Creating a Library 101

Mark Jolley

It is my personal belief that programmers new to robotics should write their own implementations of the algorithms, objects, and functions if only for the increased level of understanding that it grants.

March 28, 2026

Contents

Architecture Choice

Perhaps the most interesting choice in the creation of a library is the choice of architecture. Architecture in this case is the way in which each of your objects and functions interact with each other. One example of such is where most of the functions of the robot are contained in a single chassis class that has embedded classes for odometry, sensors, motor groups, and other such things that are contained in a chassis.

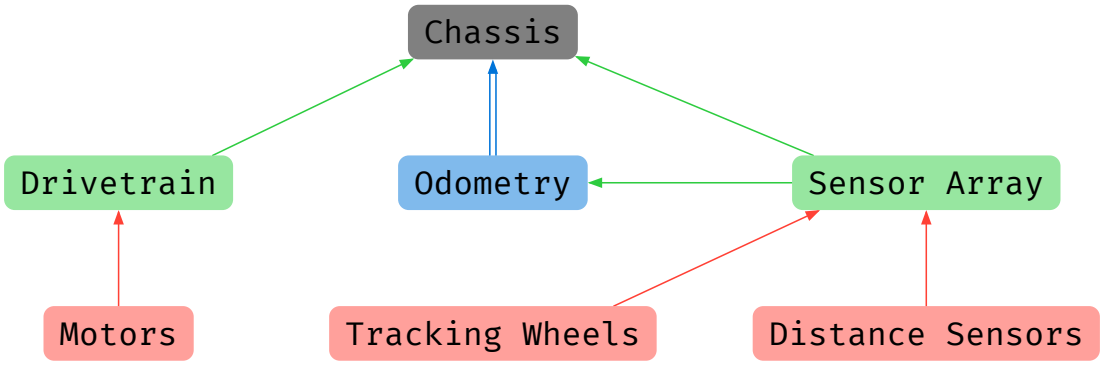


Figure 1: Gray is the complete object, red are physical electronics, green are interfaces for the physical that make it more convenient to work with them, and blue are computational classes that compartmentalize certain computations. Additionally, double arrows indicate that the class it feeds into constructs it instead of it being passed into the constructor.

As you may be able to tell now, your choice in composition is ideally the first choice that you should make when creating a library. While it doesn't change functionality, it is extremely important in the categories of ease of use, simplicity in creation, and simplicity in addition. For example, you could have no classes and simply have 200 odd functions that do everything in your library. It would work and it would be simple to add new features, but it would eventually create a decent amount of technical debt if not handled properly due to extensive amount of global variables that would be required. On the other hand, you could have your chassis class constructors take in every one of the port numbers and configuration details for your robot. This would also be feasible, but would create very irritating constructors. In my opinion, it is best to have distributed responsibility. This means that you construct multiple classes and then pass them to each other and the chassis in order to distribute the configuration and to make it easier to use each component of the chassis outside of the standard structure when necessary. In Figure 1, each single arrow represents a class that is passed into the chassis class or another class as a variable in order to distribute the load. However, I made the choice to have my odometry class be constructed by my chassis class due to the fact that I did not think that it was necessary to have my odometry class be easily accessible.

Levels of Architecture

As you may imagine, each of the classes that make up your robots code have inherent hierarchy. I break this down into four main categories: End Object, Computational, Interface, and Physical.

End Object:

An end object is a class that will be used directly and will not be passed into another class and is not created by another class.

Computational:

A computational class is a class that is created by another class to compartmentalize a set of calculations, variables, and functions.

Interface:

An interface class is a class that takes in the physical class that are typically more annoying to work with and outputs more useful values. One example would be wrapping odometry tracking wheels to provide inches traveled rather than a direct position.

Physical:

A physical class is a class that directly interacts with a port, three wire, or any other well, physical part of the robot.

To be entirely honest, this system is rather arbitrary, yet I do think that it is important to consider when creating a new class in your library. Ideally, the **physical** classes that you use are the ones provided by pros, VexCode, or whatever toolchain that you decide to use. So you don't really need to consider them. However, try to put more effort into the design of **interfaces** or any classes that are directly used, especially **end objects**. This does mean that you can afford to be a bit more sloppy with your **computational** classes given that they are unlikely to be used by anything or anyone outside of the class that they are constructed inside. The reason that this matters is that it helps to create a library that is easy to work with and avoids technical debt, this is rather important.

Polarity between Objects and Functions:

Simply put, the more concentrated the responsibility of the objects are, the more irritating it is to add features to them, yet the simpler it is to use them. However, this is only to a point, too much responsibility to a singular class could cause some annoyance. On the other hand, having everything be a function makes it extremely easy to add new features, yet more annoying to use due to the fact that you have to pollute the global scope more than would be necessary. The choice is yours.

Creating a PID Class

What even is a PID?

Before we get to creating a PID class, we must know what a PID algorithm is. PID stands for Proportional Integral Derivative loop and is a motion algorithm that will output motor voltages in order to hit a desired state.

P Component

1. Will output a value based on the current error to desired point and a tuning variable.
2. The output of this P component will be classified as $\mathcal{P}$ for the rest of this document.

$$\text{Error} = \text{Desired Position} - \text{Current position}$$

Figure 2: The Definition equation used to find the error between the current and desired states of the P component. This is fairly obvious, yet extremely important.

In order to fulfill the proportional standard of the PID loop, the value of the $\mathcal{P}$ is set equivalent to Error scaled by some tuning value in order to be able to calibrate or tune the loop.

$$\mathcal{P} = \text{Error} \cdot \text{kp}$$

Figure 3: The equation for the output of the P component of the loop when kp is some positive tuning value.

The primary purpose of the $\mathcal{P}$ is to output power Proportional to error as well as to jump start the algorithm. This is rather intuitive given that when a person is attempting to get to a point in space as fast as possible, they will run fast at first, but when they get close to the point they slow down to prevent overshooting. This is similar to the point of the proportional component.

Why It Needs Other Components

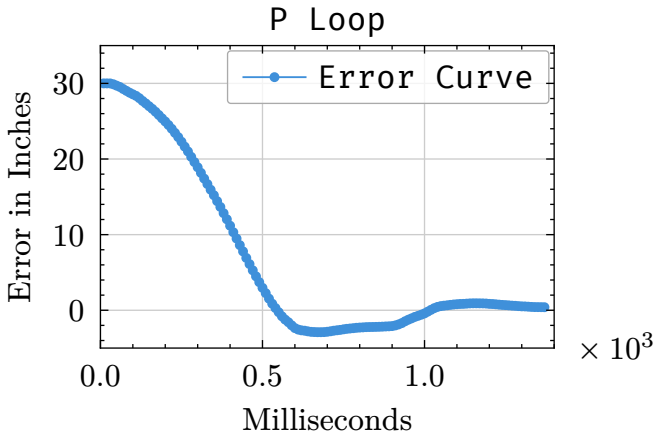


Figure 4: Here is the movement curve of a pure P loop. It overshoots.

D Component

A PID loop's primary damping and overall acceleration limiting function is the the d or derivative portion of the loop. A derivative is defined as a slope at point. In reality though, the encoders inside of the motors that are used do not output a function as to their current position, but a value. Therefore, the derivative or better known as current velocity of the robot would be something similar to this formula.

$$\mathcal{D} = \text{Error} - \text{Previous Error}$$

Figure 5: The D Component's basic output used in a PID loop's output can be defined by the equation above, though it is technically incorrect due to the fact that this is not divided by the amount of time that has passed between the measurements. However, for our purposes that can be compensated in tuning.

$$D = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Figure 6: This is the mathematical formula for a derivative. It works via a limit that takes the slope between ∞ close points.

The $\mathcal{D}$ component limits acceleration and maximum velocity. Simply put, while the chassis' velocity may be positive, the derivative of the error is hopefully negative. This resists harsh accelerations and overshoots.

1. The $\mathcal{P}$ component increases the output of the system which increases the velocity of the chassis.
2. The velocity of the chassis increasing in turn induces the derivative component's damping to the chassis.
3. This limits the maximum velocity and acceleration due to the way in which once the $\mathcal{P}$ term begins to increase the $\mathcal{D}$ term's negative output also begins to amplify, decreasing the output.

Its Affect

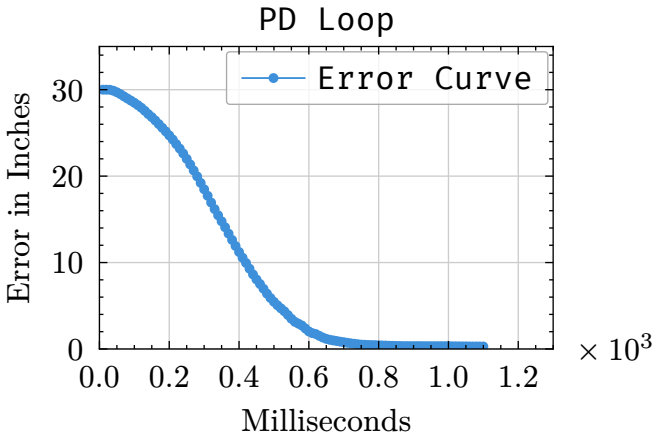


Figure 7: The addition of the D component generally eliminates overshooting and therefore permits for more accurate and consistent motions.

I Component

The $\mathcal{I}$ or integral component is mostly a butchering of its name. Much like the $\mathcal{D}$ component is really just a really small average velocity, the integral is effectively just a Riemann sum. This is to avoid the case where the other two components to the loop are not providing enough power to the system to actually move the state of the chassis. This is honestly best modeled above due to the overall severe deceleration causing a lack of power to reach the goal by the end of the motion. This is fixed by having a component keep track of all error and increment it every time the loop is updated, making it effectively an integral. As an example, that's say that the other two components are outputting 15.2 decivolts to the chassis' motors. This would not move them, but if they stayed there, the slow addition of the $\mathcal{I}$ component would cause that voltage to rise until the system could actually reach the goal state. This is best described by the figure below.

$$\sum_{t=0}^{n} E_t \tag{1}$$

Figure 8: The summation here is a representation of the Riemann sum that substitutes for the integral in this system. As previously described, since there is no possible function for discrete motor readings, a summation instead of an integral is required. In this case, E is the current error, t is time, and n is the number of update cycles until the chassis reaches the desired state.

$$\text{Summation} = \text{Error} + \text{Summation} \tag{2}$$

Figure 9: This is the basic idea of the integral component.

The Output Equation

Once each of these components are identified, they can be unified into a singular equation. Said equation does not include all of the other infrastructure required for a PID loop, such as adding the summation, and setting previous values to such.

$$\text{Output} = (\text{Error} \cdot \text{kp}) + ((\text{Error - Previous\_Error}) \cdot \text{kd}) + (\text{Summation} \cdot \text{ki})$$

Figure 10: The unified equation for power output in a PID loop.

Architecture Choice

Here you have at least in my opinion, two good options and they are listed as following

1. Create a PID class with inbuilt completion feedback.
2. Create a PID class that just does power outputs and create a separate class to handle completion.

The difference between these two options appears mostly in how they operate, for the functions that calculate output are the same. In once case you would have to initialize a separate class to track when the movement is completed instead of adding another function to your PID class that returns a boolean of whether or not the movement is complete. Personally, I prefer the second option, mostly due to the fact that you could later extend it to have further functionality such as motion chaining a good bit more easily.

The PID Algorithm

The basic process should be wrapped into a function that takes in error and outputs a power. As such it should be contained in a class. However, the basic algorithm is as such.

1. Summation += Error
2. Run $\text{Output} = (\text{Error} \cdot \text{kp}) + ((\text{Error - Previous\_Error}) \cdot \text{kd}) + (\text{Summation} \cdot \text{ki})$
3. Set Previous_Error = Error
4. Return Output

This basic sequence makes up what I like to call the update function of a PID class and is called each loop of a movement function in order to update the values.

Improvements to the Algorithm:

Integral Antiwindup: The main issue with the integral term are that if you begin to sum it too early in the motion, it guarantees an overshoot. In order to fix that there are a few possibilities.

1. Set a maximum error amount before the summation begins to be added to. An example of this would be not adding to summation if the error is above 5.
2. Set a maximum integral term. Far less effective in my experience.

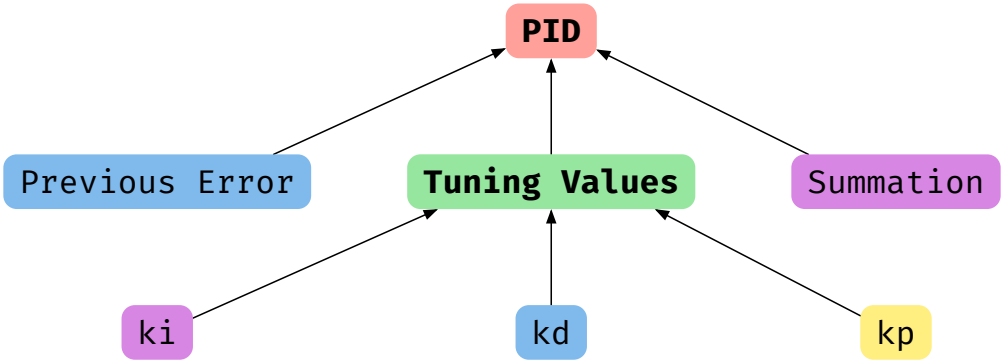


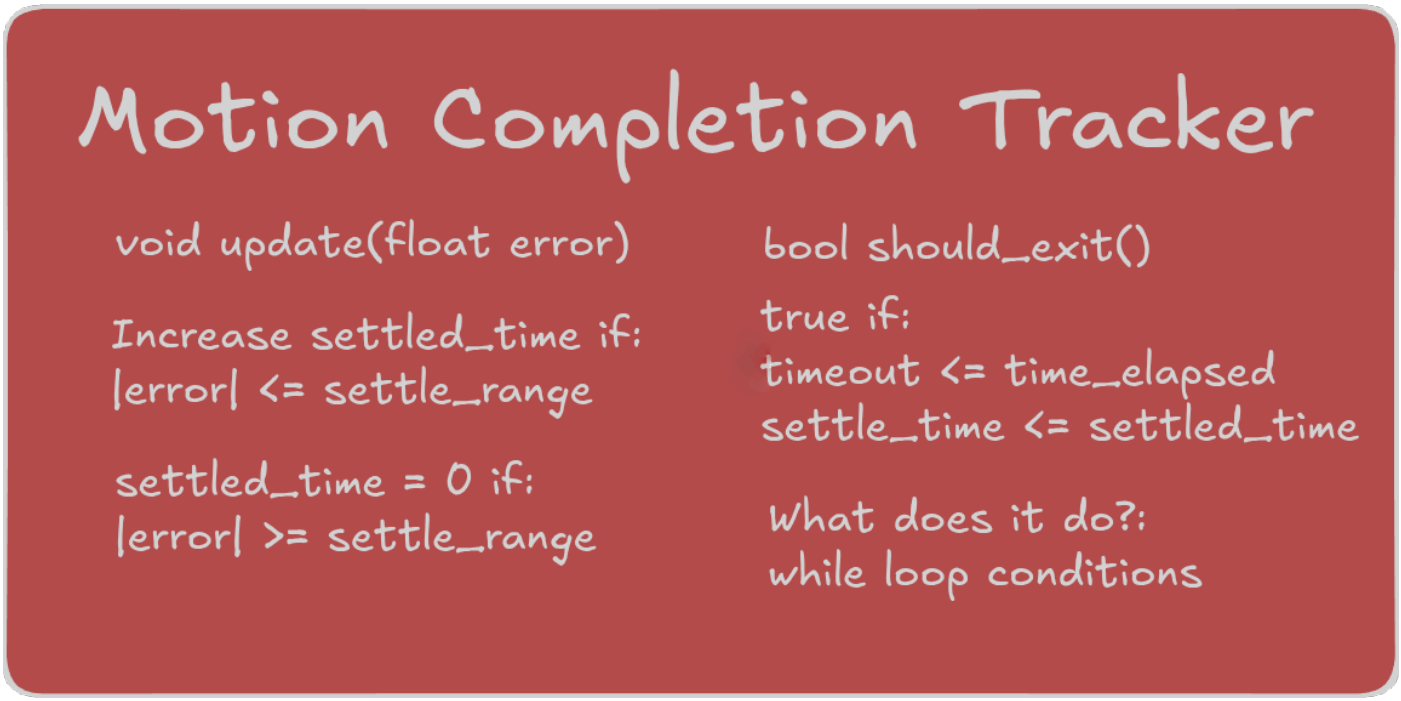
Figure 11: This diagram provides a schema that includes the required variables to be stored in a PID class. This does not include function scope however. For example, you still need an error variable as an argument for your update function, etc.

Creating a Motion Completion Tracker

As previously stated, you can keep the motion completion tracker inside of the PID class, but I personally prefer it to be separate. This is mostly due to my opinion that classes should not have things jammed into them that do not directly relate to them.

Why We Even Need One

A motion completion tracker quite simply exits whatever control loop that it is managing once its parameters are fulfilled. This avoids the use of what I like to call wait PIDs. I have seen a few implementations of the PID algorithm that has the algorithm run in parallel and just wait in autonomous until it is assumed to be complete. This as you may imagine is not that accurate or consistent due to it permitting excessive oscillation. Basically, at the end of a motion, there is a high chance that the PID would have fully settled and ended in a consistent state, but when simply waiting, its back to a partial numbers game of battery voltage stability, smart motor controllers, and drivetrain starting state.



In the above chip, the two primary programmatic functions of a tracker are listed along with their logic. The first is an update function. It should be run at the end of each PID update loop with the error in order to inform the tracker. The second function should be used as a parameter in the while loop the supports the PID. This should be a class due to the fact that it also get spawned as a blank slate, similar to the PID. I would also recommend adding the parameters as a struct that contains the variables of settle_range, settle_time, and timeout. You can also add higher levels of control such as two separate ranges for settle times.

Index of Figures