setting, the algorithm takes in a target distance parameter as its *setpoint* or goal state. The setpoint is the target for the PID algorithm to consider when calculating its output. Typically, the setpoint is a distance or some other form of physical measurement and the system uses sensors to determine the *error*, that is the difference in the current state of the robot, and its setpoint. This sensor can be an external camera, ultrasonic range finder, line tracker, or encoder (internal or external). In this MQP, the chosen sensor for system output is the internal encoder system on the VEX V5 motors. The encoder returns the rotations that the motor has traveled. Using some basic geometry, the distance a robot travels can be calculated by measuring the circumference of the wheel, and multiplying it by the number of rotations the motor has spun. Typically, this is done with simple robots with wheels and motors directly geared. This traveled-distance is subtracted from the setpoint to determine the error. Then, the P, I, and D constants combine with the error to calculate the speed of the robot and the loop repeats by polling new sensor information to determine the new error. This process is demonstrated below in Figure 1.

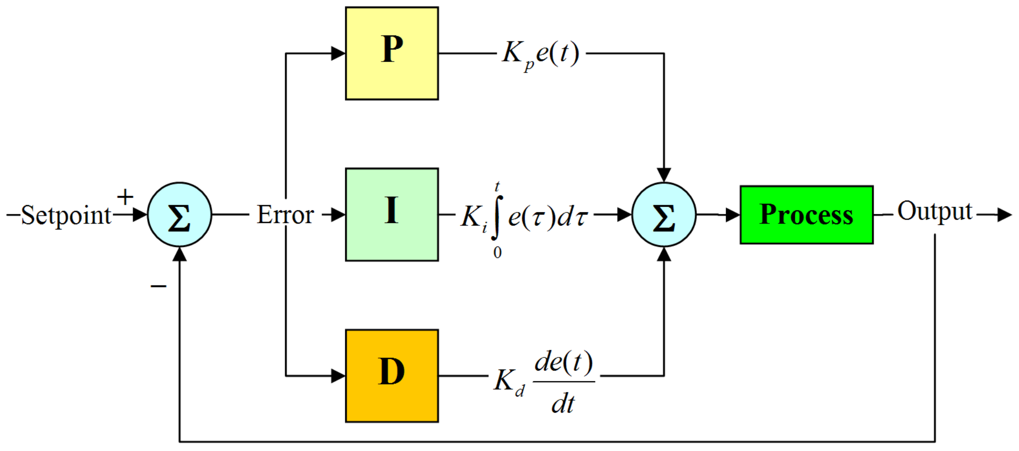


Figure 1. PID Closed-loop Control Diagram (Silver Star, 2006)

There are multiple ways to determine when the PID loop should finish executing. In some cases where the system needs to maintain a position, the loop might never end. This allows the loop to continue adjusting the actuator (the mechanism responsible for changing the state of the system) in the case that it is physically pushed off the setpoint. This will allow the actuator to respond as the error begins to change, attempting to settle back to the target. In many cases, especially in VEX where the setpoint changes multiple times throughout the course of an autonomous routine, the loop will end when the system is within an upper and lower bound of the target. For instance, if the system may determine that an error of 1% is enough to exit the loop, while in other cases a larger percent error may be introduced to allow the loop to exit sooner. The current output of the system can also be used to determine if it is attempting to settle, or still making adjustments. For example, if the speed output is higher than a predetermined threshold, the loop will continue as it is, assuming the system is still in motion and not attempting to stop. Once the system falls below this speed threshold, the loop may exit. Timeouts are also utilized to limit the amount of time a movement takes, and typically these are used as a last case scenario when the system should have already exited the loop.

2.2.2 P, I, and D Constants and Configurations

The PID controller operates using a combination of three constants: the proportional constant Kp, the integral constant Ki, and the derivative constant Kd (Tilbury and Messner, n.d.). These constants are predetermined gain values (a scalar value used to change the magnitude of an output when multiplied) that are multiplied with their corresponding calculated terms. The proportional term is calculated by finding the error of the system, and the Kp constant is multiplied, which proportionally increases the robot's response-speed based on error. With a higher error, the proportional term will yield a more immediate response, however; a high output can cause overshoot, where the robot hits its setpoint and then continues moving. The Ki constant multiplies with the summation of error to move the robot incrementally towards its setpoint, better known as the integral of the error over time. The integral adds more precision to the system, so that once the proportional term brings the robot close to its setpoint, the integration can cover the remaining distance. The combination of Kp and Ki is used in the PI control configuration of PID to quickly (Kp) and precisely (Ki) reach a setpoint. The Kd constant multiplies with the difference of error to decrease overshoot, or the derivative of the error over time. Its job is to predict the rate of change so that the system, allowing it to come to a stop sooner. The combination of Kp and Kd is used in the PD control configuration of PID to quickly (Kp) reach a setpoint while curbing overshoot (Kd). Using all three constants in conjunction implements the PID configuration, where the robot can quickly (Kp) and precisely (Ki) reach a setpoint while curbing overshoot (Kd). There are no configurations that exclude Kp as it is responsible for outputting a majority of the system's speed, where Ki and Kd are responsible for the end behavior.

2.2.3 PID Example

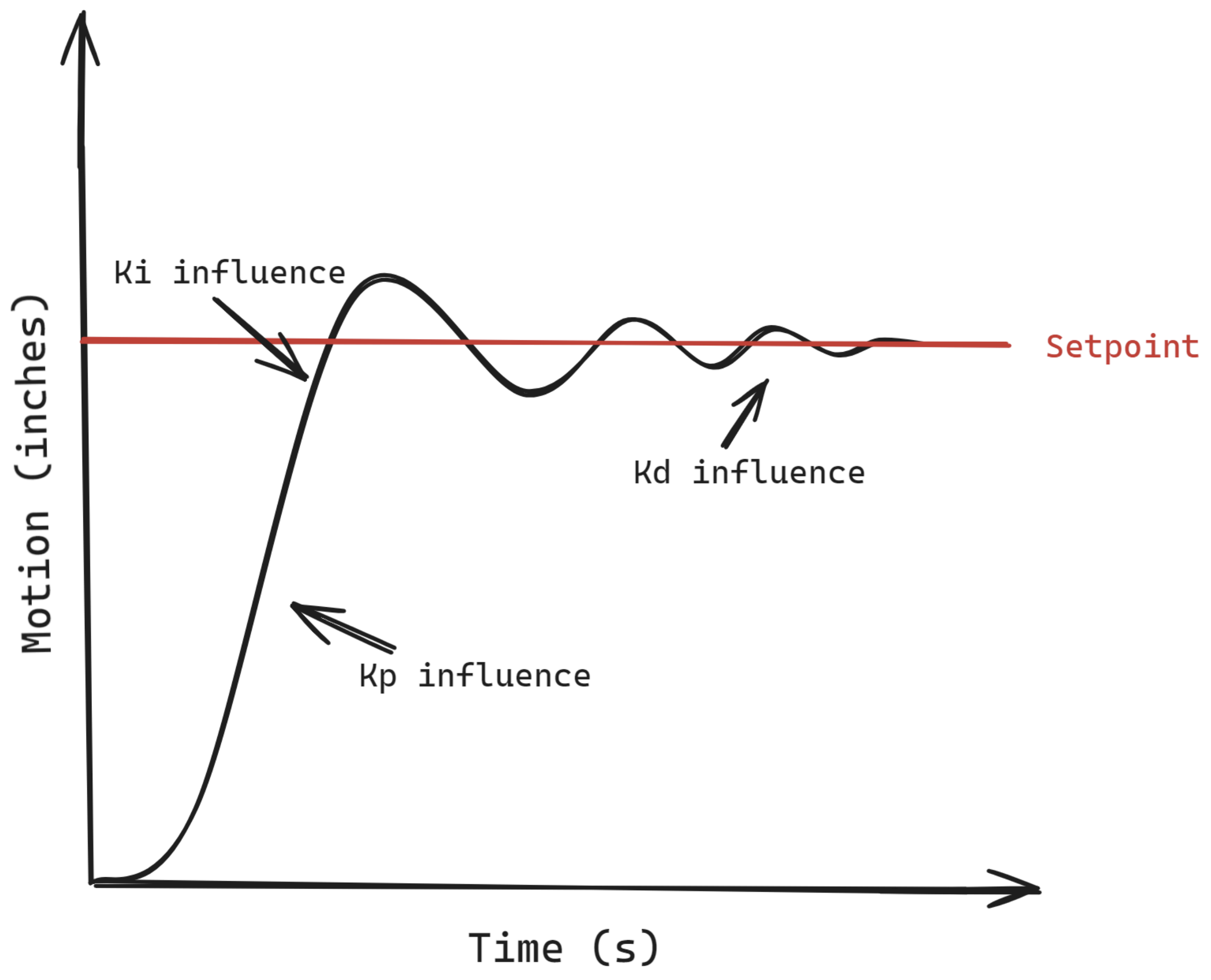


Figure 2. A representation of how each constant impacts PID control

Figure 2 displays distance readings that are outputted from a robot running a hypothetical route using PID. In this scenario, the robot is attempting to reach the setpoint at the red line on the graph. At the beginning of the route, the proportional control (using the Kp gain) has the majority of influence to quickly pick up speed and move the robot close to the red line. Then, once the robot gets close to the setpoint line, the integral control (using the Ki gain) takes over to push the robot the rest of the way to the line. In this scenario, the robot overshoots its setpoint and has to reverse direction to correct for the overshoot. This correction is shown at each bend of the curve above (and below) the setpoint line. During this overshoot is where the derivative control has its influence (using the Kd gain). The derivative control gradually reduces the overshoot in both directions until the robot eventually reaches the setpoint (+/- some remaining steady-state error).

2.2.4 PID Areas for Improvement

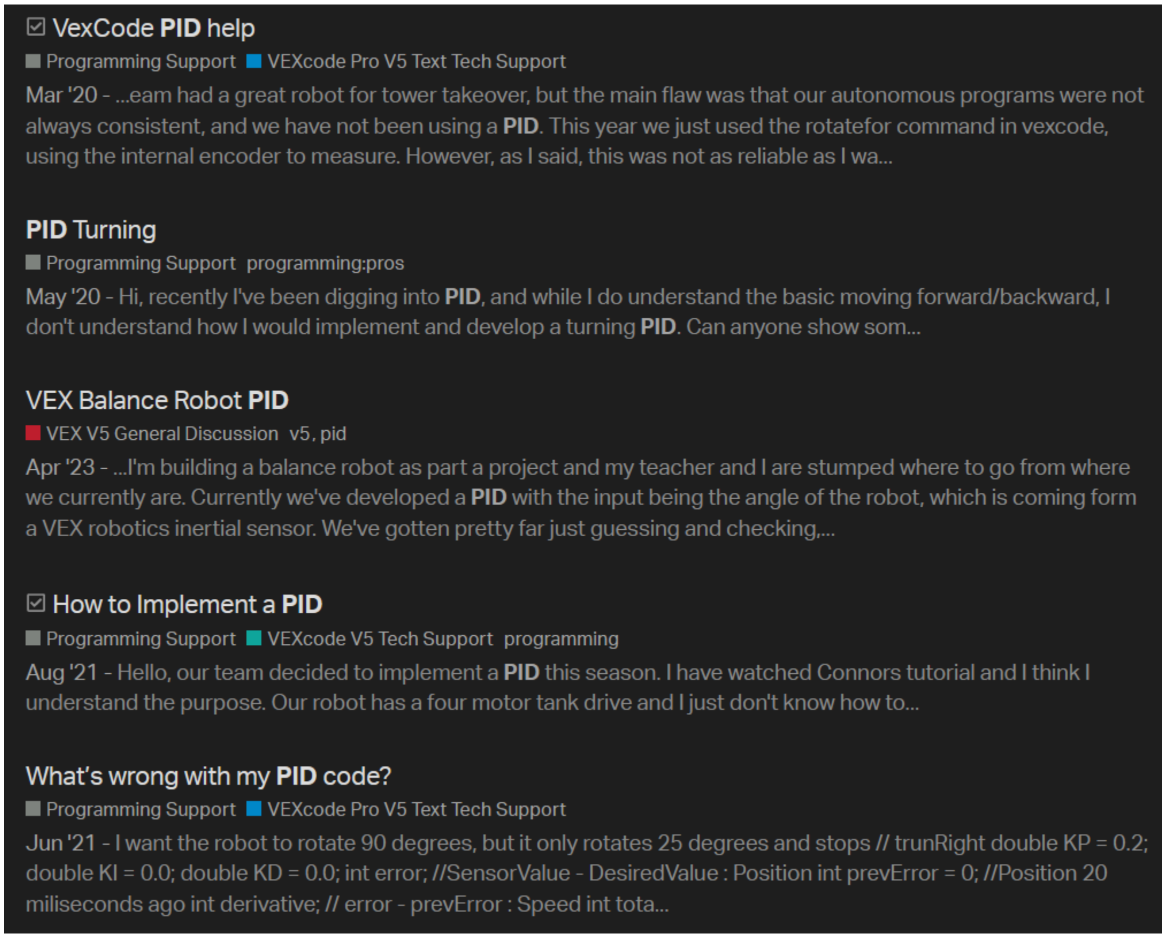
While PID serves as an adaptable and efficient algorithm for robotic motion, the algorithm has several drawbacks that can be addressed with supplemental utilities. The first implementation issue with PID occurs at the intersection of the Kp and Ki constants. When continuously integrating with Ki throughout a PID-controlled motion, the integral portion of the algorithm increases to the point where overshoot is difficult to avoid. In order to eliminate overshoot from PID motions, this continuous accumulation of error needs to be limited to specific sections of the motion. Another issue with PID is the high chance for derivative noise. The derivative portion of PID is subject to sharp fluctuations during motion that make this piece of the algorithm difficult to implement smoothly. Finally, the PID algorithm has no feature to support smooth transitions into and out of motions. PID control supports smooth motion in the intermediate stage between the start and end of the motion, but supplemental utilities are needed to prevent sudden starting and stopping.

2.3 The Current VEX Landscape

Over the course of 18 unique games, many VEX teams have found similar patterns to develop and program their robots, despite the vast differences in game mechanics. Some of these patterns relate to hardware such as joints that allow for mechanisms to rotate, while others might be within the code, such as a simple function to spin all the drivetrain motors simultaneously. Ultimately, the direction that the team decides to take will depend on their experience and exposure to these patterns. Because of the straightforward implementation of PID, many teams attempt to write their own PID algorithm from scratch, but struggle with the nuances of applying it to motors, and tuning the system. Some teams instead use third-party libraries to skip the implementation entirely, allowing for more time to be spent on other solutions.

2.3.1 Issues in Implementing PID

Among new teams, PID seems attractive because of the relatively simple code required to type out, as shown in Figure 4. Writing out a basic version of this algorithm in Python shows that it only takes a few lines to get an output that can be used to control motors. However, just because the code may appear to be simple, the integral and derivative terms alone cause confusion in teams that either don’t understand calculus or the PID algorithm itself. When teams have issues, they turn to the official VEX Forums, which allows users to make posts asking questions and discussing topics relating to all things VEX. On the forums, there are dozens of posts with titles like, “Help with PID”, or “How to implement PID” and even “PID not working” (VexForum, 2024). In these posts, users paste in their code, along with any questions they have relating to why the algorithm won’t work as they intended. While many teams get the basic structure, there are a multitude of mistakes such as hard coding values, poor understanding of the programming language itself, or just naive mistakes such as doing the wrong calculations. There are multiple responses to PID topics a day, and many new topics created each week, as seen in Figure 3.


☑ VexCode **PID** help
Programming Support VEXcode Pro V5 Text Tech Support
Mar '20 - ...eam had a great robot for tower takeover, but the main flaw was that our autonomous programs were not always consistent, and we have not been using a **PID**. This year we just used the rotatefor command in vexcode, using the internal encoder to measure. However, as I said, this was not as reliable as I wa...

PID Turning
Programming Support programming:pros
May '20 - Hi, recently I've been digging into **PID**, and while I do understand the basic moving forward/backward, I don't understand how I would implement and develop a turning **PID**. Can anyone show som...

VEX Balance Robot **PID**
VEX V5 General Discussion v5, pid
Apr '23 - ...I'm building a balance robot as part a project and my teacher and I are stumped where to go from where we currently are. Currently we've developed a **PID** with the input being the angle of the robot, which is coming form a VEX robotics inertial sensor. We've gotten pretty far just guessing and checking,...

☑ How to Implement a **PID**
Programming Support VEXcode V5 Tech Support programming
Aug '21 - Hello, our team decided to implement a **PID** this season. I have watched Connors tutorial and I think I understand the purpose. Our robot has a four motor tank drive and I just don't know how to...

What's wrong with my **PID** code?
Programming Support VEXcode Pro V5 Text Tech Support
Jun '21 - I want the robot to rotate 90 degrees, but it only rotates 25 degrees and stops // trunRight double KP = 0.2; double KI = 0.0; double KD = 0.0; int error; //SensorValue - DesiredValue : Position int prevError = 0; //Position 20 miliseconds ago int derivative; // error - prevError : Speed int tota...

Figure 3. A screenshot of the official VEX forum with the search query: “PID”(VEXForum, 2024).

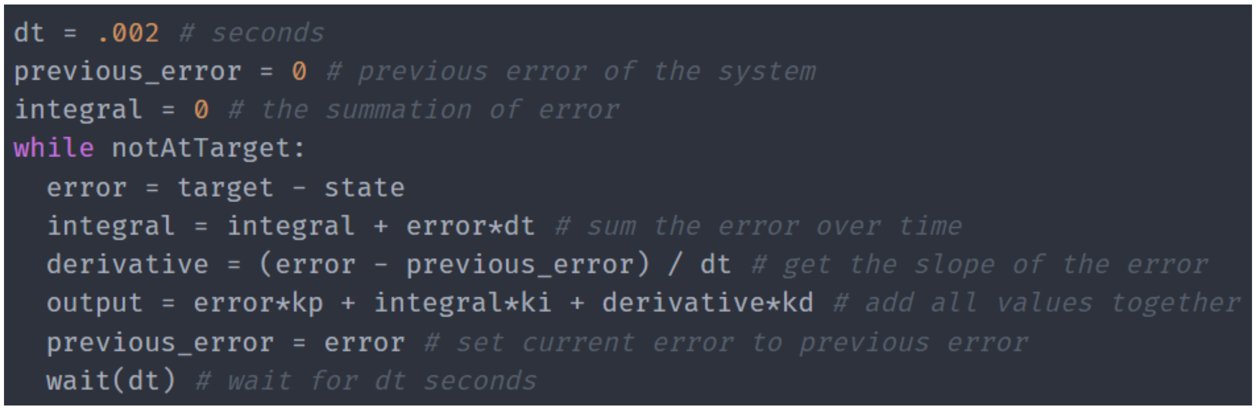

```python
dt = .002 # seconds
previous_error = 0 # previous error of the system
integral = 0 # the summation of error
while notAtTarget:
  error = target - state
  integral = integral + error*dt # sum the error over time
  derivative = (error - previous_error) / dt # get the slope of the error
  output = error*kp + integral*ki + derivative*kd # add all values together
  previous_error = error # set current error to previous error
  wait(dt) # wait for dt seconds
```

Figure 4. Pseudocode of a basic PID algorithm in Python

2.3.2 Third Party Solutions

Despite the abundance of PID-related questions on the forums, it is still one of the most frequent topics. Teams that are either unsatisfied with their own implementation, or want something that is a bit more advanced, can opt to use a third-party library, or use pre-existing code that someone else has written. Nearly all of these resources are tailored for the intermediate to advanced level programmers looking to get an edge on their competition, leaving the beginners behind. One of the most popular third party resources is called PROS, which is a library of functions similar to VEXCode that allow the users to control the V5 system in various ways (PROS, n.d.). The library was developed by students at Purdue University, and is now open source, allowing anyone to inspect and suggest additions to the source code. PROS was built for users looking to control their robot in a more advanced manner, offering functions to control and monitor everything from current draw to motor speeds. The library is organized slightly differently than VEXCode and comes down to personal preference on whether a team adopts PROS over VEXCode.

One of the biggest reasons many programmers switch to PROS is for its excellent third party library integration. PROS comes preloaded with OkapiLib, a library that introduces even more advanced motion algorithms and functions for quick and precise autonomous movement (Okapi, n.d.). Okapi depends on the PROS API in order to control the V5 system, making OkapiLib exclusive to projects that use PROS as opposed to VEXCode. PROS also comes with LVGL, an open source C embedded graphics library that is used for things such as displaying sensor data, and autonomous selector screens (LVGL, n.d.). However, this library is not exclusive to PROS as a port is available for VEXCode users as well, though importing it is slightly more involved. Pros also supports templating, and a very popular example is EZ-Template (Zarchi, n.d.). EZ-Template utilizes the PROS API and offers users a project with empty initializers and constructors, allowing the user to input their robot’s features, constants, and other parameters, allowing simple setup. This template has been successful in competition, helping team 7686B win the world championship. EZ-Template has inspired a VEXCode alternative called JAR-Template which follows a similar pattern of inputting robot-specific parameters in a template project (JAR-Template, n.d.).

Although these solutions are extremely accurate, well-documented and community favorites, they all tailor to the advanced teams that have access to a multitude of sensors, programming knowledge and testing time. Many of the libraries require the use of external encoders, gyroscopes, and other advanced sensors that beginner teams might not be familiar with. The aim for WPID was to provide a VEXCode library that does not require any of these sensors, and only relies on the motors used in the construction of the robot. This allows for new teams to quickly implement WPID with minimal setup, and provide an out-of-the-box experience. The implementation of WPID is also different compared to these libraries and templates, as WPID is offered as a static library as opposed to a template, in order to reduce clutter in a project. However, the source code is available on the GitHub repository, linked in the bibliography section of this paper.

3.0 Methodology

The main goal of the project is to develop a simple PID library written with the VEXCode API. The VEXCode API allows beginner teams to easily incorporate the library in an already familiar environment that is also officially supported by VEX. Along with the library, tutorials and documentation will be developed to assist teams with the usage of the library, to help quickly implement and use the library effectively.

3.1 Coding the WPID Library

Our process of developing the library mirrors many teams starting their own program from scratch. It first started with tinkering with the VEXCode to further our understanding of what the API has to offer. This included writing a crude PID algorithm that would be used as the basis of the project. Once we were satisfied with our basic implementation, we started to hammer out the details of what exactly we expected to be in the library.

3.1.1 Structure and Organization

It was clear that because PID can be applied to many different types of actuators, mechanisms and drivetrains, it was important to develop the PID functions with that in mind. Since the algorithm would be used across various instances of classes that each have a collection of unique constants and data, it made sense to build a PID class to store these functions and variables. Doing so would allow any object to create an instance of PID and utilize the functions with its own parameters.

As for the mechanisms used in VEX, they can essentially be boiled down to a collection of motors used to manipulate and spin structural components such as gears, c-channel, and wheels shown in Figure 5. Ultimately, an aptly named **Mechanism** class was designed to store a group of motors, and a PID object to use for spinning the motors with the PID algorithm. The Mechanism class serves as a facade for vex motor groups, allowing them to interface seamlessly with the PID class (Gamma, 1995).

One of the most important mechanisms of any VEX robot is the drivetrain. It is responsible for moving the robot across the field, and is used to aim the other mechanisms to manipulate game objects. The drivetrain is also just a collection of motors that manipulate wheels to drive the robot, and therefore implement instances of the Mechanism class to control each part of the drivetrain. The most common setup for a drivetrain in VEX is the tank or skid steer setup, where motors are either connected to the left or right side of the chassis. Each side is independently controlled, allowing for forward and backward movement, as well as turning on the spot. Another common drivetrain is the H-Drive setup, which adds a third wheel perpendicular to the other drive wheels of a tank setup, allowing for side to side motion, with an example shown in Figure 6. Many of the drivetrains are similar in what they are able to accomplish, so a pure virtual class called **Chassis** was developed to hold common functions and variables used across all types of drivetrains. The **Tank** and **HDrive** classes extend this Chassis class, allowing them to implement these functions.

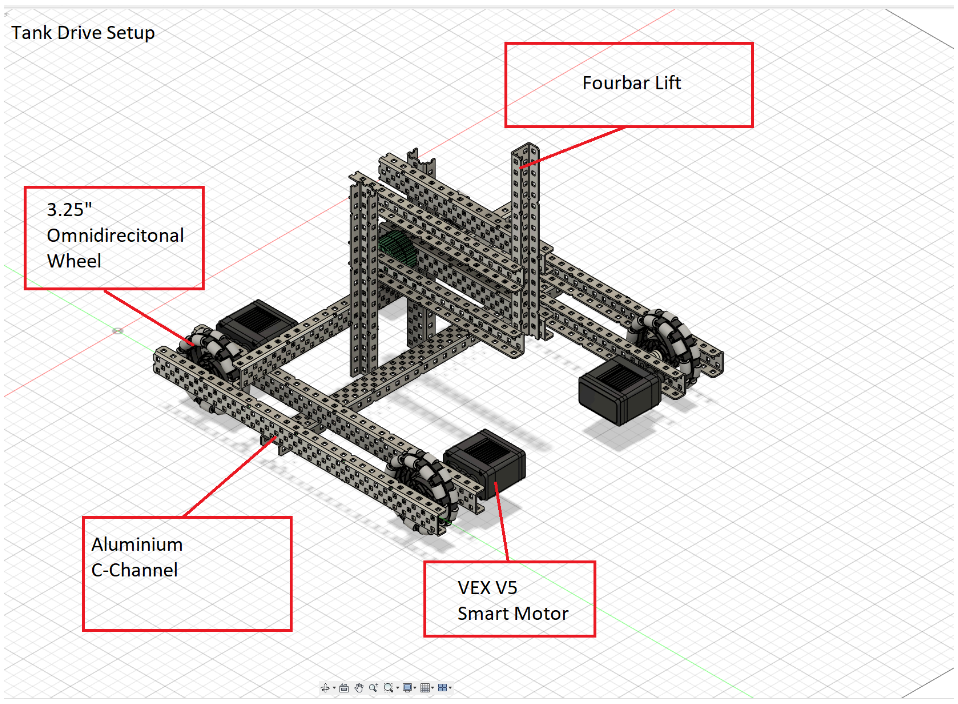


Figure 5. Basic CAD of a Tank Drive Chassis. This iteration also features a four bar lift.

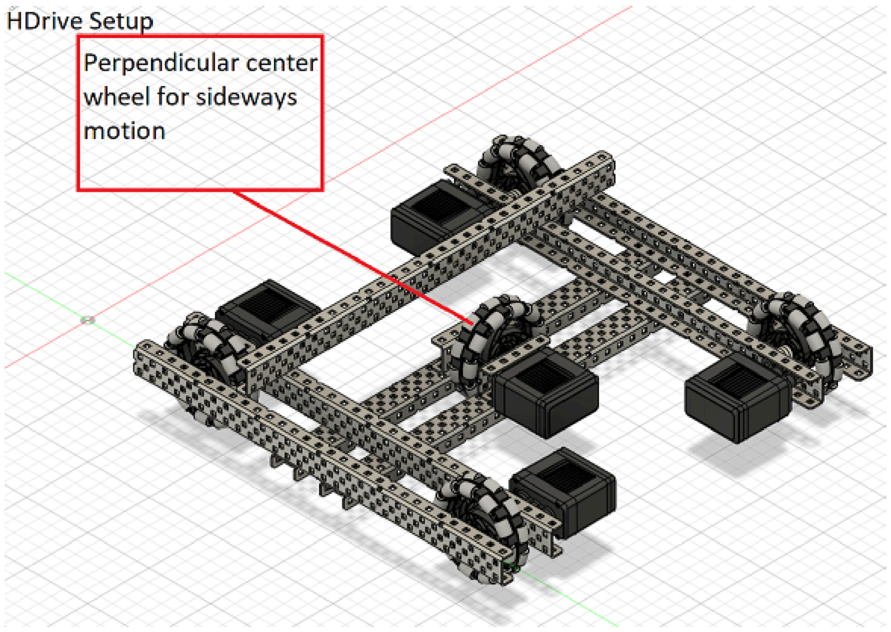


Figure 6. Basic CAD of an HDrive Chassis. Similar to Tank, the HDrive chassis features a skid steer style setup, with an additional center wheel for sideways motion.

3.1.2 Implementing the PID Class

The PID class contains multiple functions and variables to assist in the calculations for each Mechanism that utilizes PID. These variables consist of the required PID constants, in addition to a reference to the previous error, previous integral, and the error bounds. The actual algorithm is relatively simple, but includes a few additions to improve the effectiveness. The beginning portion of the code mirrors exactly what is shown in Figure 1. One of the additions includes a saturation check to make sure that the output of the PID loop does not exceed a maximum value set by the user. For example, if the user only wants the motors to spin at 20% speed, but the output of the algorithm tells the motors to run at 50% speed, the output will be reduced to 20% in order to avoid saturation. This allows the user to control the maximum speed a mechanism can spin.

Another addition is an integral anti-windup check that only allows integration when the system is not saturated. Saturation refers to when the output of the PID algorithm the physical limitations of the actuator it is outputting to. For example, if the maximum RPM of a motor is 200, and the algorithm determines an output of 300, the system is saturated. The anti-windup check restricts the maximum value of the integral by clamping, or capping, the value to a predetermined maximum. This is done by first storing the initial calculated speed value, and comparing it to the clamped speed value after the saturation check. If the initial speed and clamped speed are not equal, that means the system was saturated. Then, the sign of the controller output before the clamping is compared with the sign of the current error. If both have the same sign, that means the integrator is increasing the magnitude of the speed. In this case, the error for calculating the integral term is set to 0, effectively making the term equal to its previous value. These two checks determine if the integrator is actually making things worse by adding onto the calculated speed when it does not need to. The purpose of the integral term is to nudge the system to its target when the proportional term is not able to, and doing a conditional integration such as this will help tame the integrator by not letting it integrate unless the system is not being saturated.

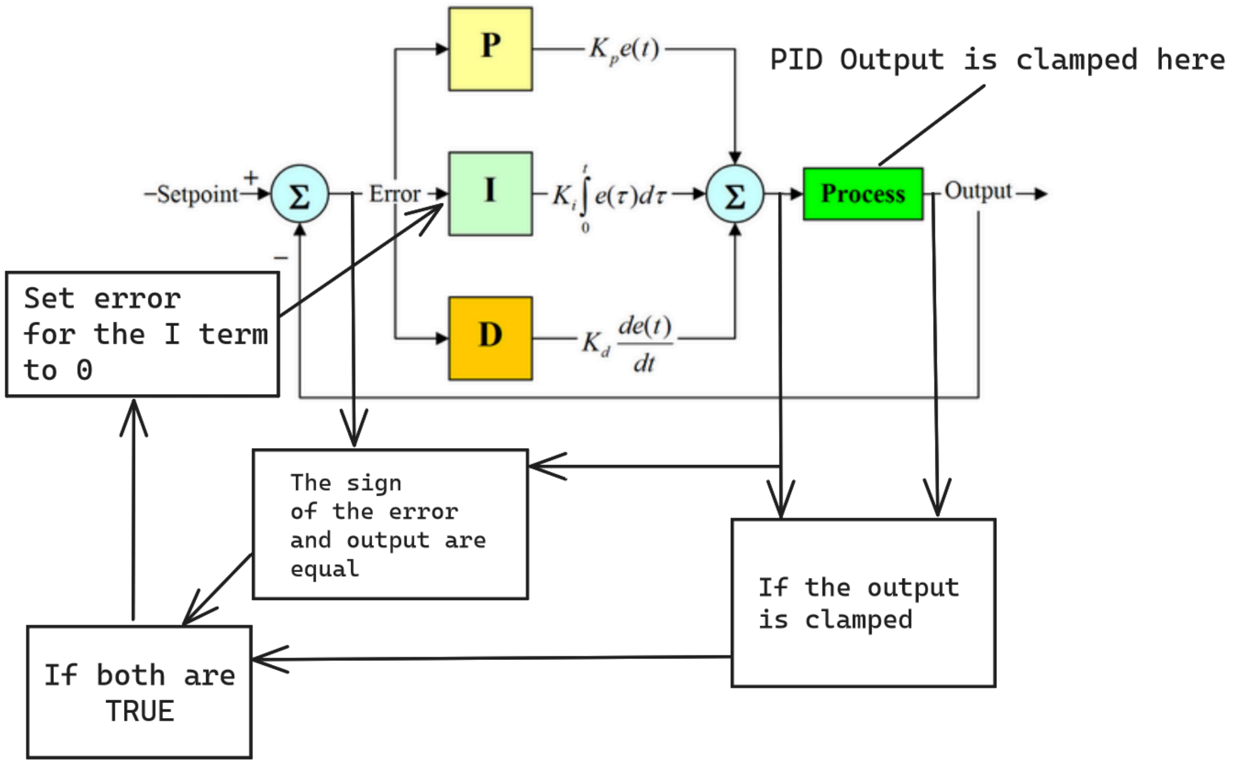
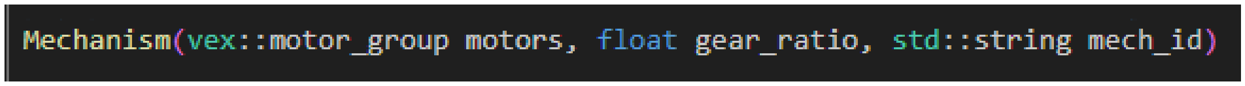

P
$K_p e(t)$
PID Output is clamped here
Setpoint
+
Σ
Error
I
$K_i \int_0^t e(\tau)d\tau$
Σ
Process
Output
−
D
$K_d \frac{de(t)}{dt}$
Set error for the I term to 0
The sign of the error and output are equal
If the output is clamped
If both are TRUE

Figure 7. A modified diagram showing the logic behind integral clamping. If the output is being clamped and the output is increasing error, then set the error of the integral term to 0.

The PID algorithm exits when the function **unfinished** determines if the error is within a predetermined upper and lower bound, and if the speed output is below the low speed threshold. A **reset** function resets the previous error and previous integral stored in the class, so that subsequent usage of the PID functions starts from an initial state. A **copy** function is used to copy PID constants to a new PID object for use when multiple mechanisms want to use the same constants, but different instances of a PID object. This was particularly useful for chassis control, where both sides of the chassis use the same constants, but store different error and integral values over the course of the loop.

3.1.3 Implementing the Mechanism Class

The fundamental basis of how WPID was developed is the idea that each component actuated by motors is just a collection of motors connected to metal, wheels or other objects. With this in mind, a **Mechanism** class was designed to hold information regarding these motors such as the actual motor objects, and the external gear ratios. The constructor takes in a vex::motor_group and a gear ratio, as well as an optional string identifier used in logging, shown below in Figure 8. The gear ratio term is the ratio of any internal and external gears used in the construction of the mechanism itself. Sometimes a gear ratio can be utilized to increase the speed of the mechanism for something like a flywheel, or increase torque for a lift. In either case, this term scales the output to account for the gearing, which ensures the position of the mechanism is accurate.



```
Mechanism(vex::motor_group motors, float gear_ratio, std::string mech_id)
```

Figure 8. Mechanism constructor.

The Mechanism class implements the PID algorithm through a private function named **spinToTarget()**. This function is responsible for calculating and driving the motors based on all the parameters available, such as the target position, and a maximum speed. It works by first aggregating all parameters required for PID calculations like the current position, the target, max speed, as well as adding the offset (explained in the next paragraph). These parameters are then used in a while loop to calculate the speed of the motors based on the current state. These parameters are passed into the PID algorithm; the output of which is checked and then relayed to the motors to spin. This function is the literal driving force behind all movement in the WPID library.

Offset is a band-aid solution for when your robot is under or overshooting its targets consistently, which works by adding a constant value to the target in the units of the motion where offset is being used. For example, during testing the robot would consistently reach 23 inches when the target was 24 inches, and to remedy this an offset of 1 inch was added to improve the precision. Offset was added to the library to help beginner teams improve their precision without needing to retune the system if there is a consistent error, but as stated before, it is a band-aid solution. This is especially useful when teams travel to a new environment with slightly different surfaces, or just to help increase the accuracy of the robot when needed. The offset can be set through the mechanism itself, and the units are in vex::rotationUnits::degree.

Other functions utilize this private function to complete both absolute and relative movement, as well as synchronous and asynchronous movement. These movement types are further explained in Section 3.1.5. Mechanism also contains other functions for polling data and setting other options such as the PID constants, and the error bounds. These functions can affect the behavior of the mechanism, and are also sometimes used for PID related calculations.

Mechanism was built to further abstract a motor_group, and was meant to be implemented in other classes that want to control Mechanisms in a specific way, such as a chassis. That is why each of the Chassis classes implement Mechanism in order to utilize all of WPID's functionality.

3.1.4 Implementing the Chassis Classes

The WPID library defines a chassis as the drivetrain of a robot. The chassis is necessary for any driving actions that a robot must take on the field. Our library's implementation of chassis consists of a hierarchy of different common chassis types used in the VRC setting: Tank Drive (also known as Skid Steer) and H-Drive.

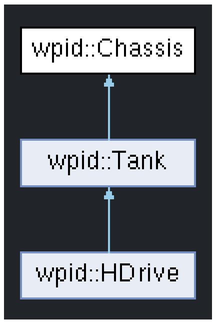
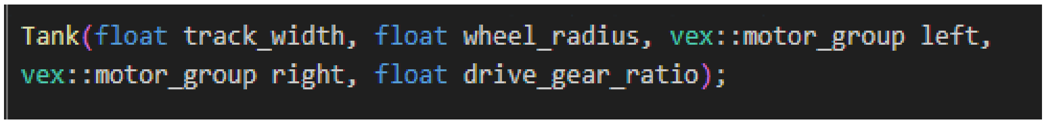


Figure 9. Chassis class inheritance diagram.

This chassis hierarchy includes one abstract class **Chassis**, its subclass **Tank**, and a Tank subclass **HDrive**. The pure virtual Chassis class contains all function signatures and fields that are shared between the available chassis types. The Tank classes declare and implement each of the functions from the Chassis class, as shown in Figure 10 below. Tank class also inherits all the data fields contained in Chassis.h. The HDrive classes declare and implement each of the functions from the Tank class, plus new functions that relate to the extra center motor group. Likewise, the HDrive class inherits all the data fields contained in Chassis but adds new data fields that relate to the center motor group.

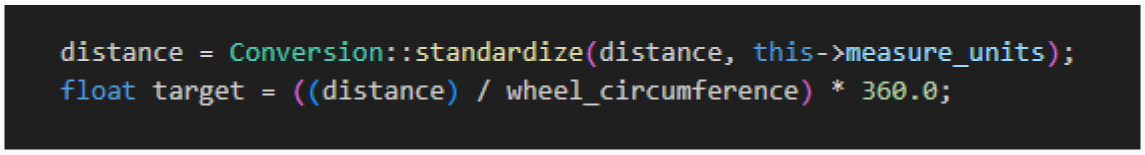
Every chassis is made up of its track width, wheel radii, motor groups, and gear ratios. The track width variable contains the distance between the centers of the front/back wheels across the body of the robot; this variable is used in turning logic. The wheel radii are used to determine how far the robot travels per motor revolution. The motor groups define how many independently moving parts are in the drivetrain. The gear ratio is used to scale the output speeds based on any external or internal gears used in the drivetrain.



```
Tank(float track_width, float wheel_radius, vex::motor_group left,
vex::motor_group right, float drive_gear_ratio);
```

Figure 10. The Tank constructor.

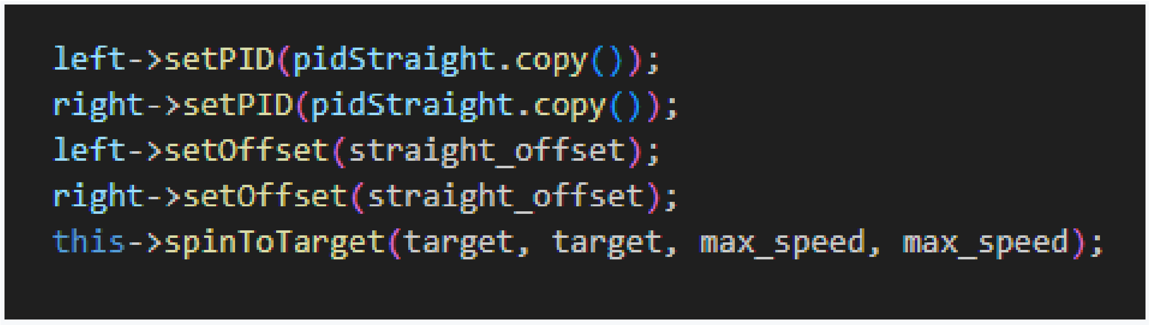
Each type of motion in any chassis class is built off of the internal **spinToTarget()** function used within the Mechanism class. Thus, we did not have to implement PID separately for straight, turning, or diagonal motion. The chassis movement functions simply pass the appropriate distances and directions to the internal mechanism function. Since each chassis movement function works in the same fundamental way, the **straight(float distance, int max_speed)** function implementation can serve as a general example of how this part of the library was written:



```
distance = Conversion::standardize(distance, this->measure_units);
float target = ((distance) / wheel_circumference) * 360.0;
```

Figure 11. Converting the distance between different units of measurement and setting the target with that converted unit.

In Figure 11 above, the **Conversion** class ensures that the entered distance units are transformed into standard inch units. Then, the target distance for the motion is translated from inches to motor degrees.

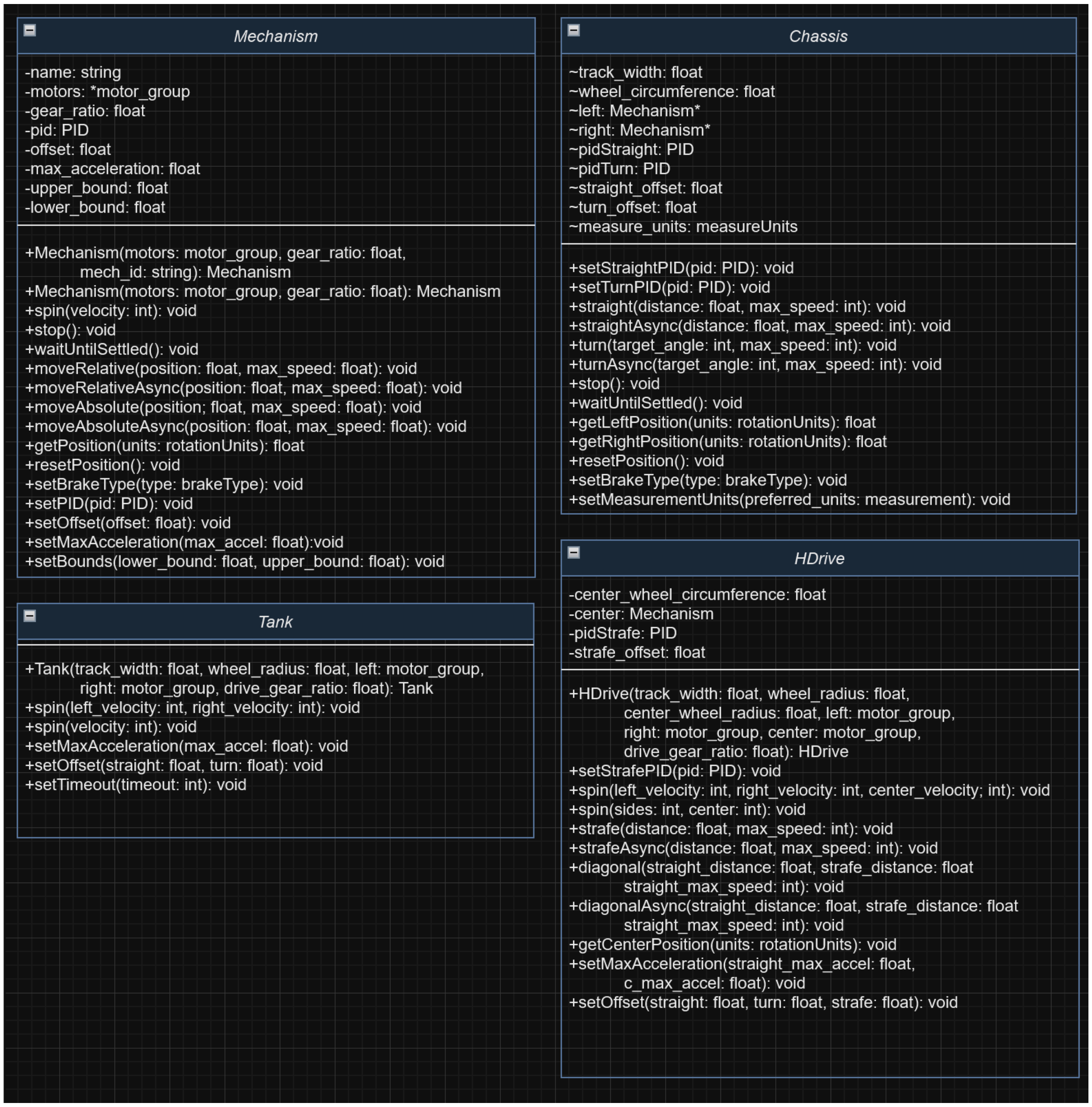


```
left->setPID(pidStraight.copy());
right->setPID(pidStraight.copy());
left->setOffset(straight_offset);
right->setOffset(straight_offset);
this->spinToTarget(target, target, max_speed, max_speed);
```

Figure 12. PID and Offset values are assigned to all motor groups before spinning the motors to the target location.

Next, the straight function assigns the correct PID object to each motor_group, as shown in Figure 12. Each PID object (pidStraight, pidTurn, and pidStrafe) has gains tuned specifically to suit the motion it corresponds to, and must be manually changed before each type of motion. The function repeats this step with offset. After these steps, the distance and max_speed parameters are sent to a function that transfers control to Mechanism movement functions.

Overall, the Chassis class and its hierarchy provide a bridge between the VEX programmer's intended instructions and the internal Mechanism class. The threading and PID implementation are details that we abstracted away to make the Chassis interface more user-friendly with a logic flow that is relatively simple to follow.



```
Mechanism
-name: string
-motors: *motor_group
-gear_ratio: float
-pid: PID
-offset: float
-max_acceleration: float
-upper_bound: float
-lower_bound: float

+Mechanism(motors: motor_group, gear_ratio: float,
        mech_id: string): Mechanism
+Mechanism(motors: motor_group, gear_ratio: float): Mechanism
+spin(velocity: int): void
+stop(): void
+waitUntilSettled(): void
+moveRelative(position: float, max_speed: float): void
+moveRelativeAsync(position: float, max_speed: float): void
+moveAbsolute(position; float, max_speed: float): void
+moveAbsoluteAsync(position: float, max_speed: float): void
+getPosition(units: rotationUnits): float
+resetPosition(): void
+setBrakeType(type: brakeType): void
+setPID(pid: PID): void
+setOffset(offset: float): void
+setMaxAcceleration(max_accel: float):void
+setBounds(lower_bound: float, upper_bound: float): void
```

```
Tank

+Tank(track_width: float, wheel_radius: float, left: motor_group,
        right: motor_group, drive_gear_ratio: float): Tank
+spin(left_velocity: int, right_velocity: int): void
+spin(velocity: int): void
+setMaxAcceleration(max_accel: float): void
+setOffset(straight: float, turn: float): void
+setTimeout(timeout: int): void
```

```
Chassis
~track_width: float
~wheel_circumference: float
~left: Mechanism*
~right: Mechanism*
~pidStraight: PID
~pidTurn: PID
~straight_offset: float
~turn_offset: float
~measure_units: measureUnits

+setStraightPID(pid: PID): void
+setTurnPID(pid: PID): void
+straight(distance: float, max_speed: int): void
+straightAsync(distance: float, max_speed: int): void
+turn(target_angle: int, max_speed: int): void
+turnAsync(target_angle: int, max_speed: int): void
+stop(): void
+waitUntilSettled(): void
+getLeftPosition(units: rotationUnits): float
+getRightPosition(units: rotationUnits): float
+resetPosition(): void
+setBrakeType(type: brakeType): void
+setMeasurementUnits(preferred_units: measurement): void
```

```
HDrive
-center_wheel_circumference: float
-center: Mechanism
-pidStrafe: PID
-strafe_offset: float

+HDrive(track_width: float, wheel_radius: float,
        center_wheel_radius: float, left: motor_group,
        right: motor_group, center: motor_group,
        drive_gear_ratio: float): HDrive
+setStrafePID(pid: PID): void
+spin(left_velocity: int, right_velocity: int, center_velocity; int): void
+spin(sides: int, center: int): void
+strafe(distance: float, max_speed: int): void
+strafeAsync(distance: float, max_speed: int): void
+diagonal(straight_distance: float, strafe_distance: float
        straight_max_speed: int): void
+diagonalAsync(straight_distance: float, strafe_distance: float
        straight_max_speed: int): void
+getCenterPosition(units: rotationUnits): void
+setMaxAcceleration(straight_max_accel: float,
        c_max_accel: float): void
+setOffset(straight: float, turn: float, strafe: float): void
```

Figure 13. UML Class diagrams of Chassis, Tank, HDrive and Mechanism. These classes are responsible for outputting PID calculated speeds to the motors.

3.1.5 *Types of Motion*

In the WPID library, there are four types of motion. *Absolute* motion moves the mechanisms to an absolute position, relative to where the motors are initialized. *Relative* motion takes the current state of the mechanism and adds the input to produce a target. *Synchronous* motion blocks any other function calls until this motion is complete. *Asynchronous* motion calls do not block code after the function call. Each type of motion has various use cases, which can be determined by the programmers and autonomous designers as needed.

Absolute motion is done by initializing the mechanism to a starting position, denoted as 0 degrees. For most mechanisms such as a lift, the starting position is usually determined by a physical limit. This initialization occurs automatically within WPID and is done when the program initializes each of the motors on startup. However, the user may change this location with the **resetPosition**() function, in the case that they employ other methods of ensuring the mechanism is at its starting point, such as using limit switches or other physical hard stops. This is useful for ensuring the robot returns to a "home" position. Absolute motion assumes the target is relative to the starting point of the mechanism. For example, if the target is 80 degrees, and the mechanism is currently at 20 degrees, the mechanism will move only an additional 60 degrees. Relative motion is done by adding the target of the function to the current position of the robot. In this case, the robot starting at 20 degrees, moving relative to 80 degrees will actually result in the mechanism finishing at 100 degrees. In short, absolute motion uses the starting point as a reference, where relative motion uses the current position of the mechanism as its reference point.

Synchronous motion is motion that does not allow any other function calls to occur until this movement is complete. This type of motion is useful when the user wishes to move a single mechanism at a time, which can help maintain the robot's balance, accuracy and precision. Sometimes moving multiple mechanisms is more efficient, which Asynchronous motion allows. These functions do not block the program and allow other function calls to occur after. When Asynchronous functions are used, a helper function called **waitUntilSettled()** can be useful in determining when the mechanism is finished moving. The wait function does indeed block the program from executing any other functions until this function returns, signifying the mechanism has finished moving. Using both of these types of movement can easily improve the efficiency of your autonomous runs by allowing multiple mechanisms to move concurrently. An example of using async motion with an H-Drive chassis can be seen in Figure 14 below.

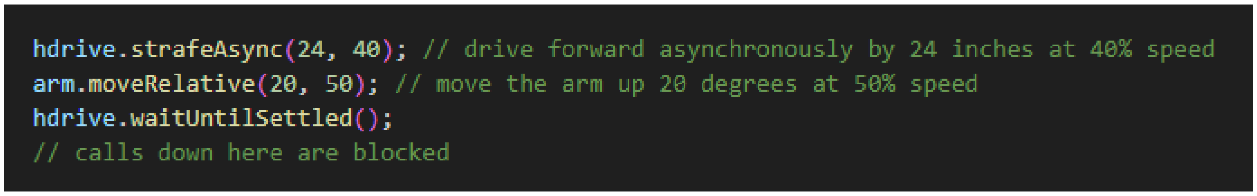

```
hdrive.strafeAsync(24, 40); // drive forward asynchronously by 24 inches at 40% speed
arm.moveRelative(20, 50); // move the arm up 20 degrees at 50% speed
hdrive.waitUntilSettled();
// calls down here are blocked
```

Figure 14. The async call allows other functions to be performed underneath, while the waitUntilSettled() function blocks the program until the chassis has finished moving.

3.1.6 Logging Capabilities

Our library will also come with two logging functions that have their own separate purpose for coding in our library. The first capability is the **LOG()** function, which is mainly focused on debugging the programmer-written code by providing feedback on the robot's movement. This is achieved with the **Logger.h** file, a component of the WPID namespace that accepts three log types when LOG() is called, DEBUG, INFO, WARN.

Flagging a LOG() type as DEBUG, INFO, or WARN does not result in different outputs, it is a visual label for the programmer to set themselves for their understanding of what is being logged. DEBUG and INFO are most commonly used for logging program data, such as robot movement parameters, directly to the console to visually read the output of the program. This helps spot where the code may be going wrong by detecting erroneous outputs in the PID's parameters. WARN is most commonly used where undesirable behavior of the robot is being detected. An excellent example of this is when the robot's movement times out after taking too long to come to a complete stop, the WARN log would display this information to the user, and they would then know that was the explanation for the robot's behavior.

The second logging capability is a graphical script in python that helps visualize PID parameters (speed, error, distance to target, Kp, Ki, and Kd) in an understandable manner. This is achieved by the robot first logging these parameters to CSV files while the robot is moving, and is saved with a naming convention that uses the VEX::timer, which starts at 0 every time the program is run. This logging capability does require VEX teams to possess micro-SD cards, and likely microSD to SD card adapters to plug into the programmer's computer.

Once the programmer plugs in the micro-SD card with the logged VEX data, they then have to transfer the files to the python_resources/VexLogs/ file path within our library, and remove any existing CSV files in that folder. Now, all the programmer has to do is run the graphData.py python file in the python_resources folder, producing a graphic similar to figure X below.

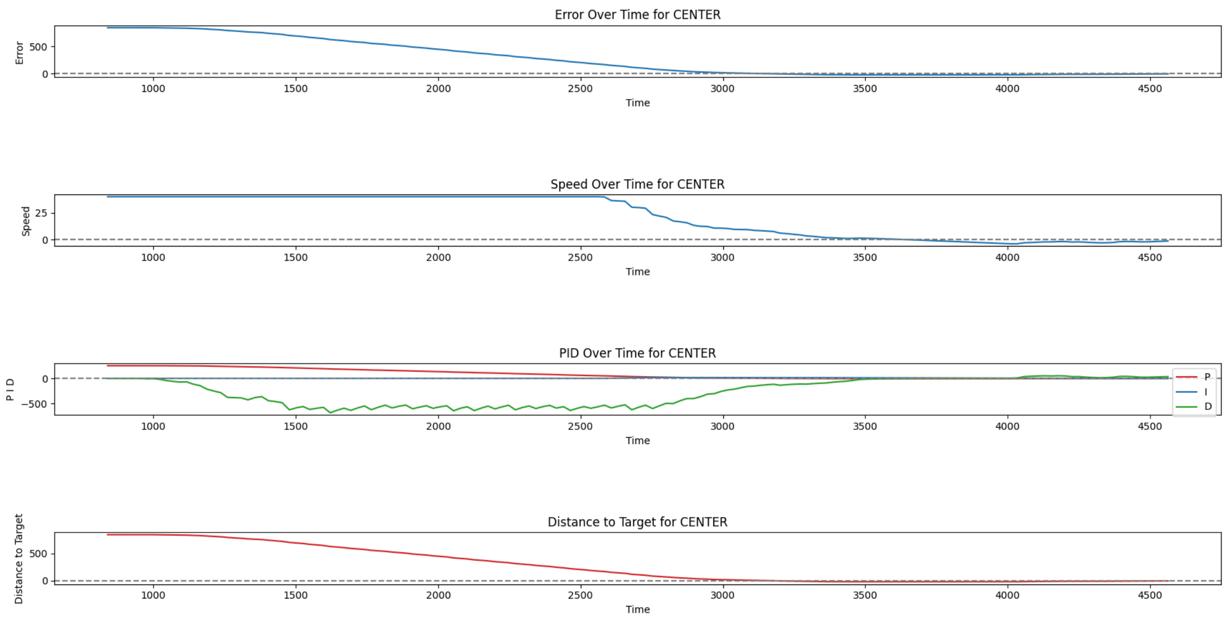


Figure 15. Example output from the graphData.py file.

The files placed in VexLogs are all pulled in at once, and sorted by which motor group and motor instruction was assigned to that file, creating a data frame per motor group that contains all the instructions given to those motors. The data is then split into each of the respective graphs shown in Figure 15 above, with Error Over Time at the top, followed by Speed Over Time, PID Over Time and finally Distance Over Time. A window with these graphs pops up for each motor group that is currently instantiated on the robot. Lastly, the programmer can use command line arguments when calling graphData.py to only include some of the PID parameters in PID Over Time. For example, calling graphData.py P I will not graph Kd, or only using D will exclude Kp and Ki. By default, all PID values are included in the graphs.

3.2 Conducting the Study on VEX Robotics Students

This study intends to interview current VEX competitors about their experience with existing control resources and collect student feedback on the WPID resources in the scope of VEX competitions. The study will also analyze the autonomous routes that students create with the library to determine the usability and comprehensibility of the resources.

3.2.1 Exploration

In the initial stages of the research, we gave students in each VEX team access to the WPID library and its supplemental materials. The presented materials will include the files needed to import the WPID library into their own projects, a documentation website, and tutorials on how to use the library. We then took a step back to allow students to explore the library, testing out features and familiarizing themselves with its core functionality. Students may also use the website tutorials and documentation for assistance, but we did not answer any questions or assist in this process. The objective of this stage was to simulate the environment where the students would discover the resource during a competition season. Student teams were expected to use their own robots, and assumed any responsibility with handling the robot and any programmed motion safely. The library does have safeguards to prevent some extreme movements, such as limiting a mechanism's motion, but this had to be set by the students in order to function. This initial exploration stage took place over a period of one week. The amount or extent of usage was not controlled during this stage of the study; the students decided themselves if they wanted to solely read the documentation or attempt to use the entire library's functionality.

3.2.2 Testing

After some exploration and experimentation, we instructed the students to perform specific tasks utilizing the library by developing a simple autonomous routine. We recommended that the students create a new project so that the code was separate from their existing programs, though not required. The nature of this 'test' was relatively low-complexity, but covered parts of the library that the students may not have been completely familiar with. The goal of this stage is to see how quickly a team can write and debug a simple autonomous routine utilizing the library. This period started after the week of exploration, and took place over one week. After the week of testing, we asked for a video of the VEX team's robot performing the routine. We also asked for access to the students' program and reviewed the written code. Any amount of completion was acceptable.

3.2.3 Interview

The final portion of the research consisted of an online or in-person interview with each VEX team. A full list of questions can be found in Appendix C. Interviews were not expected to exceed forty-five minutes. Through the interview, we gained a better understanding of how high school level VEX teams program their robots and collected student opinions on the WPID library. The interview began with basic background questions to get an understanding of the team's programming skill level and their experience with robotics as a whole. The next portion consisted of questions relating to robotics-specific ideas, such as control algorithms. Finally, we asked questions pertaining to the library itself. Some questions asked about ease of use, while others focused on the supplemental material.

3.2.4 Research Conclusion

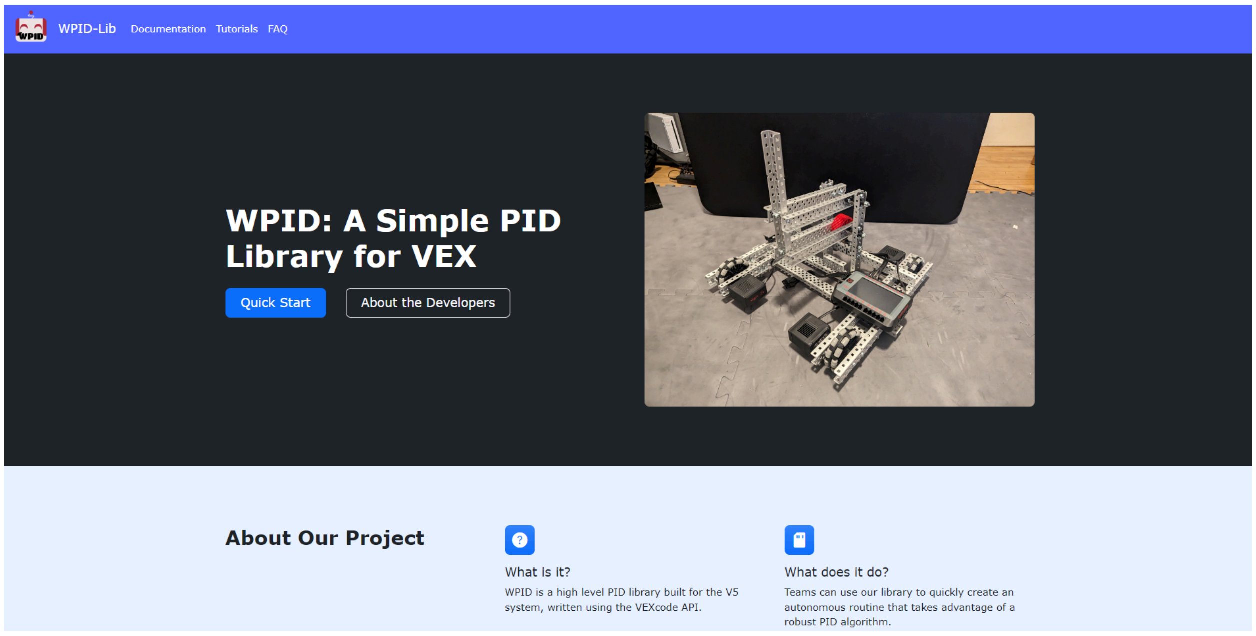
We utilized the methodology for this study to analyze how high school VEX students interact with the WPID library and its supporting website. Through the study, the team also measured student satisfaction with the created resource. Ultimately, the research team intends for students to learn and explore the library in a low-stakes environment, while also learning other important robotics concepts that are oftentimes difficult to implement due to a lack of dedicated VEX resources. The researchers have not set expectations relating to the extent of usage of the library, as it is important for the study to emulate the environment in which an independent team would come across the WPID resource after its release.

3.3 Creating the Website Documentation and Tutorials

To complement the VEX library and to assist the students with implementing and understanding PID control, a website will be developed to host the library's documentation as well as video and text tutorials on how to use the library. Ensuring the website is easy to navigate, read, and understand is a top priority to maximize the ability of the documentation and tutorials to teach beginner VEX students how to implement PID movement on their robots.

3.3.1 Website Design and Implementation

The website is a collection of static HTML pages hosted via GitHub's Pages (WPID, 2024). Through the use of CSS styling and Bootstrap, the website was created to be easy to navigate both on computer monitors and mobile devices. The appearance of the site's landing page is found in Figure 16 for desktop views and Figure 17 for a mobile view. We chose a deliberate contrast of colors for the site design, specifically on the home page, to draw the users towards the quick start section of the home page. This draws users to go right towards the introductory tutorials for using our library, shown in Figure 18, and from there, they naturally head towards the more advanced tutorials as they progress through PID development.



WPID: A Simple PID Library for VEX

About Our Project

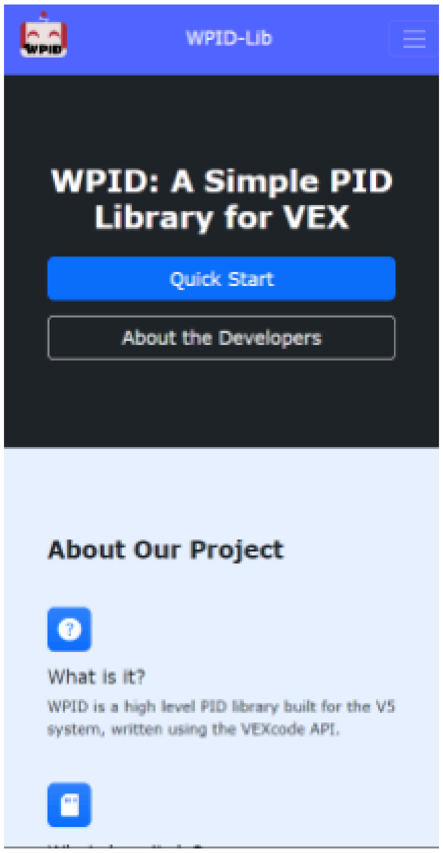
What is it?

WPID is a high level PID library built for the V5 system, written using the VEXcode API.

What does it do?

Teams can use our library to quickly create an autonomous routine that takes advantage of a robust PID algorithm.

Figure 16. The WPID Library website's landing page from a desktop view.


WPID-Lib

WPID: A Simple PID Library for VEX

Quick Start

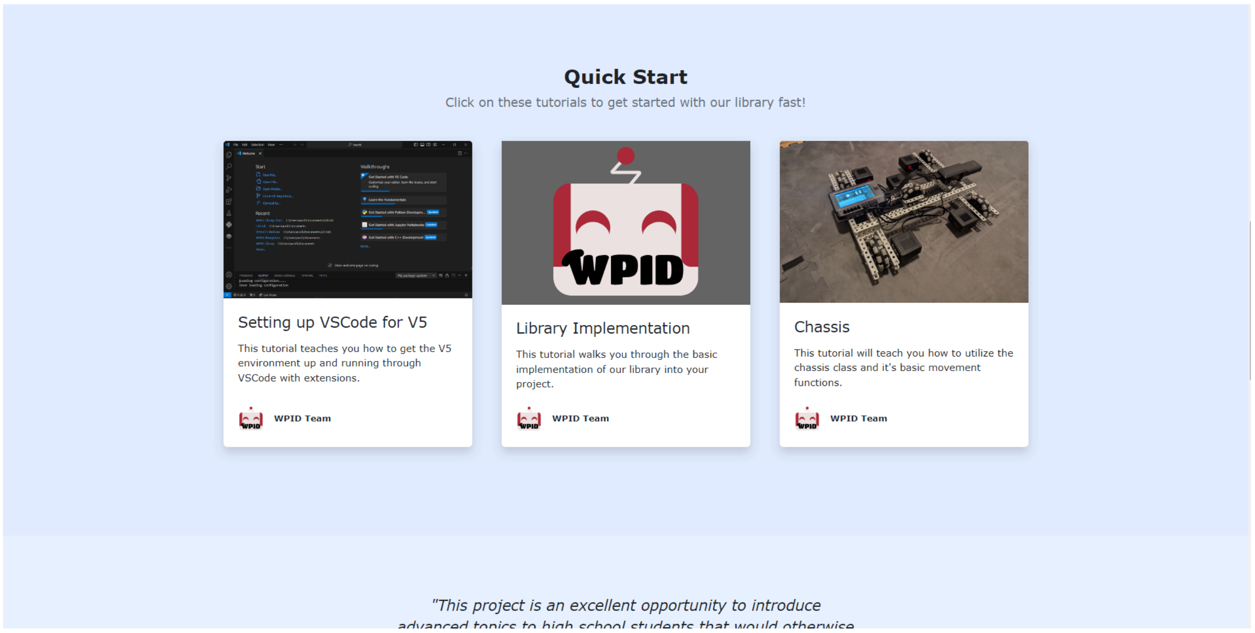
About the Developers

About Our Project

What is it?

WPID is a high level PID library built for the V5 system, written using the VEXcode API.

Figure 17. The WPID Library website's landing page from a mobile device view.


Quick Start

Click on these tutorials to get started with our library fast!

Setting up VSCode for V5

This tutorial teaches you how to get the V5 environment up and running through VSCode with extensions.

WPID Team

Library Implementation

This tutorial walks you through the basic implementation of our library into your project.

WPID Team

Chassis

This tutorial will teach you how to utilize the chassis class and it's basic movement functions.

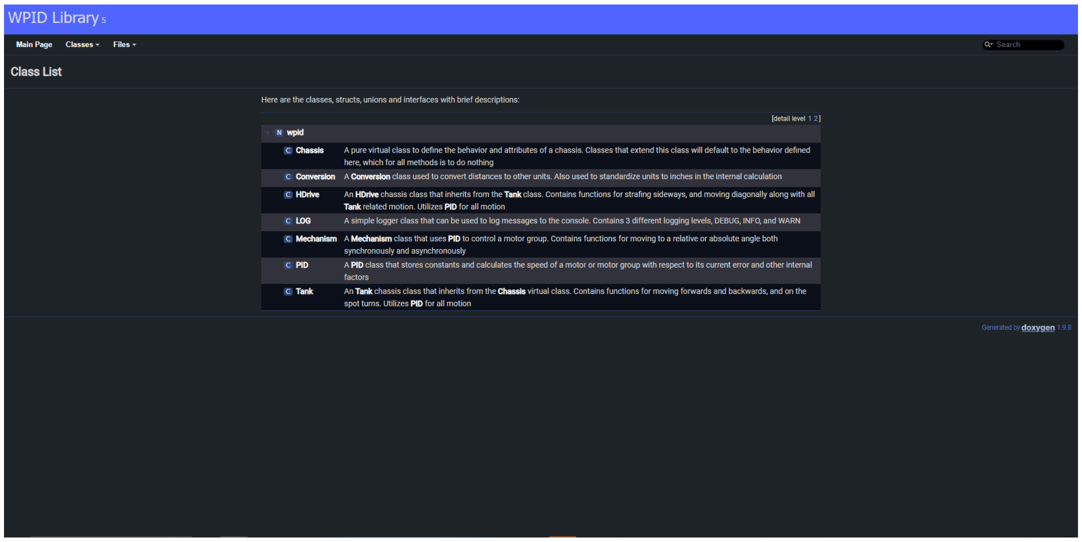
WPID Team

"This project is an excellent opportunity to introduce advanced topics to high school students that would otherwise

Figure 18. The destination of the "Quick Start" button on the landing page.

3.3.2 Documentation

The documentation of our library is hosted on this website. In order to maximize the consistency and readability of our documentation, we opted to generate the HTML, JS and CSS files with a program called Doxygen (Doxygen, 1997). This program is considered a standard tool for documenting programs written in C++, with the ability to read our entire library and generate accurate and easily readable documentation for our library. The documentation to our code can be accessed by clicking the "Documentation" hyperlink in the navigation menu, and takes you to the page depicted in Figure 19.



Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

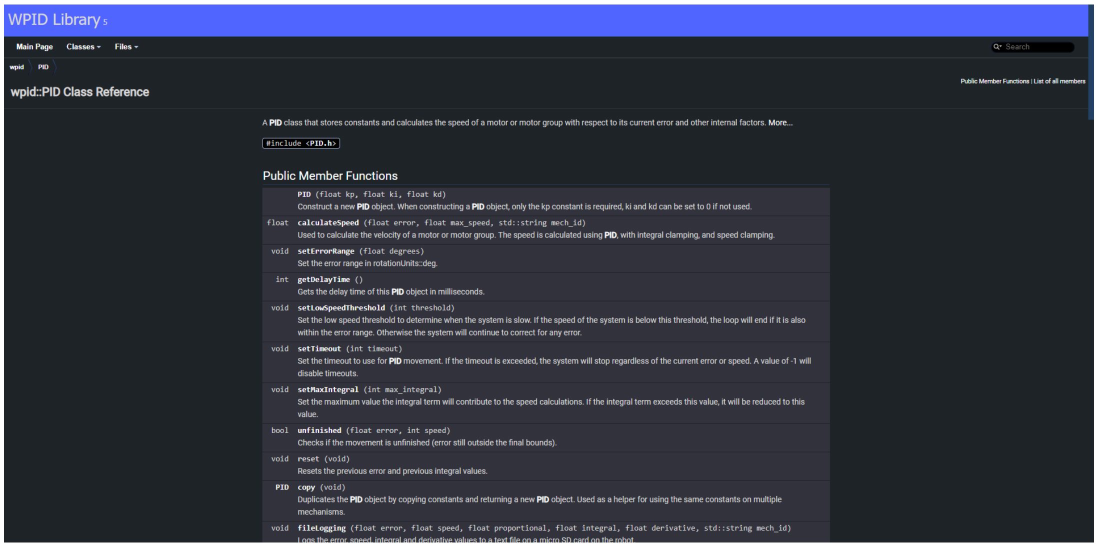
[detail level 1 2]

N wpid	
C Chassis	A pure virtual class to define the behavior and attributes of a chassis. Classes that extend this class will default to the behavior defined here, which for all methods is to do nothing
C Conversion	A **Conversion** class used to convert distances to other units. Also used to standardize units to inches in the internal calculation
C HDrive	An **HDrive** chassis class that inherits from the **Tank** class. Contains functions for strafing sideways, and moving diagonally along with all **Tank** related motion. Utilizes **PID** for all motion
C LOG	A simple logger class that can be used to log messages to the console. Contains 3 different logging levels, DEBUG, INFO, and WARN
C Mechanism	A **Mechanism** class that uses **PID** to control a motor group. Contains functions for moving to a relative or absolute angle both synchronously and asynchronously
C PID	A **PID** class that stores constants and calculates the speed of a motor or motor group with respect to its current error and other internal factors
C Tank	An **Tank** chassis class that inherits from the **Chassis** virtual class. Contains functions for moving forwards and backwards, and on the spot turns. Utilizes **PID** for all motion

Generated by doxygen 1.9.8

Figure 19. The central point of our library documentation on the website.

To run Doxygen properly, we used Javadoc style comments that Doxygen is able to ingest and generate documentation for our classes in the library. Navigating a Doxygen generated site is easy, since you can view any of our classes' functions and descriptions simply by clicking on the name shown in Figure 19. This takes the user to a page similar to Figure 20, where all the classes' constructors, destructors, public member function and regular member function are not only listed, but documented with comments. An example of the documentation of a member function is shown in Figure 21. Each function and its associated parameters have an accurate and detailed description.


WPID Library 5

Main Page Classes Files Search

wpid PID

Public Member Functions | List of all members

wpid::PID Class Reference

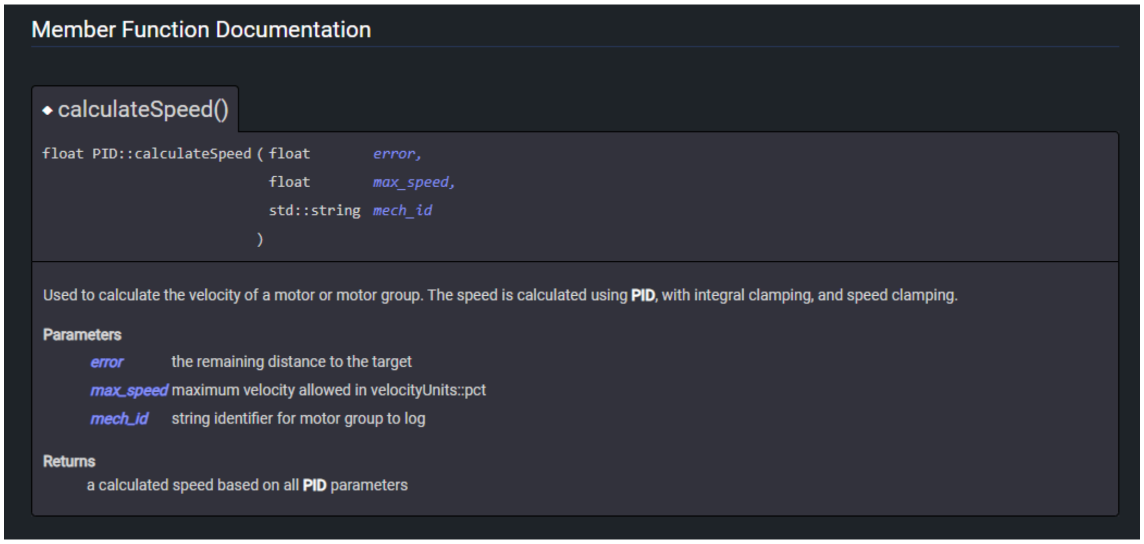
A **PID** class that stores constants and calculates the speed of a motor or motor group with respect to its current error and other internal factors. More...

```
#include <PID.h>
```

Public Member Functions

	`PID (float kp, float ki, float kd)` Construct a new **PID** object. When constructing a **PID** object, only the kp constant is required, ki and kd can be set to 0 if not used.
float	`calculateSpeed (float error, float max_speed, std::string mech_id)` Used to calculate the velocity of a motor or motor group. The speed is calculated using **PID**, with integral clamping, and speed clamping.
void	`setErrorRange (float degrees)` Set the error range in rotationUnits::deg.
int	`getDelayTime ()` Gets the delay time of this **PID** object in milliseconds.
void	`setLowSpeedThreshold (int threshold)` Set the low speed threshold to determine when the system is slow. If the speed of the system is below this threshold, the loop will end if it is also within the error range. Otherwise the system will continue to correct for any error.
void	`setTimeout (int timeout)` Set the timeout to use for **PID** movement. If the timeout is exceeded, the system will stop regardless of the current error or speed. A value of -1 will disable timeouts.
void	`setMaxIntegral (int max_integral)` Set the maximum value the integral term will contribute to the speed calculations. If the integral term exceeds this value, it will be reduced to this value.
bool	`unfinished (float error, int speed)` Checks if the movement is unfinished (error still outside the final bounds).
void	`reset (void)` Resets the previous error and previous integral values.
PID	`copy (void)` Duplicates the **PID** object by copying constants and returning a new **PID** object. Used as a helper for using the same constants on multiple mechanisms.
void	`fileLogging (float error, float speed, float proportional, float integral, float derivative, std::string mech_id)` Logs the error, speed, integral and derivative values to a text file on a micro SD card on the robot.

Figure 20. Our PID class page, representing an example of what Doxygen generates for part of our classes.


Member Function Documentation

◆ calculateSpeed()

```
float PID::calculateSpeed ( float        error,
                            float        max_speed,
                            std::string  mech_id
                          )
```

Used to calculate the velocity of a motor or motor group. The speed is calculated using **PID**, with integral clamping, and speed clamping.

Parameters

error	the remaining distance to the target
max_speed	maximum velocity allowed in velocityUnits::pct
mech_id	string identifier for motor group to log

Returns

a calculated speed based on all **PID** parameters

Figure 21. An example of a documented function within the PID class.

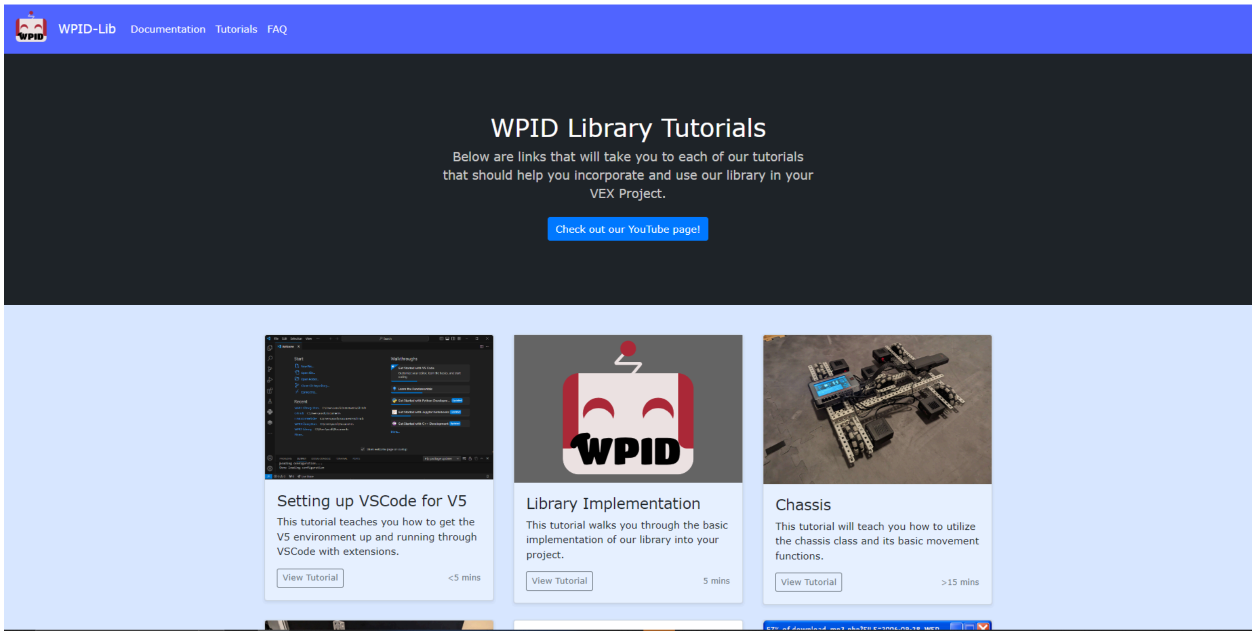
Our goal with using Doxygen is to provide clean, accurate and detailed code documentation that students can easily read and understand and do not have to be coded by hand. Doxygen is able to turn our code into a well documented project and an excellent teaching tool through its automatically generated documentation that can assist students, who may be new to programming, starting a VEX PID project.

3.3.3 Tutorials

The last portion of our website is dedicated to hosting video and/or text tutorials covering numerous topics that will allow users to get our library up and running on their machines. Figure 22 depicts the layout of the tutorials page. Every tutorial was written using markdown files, which are embedded inside the HTML files. Using markdown files is an excellent way to create easily readable tutorials that contain a mixture of code, commands, and text explanations. We have eight tutorial topics covering the following content:

- Setting up VSCode for V5
- Library Implementation
- Chassis
- Mechanism Setup/Usage
- What is PID?
- Data Logging
- Utilities
- Asynchronous Movement

Each of these topics is, at a minimum, covered by an in-depth text section that comprehensively explains the topic at a high school level. An example of what a tutorials page looks like is shown in Figure 23 If applicable, there are also video tutorials on some of the topics to act as a visual walkthrough of some of the process. This was added to assist in the implementation of our library and its code, as being able to visually see the steps a student needs to take can be easier than following text instructions. The video tutorials were recorded and uploaded to a YouTube channel called WPID Library, so the videos can be accessed both on and off of our website, with WPID Library Importing being the only video tutorial so far (WPID YouTube, 2024).



WPID Library Tutorials

Below are links that will take you to each of our tutorials that should help you incorporate and use our library in your VEX Project.

Setting up VSCode for V5

This tutorial teaches you how to get the V5 environment up and running through VSCode with extensions.

<5 mins

Library Implementation

This tutorial walks you through the basic implementation of our library into your project.

5 mins

Chassis

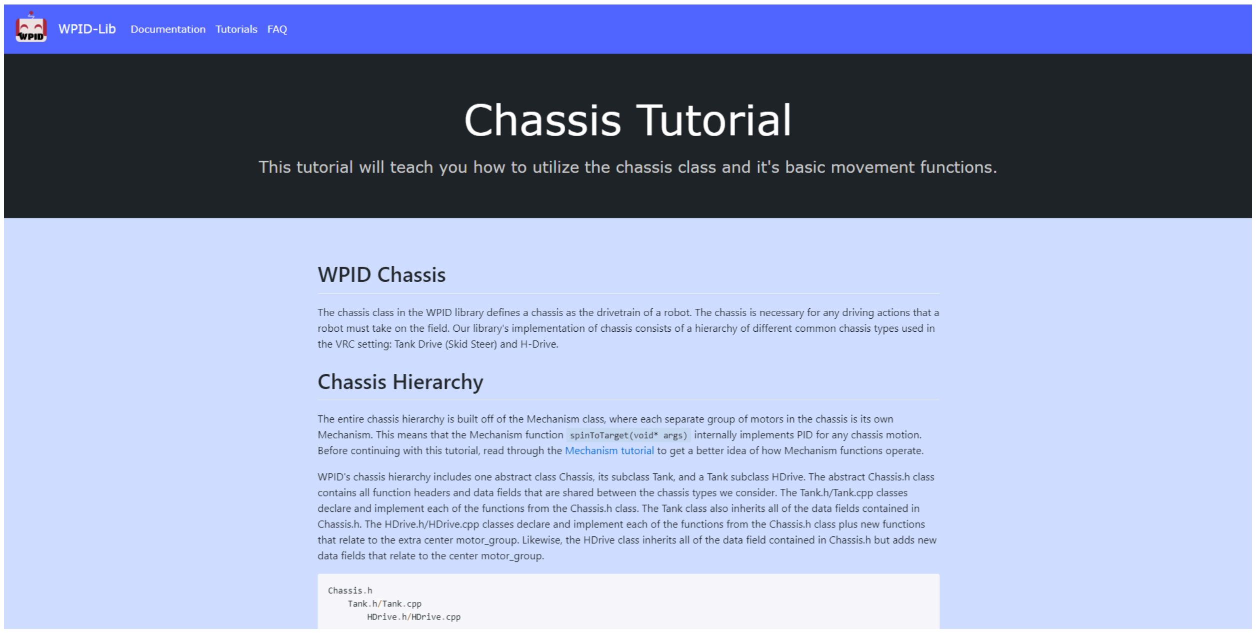
This tutorial will teach you how to utilize the chassis class and its basic movement functions.

>15 mins

Figure 22. The WPID Library Tutorials Page.

As mentioned earlier, each tutorial listed above has the goal of teaching entry-level and intermediate-level VEX students specific topics relating to implementing our library or PID control. The “Setting up VSCode for V5” tutorial teaches students how to get the V5 environment up and running through VSCode with extensions. The “Library Implementation” tutorial walks students through the basic implementation of our library into their projects. The “Chassis” tutorial teaches students how to utilize the chassis class, and its basic movement functions. The “Mechanism Setup/Usage” tutorial explains how to use our library to get the VEX Mechanism up and running in your program.

The “What is PID?” tutorial introduces the concept of PID to the students. The tutorial includes both educational explanations and code examples to assist in teaching the students the concept of PID and how to use it. The “Data Logging” tutorial exemplifies how to log the PID parameters that the VEX robot is using to a comma separated file and how to graph this data using an existing python script. The “Utilities” tutorial teaches students some utilities to assist in making the robot more consistent, which include the error bounds, bias, ramp up, integral cramping, console logging and unit conversion. Lastly, the “Asynchronous Movement” tutorial teaches students how to use the asynchronous functions to move multiple mechanisms at once.



Chassis Tutorial

This tutorial will teach you how to utilize the chassis class and it's basic movement functions.

WPID Chassis

The chassis class in the WPID library defines a chassis as the drivetrain of a robot. The chassis is necessary for any driving actions that a robot must take on the field. Our library's implementation of chassis consists of a hierarchy of different common chassis types used in the VRC setting: Tank Drive (Skid Steer) and H-Drive.

Chassis Hierarchy

The entire chassis hierarchy is built off of the Mechanism class, where each separate group of motors in the chassis is its own Mechanism. This means that the Mechanism function `spinToTarget(void* args)` internally implements PID for any chassis motion. Before continuing with this tutorial, read through the Mechanism tutorial to get a better idea of how Mechanism functions operate.

WPID's chassis hierarchy includes one abstract class Chassis, its subclass Tank, and a Tank subclass HDrive. The abstract Chassis.h class contains all function headers and data fields that are shared between the chassis types we consider. The Tank.h/Tank.cpp classes declare and implement each of the functions from the Chassis.h class. The Tank class also inherits all of the data fields contained in Chassis.h. The HDrive.h/HDrive.cpp classes declare and implement each of the functions from the Chassis.h class plus new functions that relate to the extra center motor_group. Likewise, the HDrive class inherits all of the data field contained in Chassis.h but adds new data fields that relate to the center motor_group.

```
Chassis.h
    Tank.h/Tank.cpp
        HDrive.h/HDrive.cpp
```

Figure 23. The Chassis Tutorial page, with the embedded markdown file containing the text based tutorial.

3.4 *Measuring Success*

This MQP project is intended to be an educational resource for high school students. Based on this goal, the best way to measure success for the project was to assess results directly from the students. In our evaluation, we considered two different types of success: success of the beta version of WPID and success of the MQP overall. The success of the library was measured based on an analysis of the feedback given from the VEX teams who participated in the study. We also examined the accuracy of our code by studying the videos and code bases provided by the VEX teams. The success of the MQP was based on student interest in our project's purpose, collection of actionable suggestions for improvement, and donation of the MQP materials to a VEX team.

3.4.1 *Autonomous Route Analysis*

The success of the beta release of the WPID library was measured in part through the analysis of student-created WPID autonomous routines. Because our study did not require a specific amount of completion, we evaluated the code bases and videos more qualitatively. In our evaluation, we considered the student's experience with C++, VEX, and general programming concepts (inheritance, objects, enumerations). The specific route that was given to the students was designed to include multiple turns and straight forward movement. A full image of this route can be found in the appendix.

3.4.2 Interview Responses

The success of the beta release of the WPID library was also determined by the responses given to our interview questions listed in Appendix C. The seven questions on the second half of the interview cover the students' interactions with our library. We created a spreadsheet with interview responses that reflected the results in a pro vs. con format. Using these summary statistics, we were able to identify overall positive and negative trends regarding different aspects of our resources. We decided that a net positive result (pro ratio of ≥ 50%) would signify success for the beta version of the WPID library. The results we collected from this analysis are depicted and interpreted in Section 4.2.

3.4.3 Donation of Materials

At the completion of our MQP and after we presented our MQP in April, we disassembled the robot and donated the materials to one of the high schools we interviewed for the study. This high school was the Moses Brown School in Providence, Rhode Island, where Jair Meza attended until 2020. We chose this school as a special thank you for participation in the study, and for the ease and cost of transporting the materials to them as opposed to shipping the materials to another team somewhere else in the country.

4.0 Results

After interviewing several VEX teams, gathering and analyzing their code/videos, and running our own structured tests, we were able to determine the benefits, drawbacks and accuracy of our VEX library and website.

4.1 Robot Testing

In order to verify the accuracy of our PID algorithm, we first created a series of tests using the fundamental types of motion that compose VEX autonomous routes. Every motion that our library can support can be broken down into straight and turn movements. Strafing is simply moving straight along the horizontal axis, while moving diagonally is a combination of straight and strafing motions. Thus, we decided to test the straight and turn functions with fully tuned PID. We decided to split these tests by distance, since our PID algorithm is based on the motor encoder readings. We found that with default parameters and a tuned PID system, the robot was able to end up relatively close to its setpoint within a reasonable amount of time. The average error across all straight and turn motions were 11% and 8.2% respectively. However, adding features in our library that support the PID algorithm for further tests yielded much better results. Adding in a tuned offset for each specific distance decreased the straight and turn average errors to 1.3% and 1.6%. According to these results, the library is effective at navigating a robot to its general target location (within ~10%), however; the library should be used with a tuned offset to get the most precise motion results (within ~1.5%). The reason for this difference could be due to a number of factors: wheel slippage, drift, motor slop (where the motors start running slightly before the wheels start spinning), and build issues all potentially impact the robot's ability to reach its target destination. Offset is able to counteract these factors in a way that the PID algorithm cannot; it acts as a manual correction value.

Next, we completed a summative test with the complex square motion. This motion started at a point that we labeled {0 in, 0 in, 0°} and included four periods of straight movement followed by turning. We tested this motion with offset, based on the conclusion from our previous tests. We started the test runs using the offsets we had identified earlier for 24 in and 90°. In this summative test, the robot ended its motion at the average point of {0.3 in, 0.8 in, 2°}, where each number is the absolute value of our average distance away from the starting point. Because this complex motion consists of a series of eight distinct movements, the team found this average error to be acceptable in a competition setting. Overall, the team concluded that our library is effective at moving the robot close to its setpoint during simple and complex motions.

Motion	Speed	Expected Position	Actual Position	Changed Parameters	Average Error
Straight	15	4 in	{3.25, -, -, 3.5, 3.38} in	max_acceleration = 0.5	17%
Straight	50	24 in	{22.75, -, 23.25, 23, 23.25} in	None	4.2%
Straight	50	60 in	{57, 57.5, -, 57.75, -} in	None	4.2%
Turn	25	40°	{40, 31, 36, 39, 35}°	None	9.5%
Turn	25	90°	{82, -, 80, 85, 85}°	None	8.0%
Turn	25	180°	{168, 167, 168, 166, 168}°	None	7.0%
Straight	15	4 in	{4, 3.875, 4, 4, 4} in	offset = 0.674 in	0.63%
Straight	50	24 in	{23.75, 23.5, 23.75, 23.5, 23.75} in	offset = 1 in	1.5%
Straight	50	60 in	{61.25, 61, 60.75, 61, 61} in	offset = 2.5 in	1.7%
Turn	25	40°	{39, 39, 40, 39, 39}°	offset = 3.8°	2.0%
Turn	25	90°	{90, 87, -, 89, 88}°	offset = 7.2°	2.0%
Turn	25	180°	{178, 176, 179, 179, 180}°	offset = 12.6°	0.89%
Square	35	{0 in, 0 in, 0°}	[{0, -0.5, -1}, {0, -1.5, -2}, {0, 0, -3}, {-1.5, -1, -1}, {0, 1, -3}] in °	straight_offset = 1 in turn_offset = 3.5°	N/A

Table 1. Table of results from simple and complex motion tests with WPID

4.2 VEX Student Trials

Our original goal was to have eight teams use our library and provide feedback, code, and video documentation of the teams using our library. Unfortunately, due to factors including lack of knowledge in C++, the short window for the study to be completed, and the study taking place in the middle of competition preparations, we were only able to interview three of the teams.

4.2.2 Successes

As a result of the interviews with three VEX teams, we were able to determine that this library is a success. Given the aforementioned success threshold being a positive ratio of Pros vs Cons (>50% Pros), we have far exceeded that value with a ratio of 2 Pros to every Con, or a percentage of about 66%. Two teams mentioned how well the code was documented, and stated that the documentation itself was very helpful in understanding the code. The third team made mentions of how the material on the website was “easy to understand” and that they had “no problems with the website.” The code itself was stated to be “easy to use,” with one team stating that they were “excited about the control” over the robot. Last and most importantly, two of the teams said they were excited to continue using our library after the study, as they felt it was a crucial resource to assist with their VEX autonomous motion.

During one of the team’s autonomous routines, the precision and consistency of each movement was relatively high. The routines were a recorded video of the robot performing a route, which was difficult to visually inspect the performance. Accompanying the video was the project used to program the robot for this route. Typically, robots moving autonomously are subject to error based on external factors. This error could throw the heading of the robot off by a few degrees, but this was not the case during this autonomous routine. The programmer was able to use a consistent angle to turn the robot, implying the degree of error was low enough to allow for this copy-paste behavior. Sometimes, changing the input by a few degrees is necessary when the code results in inconsistent turns, though this was not the case for this run. This was anticipated based on our previous in-house testing, as we were able to determine the library provides a consistent movement. There was only one autonomous routine to analyze, so there is a low sample size, but overall, the team is confident that the consistency of the library, through our testing and analysis, is sufficient for a new team to score some points.

4.2.3 Drawbacks

With the success came some drawbacks as well. The first one being that our sample size in the study was not as large as we had hoped for. Due to the study being conducted in the middle of the VEX competition season, a majority of our teams did not have the time to attempt our library, and of the three teams that did, two could not put as much time as they would have hoped for in the timespan we were working with.

As far as the code itself goes, the largest hurdle to the library was the coding language it was written in. Most high schoolers are unlikely to know C++ to the degree that was required when using the library, especially if they had not used VEXCode before, and it was noted that our documentation and tutorials on the website lacked any guidance on how to use the C++ programming language. Outside of C++, one team noted that they were not able to run the code as perfectly as they would have wanted to. This is likely due to the robot not being fully tuned as a result of the lack of time these teams had to work on the library during competition season, but could also be through an error on the utilization, though it is unclear as there was not enough time to analyze their program.

4.3 Future Project Opportunities

Although the library has proven to be successful in terms of what was measured, there are still multiple facets that need to be flushed out before the release version is given to the community. One crucial aspect is to improve the precision of the PID algorithm. This can include adjustments to calculations, adding some new features to make it easier and more predictable to tune, as well as enhancing the experience of using the PID class. Something that can be of great help is a more sophisticated derivative filtering system. At the moment, a low pass filter is built into our PID calculations. This filter's effects are extremely difficult to notice without using a high Kd gain, though a high gain typically results in poor settling behavior. Another major upgrade could be to simplify the overall class structure and functions to make it easier to read documentation, as well as use the library itself. There are some functions within the library that are somewhat ambiguous, which is only cleared up by reading the documentation and tutorials thoroughly. For example, functions pertaining to max acceleration are difficult to understand as the parameters to these functions are in odd units that aren't typically brought up otherwise. One improvement could be to automatically determine the acceleration during PID calculations so that the user has a smooth experience and does not have to interact with this function at all. Some other improvements include, but are not limited to, better thread safe logging, support for internal gear cartridges, a function to "engage" the wheels in order to reduce the slop within motors before a movement is called, swing and arc turns, simplifying the inheritance, and adding other drive train models that are common. Some advanced features to add could include support for external encoders, motion profiling, and drive to point algorithms, though these are slightly out of the scope for the library, as they are more suited for the advanced programmer. Ultimately, as long as the library is capable of consistent autonomous movement, with a very simple API that beginners who are less familiar with C++ can use effectively, then the WPID library will be capable of raising the skill floor when it comes to the autonomous period in a VEX VRC competition.

Bibliography

Barr, Michael, “Introduction to Closed-Loop Control and PID”, https://barrgroup.com/embedded-systems/how-to/pid-closed-loop-control, February 5th, 2024, 2002

Collimator, “Modern Control System Design: Building the Ultimate Automatic Response Systems”, https://www.collimator.ai/post/what-is-control-system-design, February 5th, 2024, 2023

Dawn Tilbury and Bill Messner, “Introduction: PID Controller Design”, https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID, February 5th, 2024, n.d.

Doxygen, https://www.doxygen.nl/, February 23, 2024, 1997

Emerson, “The PID Controller & Theory Explained”, https://www.ni.com/en/shop/labview/pid-theory-explained.html, February 5th, 2024, 2022

Gamma, E, “Design patterns elements of reusable object-oriented software”, February 15, 2024. 1994

Global Industry Analysts, “Global PID Controllers Market to Reach $1.6 Billion by 2026”, PR Newswire, https://www.prnewswire.com/news-releases/global-pid-controllers-market-to-reach-1-6-billion-by-2026--301506357.html, February 5th, 2024, 2022

Jackson Area Robotics, “JAR-Template”, **https://jacksonarearobotics.github.io/JAR-Template,** February 24, 2024, n.d.

LVGL, “Light and Versatile Graphics Library”, https://lvgl.io/, February 24th, 2024, n.d.

MATLAB, “Anti-windup for PID control | Understanding PID Control, Part 2”, https://www.youtube.com/watch?v=NVLXCwc8HzM, February 5th, 2024, 2018

Okapilib, “OkapiLib Index Page”, https://okapilib.github.io/OkapiLib/index.html, February 24th, 2024, n.d.

Purdue ACM SigBots, “PROS First Time Users Guide”, https://pros.cs.purdue.edu/v5/getting-started/new-users.html, February 24th, 2024, 2023

REC Foundation, https://roboticseducation.org/about-us/, February 16, 2024, n.d.

SilverStar, “PID Feedback Loop”, https://en.wikipedia.org/wiki/File:PID-feedback-loop-v1.png, February 12, 2024, 2006

“VEXcode API Reference”, https://api.vexcode.cloud/v5/, 2024, n.d.

VEXForum, https://www.vexforum.com/search?q=PID, February 5th, 2024, 2024

VEX, https://www.vexrobotics.com/support/about-vex, February 16, 2024, n.d.

VEX Robotics, https://www.vexrobotics.com/v5/competition/vrc-current-game, February 5, 2024, 2024

WPID, https://wpidlib.github.io/WPID-Library-Docs/index.html, February 16, 2024, 2024.

WPID YouTube, https://www.youtube.com/channel/UCotYEsT_Sekt51_GJWMwgQw, February 16, 2024, 2024.

Zarchi, Jess. “EZ-Template”, https://ez-robotics.github.io/EZ-Template/, February 24, 2024, 2024.

Appendix A: Informed Consent Form for Over-Eighteen Participants

Informed Consent Agreement for Participation in a Research Study

Investigators: Jair Meza, Austin Rebello, Brianna Sahagian

Contact Information: jdmeza@wpi.edu, amrebello@wpi.edu, bnsahagian@wpi.edu

Title of Research Study: WPID: A Simple PID Library for VEX

Introduction:
You are being asked to participate in a research study. Before you agree, however, you must be fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that you may experience as a result of your participation. This form presents information about the study so that you may make a fully informed decision regarding your participation.

Purpose of the study: VEX robotics members in high school must learn different ways to control their robots during their competition seasons. Oftentimes, beginner teams looking to take the leap to more intermediate control algorithms struggle with finding quality resources. These resources are scattered across the web, and existing code for control is either too advanced or lacks understandable documentation. Through our MQP project, the team has created a collection of tutorials and well-documented code to facilitate the learning process for VEX students. This study intends to interview and survey current VEX competitors about their experience with existing control resources and collect student feedback on the WPID resources in the scope of VEX competitions.

Procedures to be followed: In the initial stages of the research, the researchers will give students in each VEX team access to the WPID library and its supplemental materials. The researcher team will take a step back and allow students to explore the library, testing out features and familiarizing themselves with its core functionality. Students may also use the website tutorials and documentation for assistance, but the researchers will not answer any questions or assist in this process. Student teams are expected to use their own robots, and assume any responsibility with handling the robot and any programmed motion safely. The library does have safeguards to prevent some extreme movements, such as limiting a mechanism's motion, but this must be set by the students in order to function. This initial exploration stage will take place over a period of one week. The amount or extent of usage will not be controlled during this stage of the study; the students may decide themselves if they want to solely read the documentation or attempt to use the entire library's functionality. After some exploration and experimentation, the researchers will instruct the students to perform specific tasks utilizing the library by developing a simple autonomous routine. This period will start after the week of exploration, and will take place over one week. After the week of testing, the researchers will ask for a video of the VEX team's robot performing the routine. The research team will also ask for access to the students' program and review the written code. Any amount of completion is acceptable. The final portion of the research consists of an online or in-person interview with each VEX team about how the team approaches programming their robots and about the team's experience with the WPID library. It is expected that this interview will take no longer than 45 minutes.

Risks to study participants: The probability and magnitude of harm or discomfort anticipated in the research are not greater than those ordinarily encountered in daily life or during the performance of routine physical or psychological examinations or tests.

Benefits to research participants and others: Participants will receive educational value from the tutorials on WPID and through the utilization of the library. Topics covered in the tutorials include the PID robotics control algorithm, setup of the coding environment and the WPID resource, and utilization of the library's functionality. The study will also encourage students to practice developing autonomous routines using a custom set of functionalities based on resources they are familiar with. The larger VEX community will be able to access the WPID library and associated website resources upon completion of the MQP project. The MQP resources will retain the same educational value seen during the study, accessible to VEX teams through the Internet.

Record keeping and confidentiality: Only the investigators will have access to the records collected from this study. All interviews are anonymous, with no individual's names tied to any interview response, thus maintaining full confidentiality. This consent form is the only personal identification tying an individual to our study, and will not be shared publicly. Records of your participation in this study will be held confidential so far as permitted by law. However, the study investigators, the sponsor or its designee and, under certain circumstances, the Worcester Polytechnic Institute Institutional Review Board (WPI IRB) will be able to inspect and have access to confidential data that identify you by name. Any publication or presentation of the data will not identify you, only your interview responses may be presented publicly.

For more information about this research or about the rights of research participants, or in case of research-related injury, contact:
Investigators: Please see the top of the consent form for our contact information.
IRB Manager: Ruth McKeogh, Tel. (508)-831-6699, Email: irb@wpi.edu
Human Protection Administrator: Gabriel Johnson, Tel. (508)-831-4989, Email: gjohnson@wpi.edu

Your participation in this research is voluntary. Your refusal to participate will not result in any penalty to you or any loss of benefits to which you may otherwise be entitled. You may decide to stop participating in the research at any time without penalty or loss of other benefits. The project investigators retain the right to cancel or postpone the experimental procedures at any time they see fit.

By signing below, you acknowledge that you have been informed about and consent to be a participant in the study described above. Make sure that your questions are answered to your satisfaction before signing. You are entitled to retain a copy of this consent agreement.

____________________________ Date: ___________________

Study Participant Signature________________________________

Study Participant Name (Please print)

_____________________________________ Date: ___________________

Appendix B: Informed Consent Form for Under-Eighteen Participants

Informed Consent Agreement for Participation in a Research Study

Investigators: Jair Meza, Austin Rebello, Brianna Sahagian

Contact Information: jdmeza@wpi.edu, amrebello@wpi.edu, bnsahagian@wpi.edu

Title of Research Study: WPID: A Simple PID Library for VEX

Introduction:

You are the parent/guardian of ______________________________, a student being asked to participate in a research study. Before you agree, however, you must be fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that your child may experience as a result of their participation. This form presents information about the study so that you may make a fully informed decision regarding your child's participation.

Purpose of the study: VEX robotics members in high school must learn different ways to control their robots during their competition seasons. Oftentimes, beginner teams looking to take the leap to more intermediate control algorithms struggle with finding quality resources. These resources are scattered across the web, and existing code for control is either too advanced or lacks understandable documentation. Through our MQP project, the team has created a collection of tutorials and well-documented code to facilitate the learning process for VEX students. This study intends to interview and survey current VEX competitors about their experience with existing control resources and collect student feedback on the WPID resources in the scope of VEX competitions.

Procedures to be followed: In the initial stages of the research, the researchers will give students in each VEX team access to the WPID library and its supplemental materials. The researcher team will take a step back and allow students to explore the library, testing out features and familiarizing themselves with its core functionality. Students may also use the website tutorials and documentation for assistance, but the researchers will not answer any questions or assist in this process. Student teams are expected to use their own robots, and assume any responsibility with handling the robot and any programmed motion safely. The library does have safeguards to prevent some extreme movements, such as limiting a mechanism's motion, but this must be set by the students in order to function. This initial exploration stage will take place over a period of one week. The amount or extent of usage will not be controlled during this stage of the study; the students may decide themselves if they want to solely read the documentation or attempt to use the entire library's functionality. After some exploration and experimentation, the researchers will instruct the students to perform specific tasks utilizing the library by developing a simple autonomous routine. This period will start after the week of exploration, and will take place over one week. After the week of testing, the researchers will ask for a video of the VEX team's robot performing the routine. The research team will also ask for access to the students' program and review the written code. Any amount of completion is acceptable. The final portion of the research consists of an online or in-person interview with each VEX team about how the team approaches programming their robots and about the team's experience with the WPID library. It is expected that this interview will take no longer than 45 minutes.

Risks to study participants: The probability and magnitude of harm or discomfort anticipated in the research are not greater than those ordinarily encountered in daily life or during the performance of routine physical or psychological examinations or tests.

Benefits to research participants and others: Participants will receive educational value from the tutorials on WPID and through the utilization of the library. Topics covered in the tutorials include the PID robotics control algorithm, setup of the coding environment and the WPID resource, and utilization of the library's functionality. The study will also encourage students to practice developing autonomous routines using a custom set of functionalities based on resources they are familiar with. The larger VEX community will be able to access the WPID library and associated website resources upon completion of the MQP project. The MQP resources will retain the same educational value seen during the study, accessible to VEX teams through the Internet.

Record keeping and confidentiality: Only the investigators will have access to the records collected from this study. All interviews are anonymous, with no individual's names tied to any interview response, thus maintaining full confidentiality. This consent form is the only personal identification tying an individual to our study, and will not be shared publicly. Records of your child's participation in this study will be held confidential so far as permitted by law. However, the study investigators, the sponsor or its designee and, under certain circumstances, the Worcester Polytechnic Institute Institutional Review Board (WPI IRB) will be able to inspect and have access to confidential data that identify your child by name. Any publication or presentation of the data will not identify your child, only their interview responses may be presented publicly.

For more information about this research or about the rights of research participants, or in case of research-related injury, contact:
Investigators: Please see the top of the consent form for our contact information.
IRB Manager: Ruth McKeogh, Tel. (508)-831-6699, Email: irb@wpi.edu
Human Protection Administrator: Gabriel Johnson, Tel. (508)-831-4989, Email: gjohnson@wpi.edu

Your child's participation in this research is voluntary. Your child's refusal to participate will not result in any penalty to you or any loss of benefits to which you may otherwise be entitled. They may decide to stop participating in the research at any time without penalty or loss of other benefits. The project investigators retain the right to cancel or postpone the experimental procedures at any time they see fit.

By signing below, you acknowledge that you have been informed about and consent for your child to be a participant in the study described above. Make sure that your questions are answered to your satisfaction before signing. You are entitled to retain a copy of this consent agreement.

______________________________________ Date: ____________________

Study Participant Parent/Guardian Signature

____

Study Participant Parent/Guardian Name (Please print)

______________________________________ Date: ____________________

Signature of Person who explained this study

Appendix C: VEX Team Interview Questions

Background Questions

1.) What is your background in programming? Are you mostly self-taught? Have you taken classes?

2.) What is your approach to programming the robot? What resources do you use to do research when you go to write code for a new season?

3.) Has your team considered using third party libraries before?
If yes: What libraries (if any) has your team used in competition?

4.) Do you use VEXCode Blocks, VEXcodeText, or VSCode? If you use VSCode, are you using VEXCode or PROS? If you are using PROS, are you using OkapiLib? Which language are you using, Python or C/C++?

Control

5.) Have you used any control algorithms? Which ones and why did you choose them? Which one is your team most familiar with?

6.) Were you familiar with PID before this study? What were your initial opinions on PID?

7.) Do you know about other advanced control algorithms? If so, what was your decision-making process to proceed with one algorithm over the others?

Questions Relating to Our Library

1.) What did you find useful about this coding library and website?

2.) Were there any problems with implementing this library?

3.) Did you find the documentation portion of the website helpful?

4.) Did you find the tutorial portion of the website helpful?

5.) What do you think we can do to improve this coding library and website?

6.) Would you consider using this library for future VEX competitions? If you were a team with less experience, would you consider using this library?

7.) Would you like to add anything that we didn't get a chance to touch on?

Appendix D: Roughly Coded Interviews for Pro vs Cons

	PROS	CONS
PROGRAMMER ONE	Code Documentation was a very helpful and useful resource	C++ Syntax was a bit of a challenge
	When asked what could be improved on, stated that "they did not have any issues, understood the concepts well going in, and all topics discussed made sense as a result	Lack of C++ knowledge was the major difficulty, website lacked any C++ resources
	"If I was new to VEX, this is a great first step" but since this user was more advanced, they had more advanced movement algorithms	Felt weird to use internal theta when external sensors were available (not every team has sensors available)
	Did not have any qualms with site layout	Unable to run as perfectly as they would've wanted (perhaps a lack of time to tune the robot)
PROGRAMMER TWO	Documentation helped understand the ideas of the code more	Video was private, not viewable (fixed prior to interview)
	"Will be using this more after the study, will likely be able to score more points with this library."	
PROGRAMMER THREE	code made easy to move	jumpy? causes to go off course
	slow down to stop, excited about the control	
	Easy to understand material	
	No problems with code itself or website	
	"Yeah, will look at [the library next season] once bug is fixed/ use school computers"	