


Technical Notebook for Programming

Made by AndreiDani





Content

1. Introduction to the Problem
 2. Proportional - Integral - Derivative Controllers
 - a. Integral Windup (limiting the integral)
 - b. Rounding using division
 3. Object Stall Detection (to see if the robot's stuck)
 4. Speed Controllers. Accelerating Functions
 5. Gyro Forwards / Backwards (Moving)
 6. Turn Left / Right -> Arc Turns (Formulas)
 7. Calibrating the constants and parameters
 8. Similarities & Differences between the old and new code
 9. Links (Github, Wikipedia, Desmos, ...)
 10. Further reading (Wikipedia, Github, ...)
- 



1. Introduction to the Problem

For moving we use 4 main functions:

- Move Forwards / Backwards
- Turn Left / Right

But how do we implement them?

a. The first approach


For all functions, we can apply constant speeds and check if the distance or the targeted angle has been reached. Then we stop the robot.

Problems: This approach is correct in theory, but when it is put into practice, the robot may deviate from its actual trajectory because of external factors. (ex. worn-out tires, wobbly table, slippery part of the map, etc.)

b. A new concept

For all functions, we can use a ***Proportional-Controller***. Its role is to fix the current error and make the robot return to its initial path. The error is set to ***(SETPOINT - THE READ VALUE)***. Then the proportional - ***(ERROR * KP)*** is applied, alongside the speed given. This concept is used by a lot of FLL teams.

Problems: This approach fixes the previous problem, but what if the center of mass of the robot with the system is not in the center, but rather on the one side? This will make the robot lean to that side, and therefore go in that direction because this type of controller can't fix this kind of error.



c. Our approach to this problem

For all functions, we use a **Speed Controller**, but for moving forward / backward we also use a **PID Controller** that runs in parallel (at the same time as the **Speed Controller**).

2. PID Controllers

Proportional - Integral - Derivative Controllers (or **Three Term Controllers**) are used when there is an input value (like from a gyroscopic sensor) and a value that it needs to reach.

The first step is getting the error, which is the difference between **A SETPOINT** and **THE INPUT VALUE**. The correction / output is then calculated by the **P**, **I** and **D terms**.

1. **The Proportional** is responsible for fixing the current error. Without this, the response time is slow and will create offsets from the robot's actual path.

2. **The Integral** is responsible for fixing errors that may have accumulated over the past. For example, if the robot's always going to the right side, because of the center of mass, the integral will become higher and higher, until they cancel out.

3. **The Derivative** is used for predicting the next error, using the difference between the current and last error (rate of exchange). This concept is used to minimize the next error.

4. **The Output** is the sum of these values times the constants.
Output = Proportional * Kp + Integral * Ki + Derivative * Kd.

a. Integral Windup

Integral Windup is a modification done to the PID Controller. This makes the integral always be in the range of $[x, -x]$, where x is a constant value (parameter - integralLimit).

Fix: Let's say that the robot is moving forward in a straight line, but then it collides with a mission and gets stuck. This will generate an error that can't be fixed and the integral will reach infinity as time passes. If this happens the program crashes. To make sure this doesn't happen we limit the integral, which also makes the output always be in bounds.

b. Rounding using division

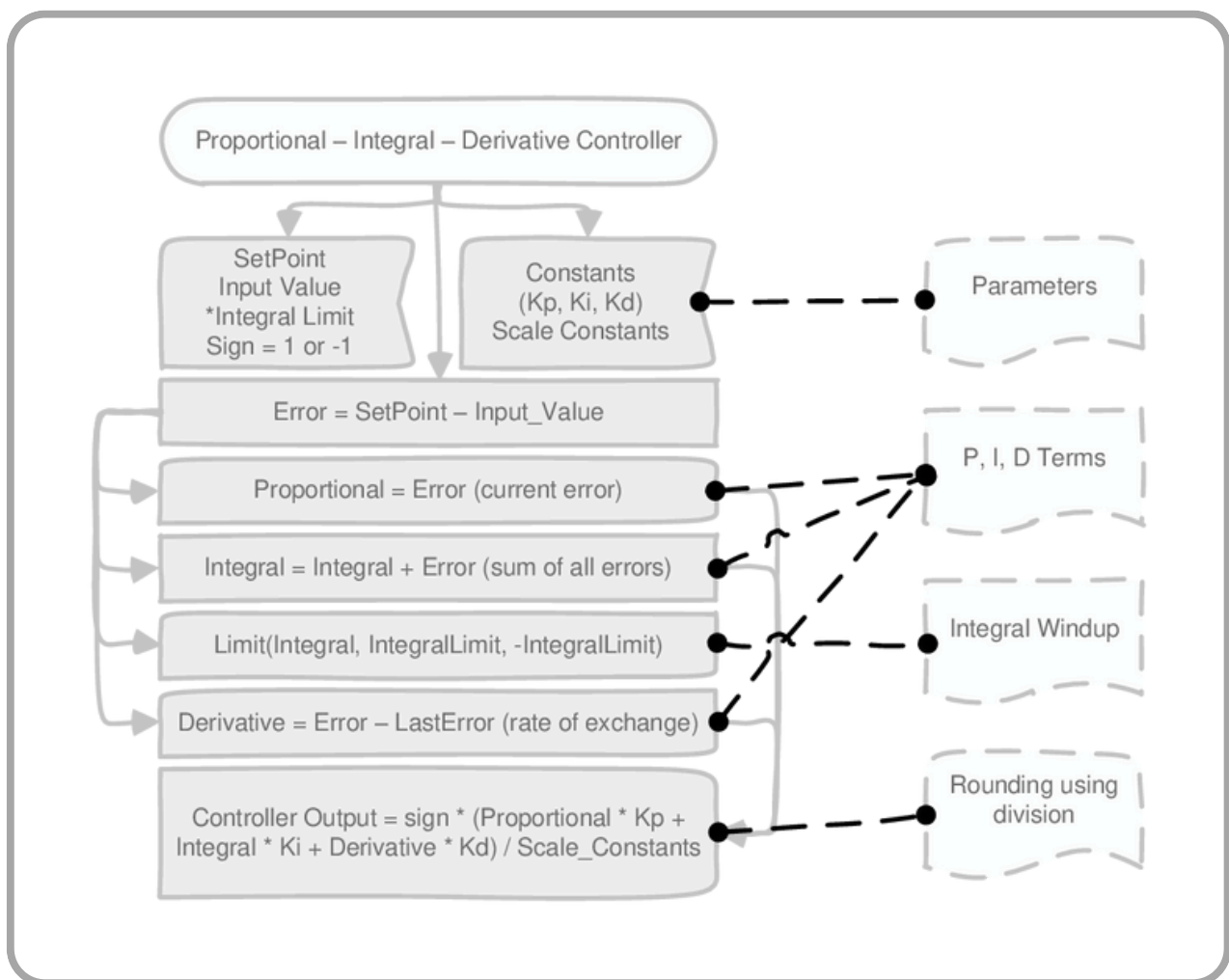
Kp, Ki, and Kd are written as fractions of $x / scale$, where x is the constant multiplied by *the common scale* and *scale* is a parameter in the form of $10 ^ {the number of decimals}$ (10 to the power of the number of decimals).

Output = (Proportional * Kp + Integral * Ki + Derivative * Kd) divided by **Scale * sign**. Because **the terms P, I, and D** are always integers, the final division will approximate the **Output** to the nearest integer, without the need for *extra decimals*.

This optimization makes the **Output** be calculated in less time, because the hub doesn't have to do floating point arithmetics, which are unoptimized, unlike operations with *whole numbers (integers - int)*.

Note: This concept / principle is also used to transform the distance into degrees and in the **Arc Turns Formulas**.

Example: Let's say we have **Kp** = 2.5, **Ki** = 0.002, and **Kd** = 0.75. If we want to write them as fractions they become **Kp** = 25 / 10, **Ki** = 2 / 1000, and **Kd** = 75 / 100. The common scale becomes 1000, **Kp** = 2500 / 1000, **Ki** = 2 / 1000, and **Kd** = 750 / 1000. When we pass them through the parameters they become **Kp** = 2500, **Ki** = 2, **Kd** = 750, and **Scale** = 1000. The *Correction becomes = (2500 * P + 2 * I + 750 * D) / 1000.*

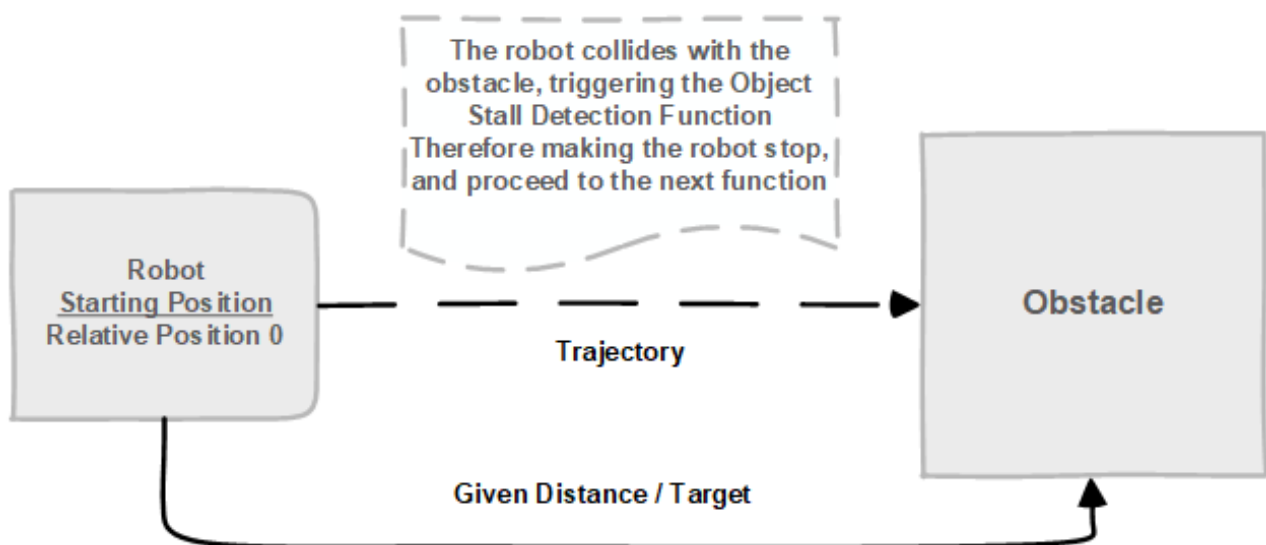


3. Object Stall Detection

Object Stall Detection helps us see if the robot is stuck or barely moving. If this occurs, the robot stops and proceeds to the next function, instead of entering an infinite loop.

This function is implemented using *two relative positions of the robot* and two parameters: an iterator (representing the frequency of calls to this function) and a threshold (defining the difference between the robot moving and being stuck).

If the difference between the two relative positions is smaller than the set threshold the robot says that it's stuck and stops the current function, proceeding to the next.



Note: This is implemented into all functions, but it's not a modification made to the PID Controller.

Note: Object Stall Detection is also used when aligning the robot to a mission / wall in a run.

