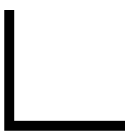


Technical Notebook for Programming

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. Easing and Time Functions
5. Gyro Forwards & Backwards (Moving)
6. Turn Left & Right (All kinds of turns)
7. Arc Turns (Moving in a circle)
8. Tuning the P, I, and D constants and parameters
9. Old code vs. New Code (Similarities & Differences)
10. Links & Graphs (Github, Wikipedia, Desmos, etc.)
11. Further reading (Wikipedia, Github, etc.)

Preview

In this part of the technical notebook, you will learn about Proportional - Integral - Derivative Controllers and Speed Controllers, how they work, and how to tune the parameters in order to make the robot produce constant results.



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 change). 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

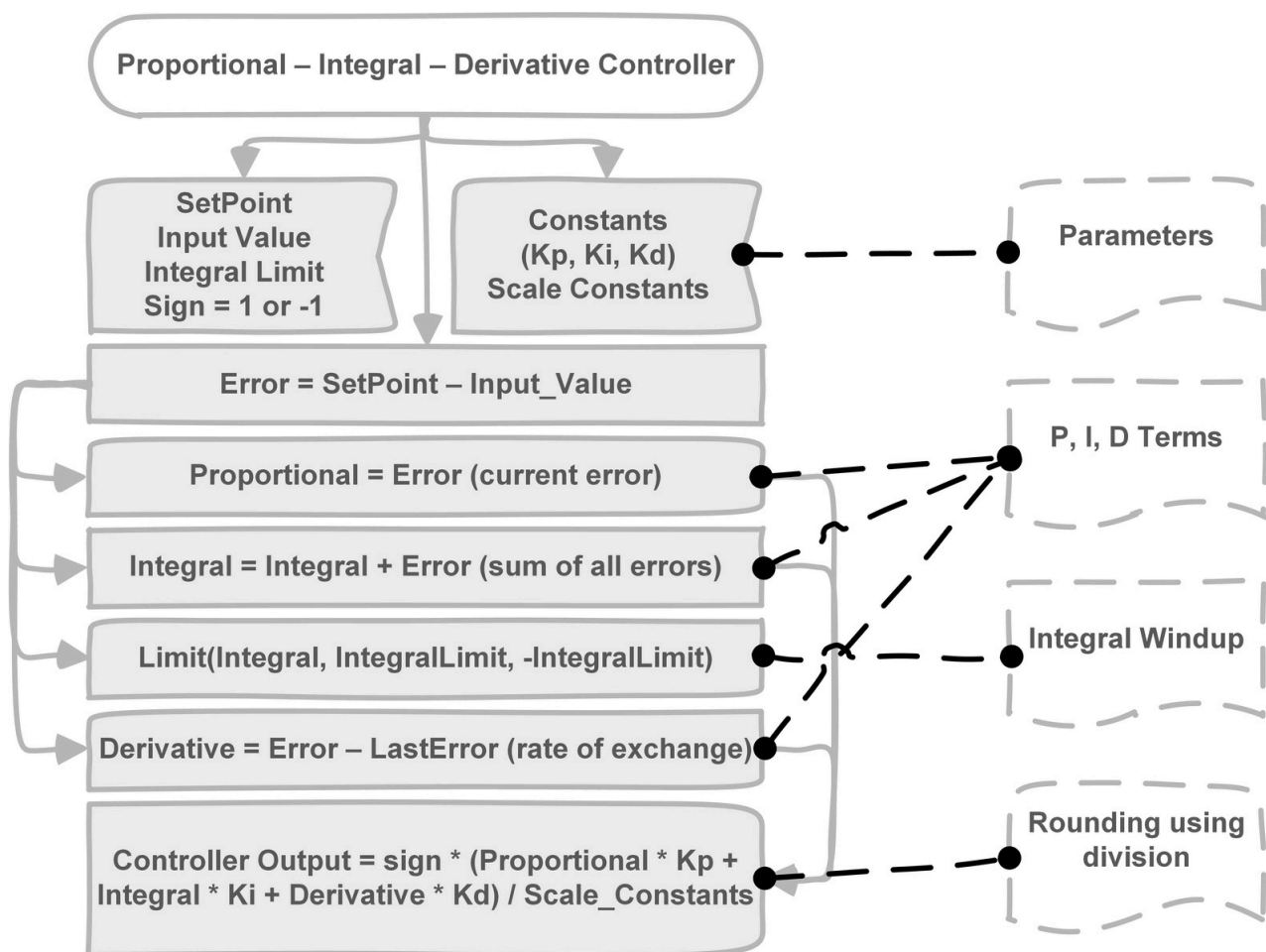
Kp, Ki, and Kd are written as fractions of $x / scale$, where x is the constant multiplied by *the common scale* and *scale* is a parameter in the form of $10 ^ {the number of decimals}$ (10 to the power of the number of decimals).

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 operations, which are unoptimized, unlike operations with *whole numbers (integers - whole num., floats - rational num.)*.

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.



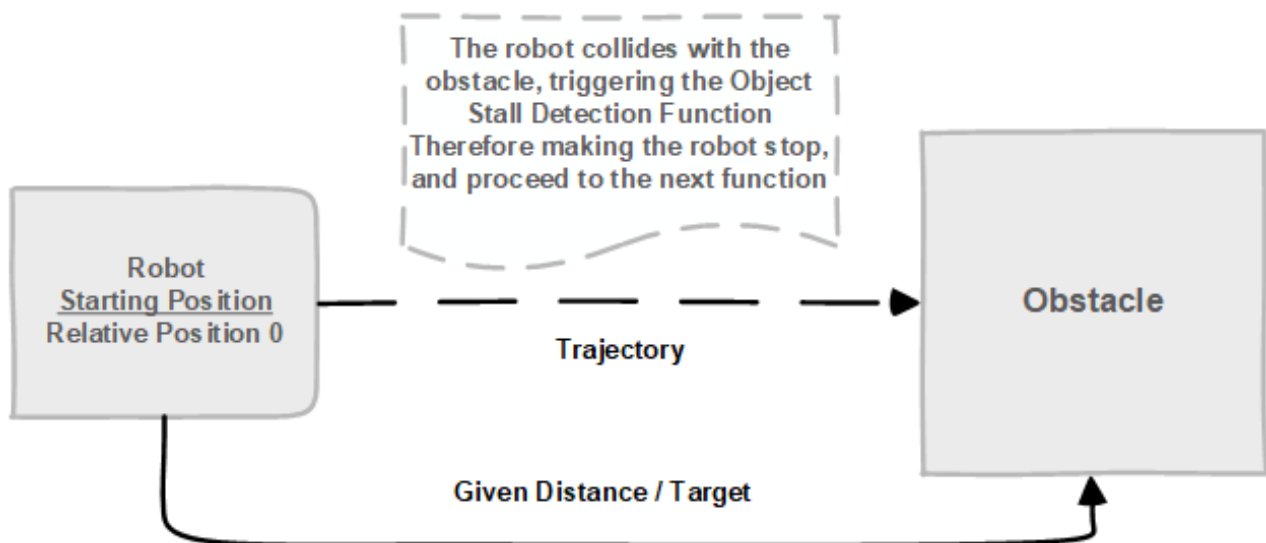
De schimbat scheme logice
Font Roboto (de incercat)

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 and it's a modification made to the Speed Controller.

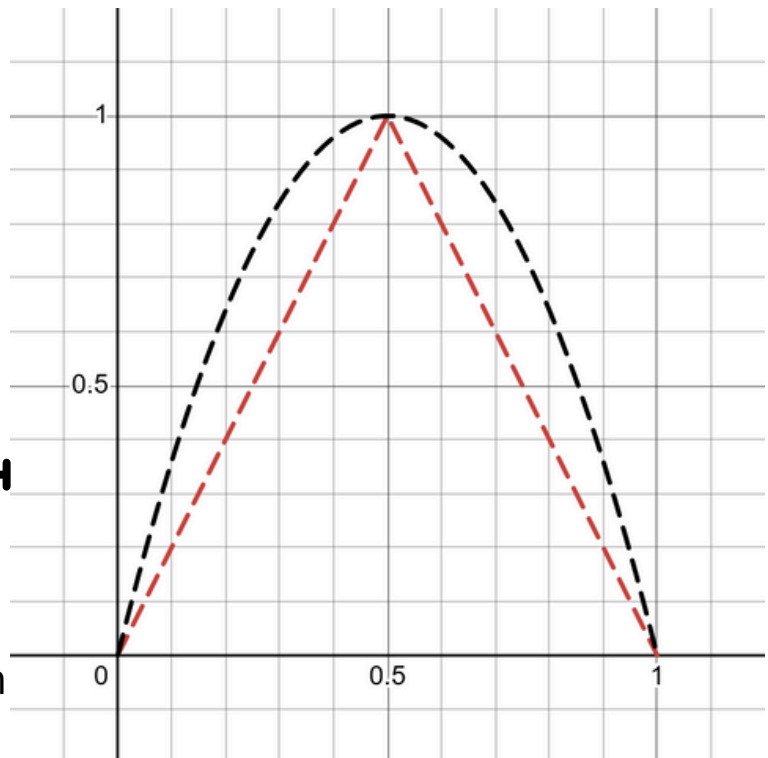
Note: Object Stall Detection is also used when aligning the robot to a mission / wall in a run.

4. Speed Controllers

Speed Controllers are used to dynamically change the robot's speed. This is implemented using an easing function - $F(x)$ that will be multiplied by the speed and a time function that will return the value x (representing how much of the action the robot has completed, ranging from $[0, 1]$).

a. Easing functions

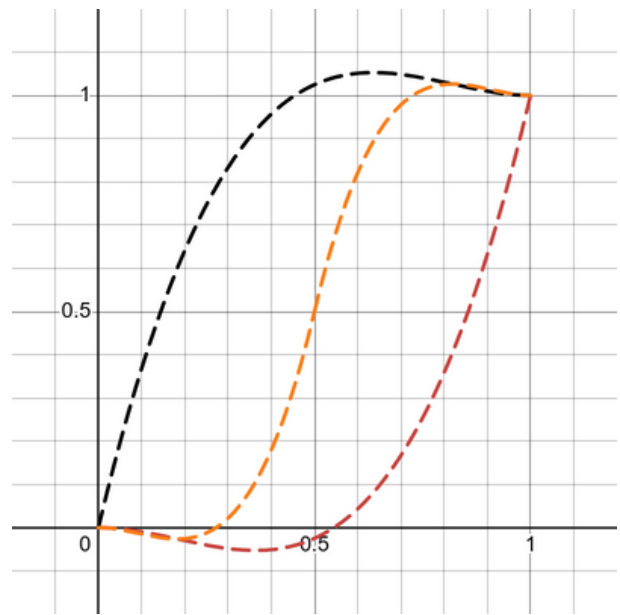
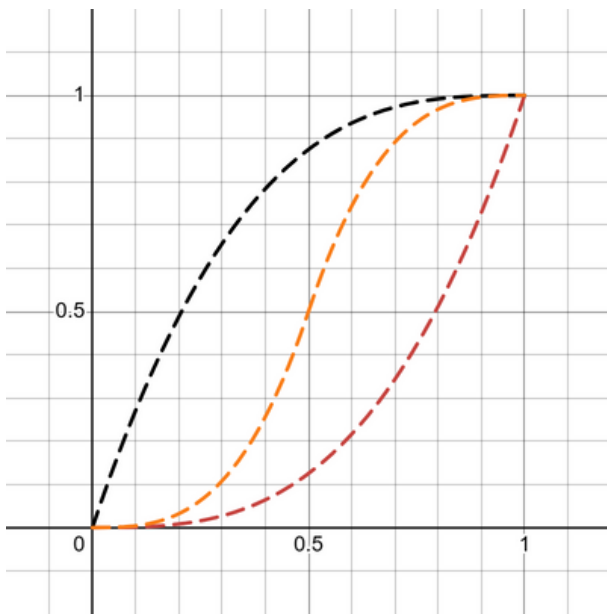
Easing functions specify the rate of change of a parameter over time. The most common function is **Ease Out In**, represented in the following graph.



Fix: With these kinds of functions, the robot doesn't have any sudden changes in its speed.

Ease Out In Linear, Ease Out In Quadratic (Quad.)

- $f1(x) = 1 - \text{abs}(1 - 2x) \{0 \leq x \leq 1\}$
- $f2(x) = 1 - \text{abs}(1 - 2x)^2 \{0 \leq x \leq 1\}$



Ease In, Ease Out, Ease In Out Quad. / Cubic (x^2 or x^3)

- $f_3(x) = x^2 \{0 \leq x \leq 1\}$
- $f_4(x) = 1 - f_3(1 - x) \{0 \leq x \leq 1\}$
- $f_5(x) = f_3(2x) / 2 \{0 \leq x \leq 0.5\}$
- $f_5(x) = (1 + f_4(2x - 1)) / 2 \{0.5 \leq x \leq 1\}$

Ease In Back, Ease Out Back, Ease In Out Back

- $f_6(x) = (c + 1) * (x^3) - c * (x^2) \{0 \leq x \leq 1 \text{ and } c \in [0, 2]\}$
- $f_7(x) = 1 - f_6(1 - x) \{0 \leq x \leq 1 \text{ and } c = \text{a constant}\}$
- $f_8(x) = f_6(2x) / 2 \{0 \leq x \leq 0.5\}$
- $f_8(x) = (1 + f_7(2x - 1)) / 2 \{0.5 \leq x \leq 1\}$

b. Time Functions

Time functions return a value in the range of [0, 1] = [0%, 100%], representing how much of an action has been completed. To determine these values we use two formulas.

$$\text{getTimeDistance()} = (\text{position} - \text{lastDistance}) / \text{distance}$$

This formula is used for Moving Forward / Backward.

1. **Position** = the sum of the motors' relative positions (in degrees, always positive = the robot's position)
2. **Last Distance** = the distance given to the last moving function of the same type (forward / backward), 0 if the previous function is a different type
3. **Distance** = the given distance the robot needs to travel

$$\text{getTimeAngle()} = \text{angle} / \text{target}$$

This formula is used for Turning Left / Right and Arc Turns.

1. **Angle** = the current angle of the robot
2. **Target** = the given angle the robot needs to turn



5. Gyro Forwards & Backwards


Gyro Forwards & Backwards are used for moving in a straight line, for a given distance. This can be done by ***precalculating some values*** at the start of the function (*trajectory, speed change, the transformed distance, and the robot's angle*). Then, we use a ***PID Controller*** to fix the robot's trajectory, and a ***Speed Controller with Object Stall Detection*** to dynamically change the speed and to see if the robot is stuck, minimizing the overall duration of the function.

To see if the distance has been traveled we transform it in degrees using the following formula: $d = (\text{circumference} * d) = (2045 * d) / 1000$ (using **the principle of Rounding with division**). Then we compare the relative position of the robot with the distance in degrees and stop the robot if its position is greater than or equal to the given distance.

6. Turn Left & Right

Turn Left & Right are used to make pivot (one wheel) turns and spin turns (both wheels moving in opposite directions). This is implemented using a ***Speed Controller with Object Stall Detection*** and two parameters: **kLeft** and **kRight**, that act like multipliers for the wheels' speed.

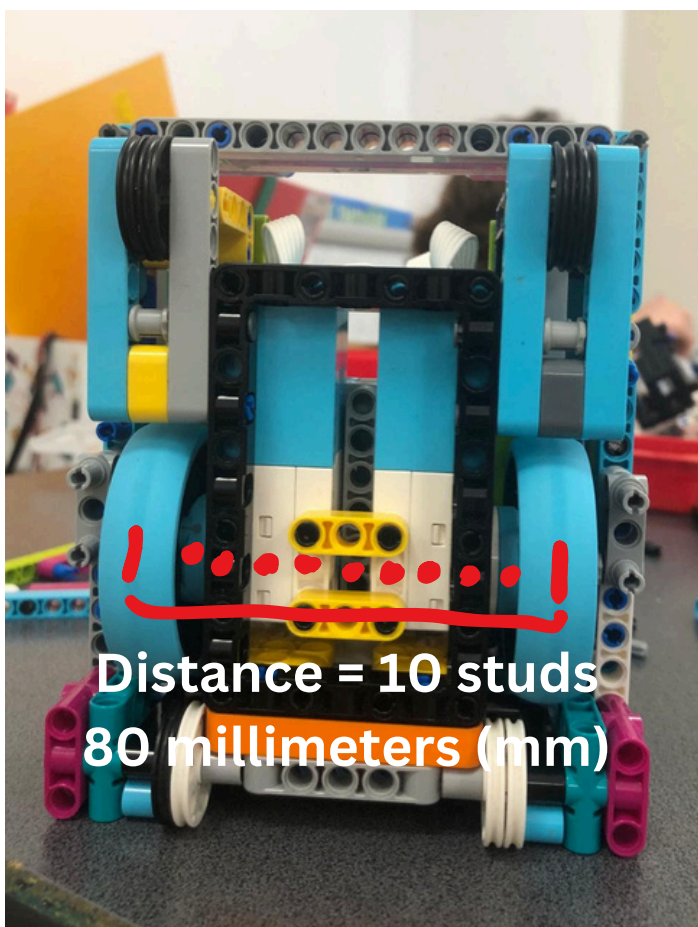
If the parameter is 1 the wheel moves forward, if it's 0 the wheel doesn't move and if it's -1 the wheel moves backward.



7. Arc Turns

Arc Turns make the robot move in a circle, with a radius of r (the distance from the circle's center to the outside wheel).

Arc Turns use the same algorithm, but $kLeft$ and $kRight$ are calculated using the following formulas.



1 Stud = 8 millimeters

$r1$ = distance from the center of the circle to the outside wheel

$$\text{circumference } 1 = (2 * \pi * r) = (314 * 2 * r) / 100$$

$r2$ = distance from the circle's center to the inside wheel = ?

dist = distance from the outside wheel to the inside wheel
 $\text{wheel} = (9 + 1/2 + 1/2) \text{ studs} = 10 \text{ studs} = 80 \text{ millimeters}$

$$r2 = r1 - \text{dist} = r1 - 80 \text{ mm.}$$

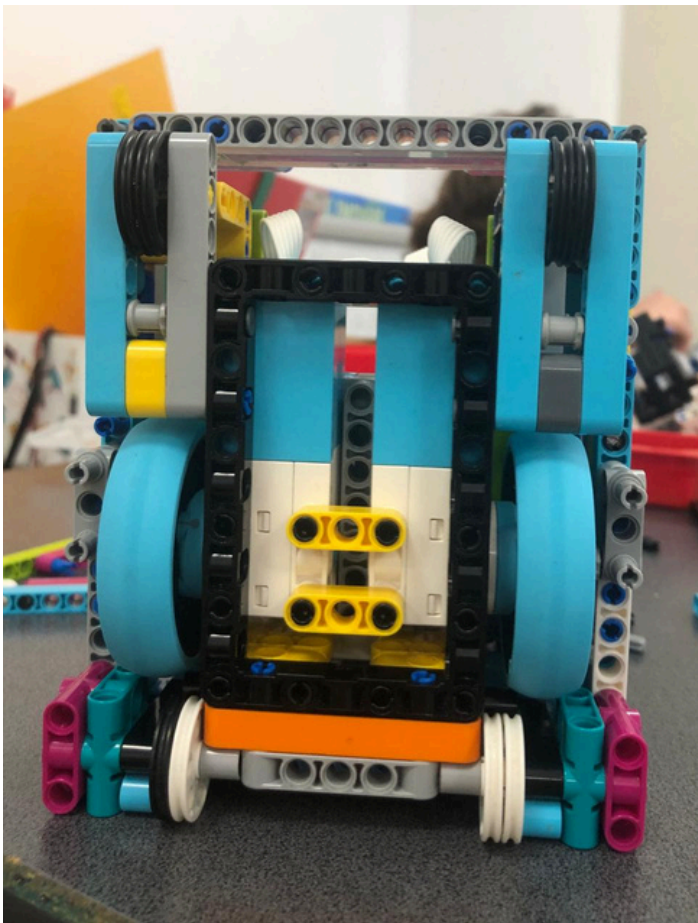
$$\text{circumference } 2 = (2 * \pi * r2) = (314 * 2 * r2) / 100$$

$kOutside$ becomes 1000 and **$kInside$** = $(2 * \pi * r2) / (2 * \pi * r1)$
 $= r2 / r1 = (r1 - 80) / r1 = \textbf{(radius - 80) / radius.}$

In the **Speed Controller**, the final output needs to be divided by 1000 to make the speed be in the range of $[-1100, 1100]$.

7. Arc Turns

Arc Turns are used to make the robot move in a circle, with a radius of r (the distance from the center of the circle to the outside wheel). This is implemented the same way that Turns are, but $kLeft$ and $kRight$ are calculated using a formula.



1 Stud = 8 millimeters

$r1$ = distance from the center of the circle to the outside wheel

circumference 1 = $(2 * \pi * r1)$ = $(314 * 2 * r1) / 100$

$r2$ = distance from the circle's center to the inside wheel = ?

dist = distance from the outside wheel to the inside wheel = $(9 + 1/2 + 1/2)$ studs = 10 studs = 80 millimeters

$r2 = r1 - \text{dist}$

circumference 2 = $(2 * \pi * r2)$ = $(314 * 2 * r2) / 100$

If the outside wheel is the left wheel **$kLeft$ becomes 1000** and **$kRight = (\text{circumference2}) / (\text{circumference1}) = (2 * \pi * r2) / (2 * \pi * r1) = r2 / r1 = (r1 - 80) / r1 = 1000 * (\text{radius} - 80) / \text{radius}$** otherwise if the outside wheel is the right wheel **$kRight$ becomes 1000 and $kLeft = 1000 * (\text{radius} - 80) / \text{radius}$.**

In the **Speed Controller**, the final output needs to be divided by 1000 to make the speed be in the range of $[-1100, 1100]$.

8. Tuning the parameters

Tuning the constants P , I , and D and the parameters is an important process in producing constant results and scoring the maximum amount of points, in the smallest amount of time.

a. The PID Constants

To tune the constants K_P , K_I , and K_D we use the following steps:

1. K_P , K_I , and K_D all become 0.
2. K_P is set to a random value (like 0.75), and then we increase / decrease it until the robot's direction oscillates.
3. K_I is then set to a small value (like 0.001), and then we increase / decrease it until the path becomes somewhat straight.
4. K_D is set to a big value (like 5), and then we increase / decrease it until the path becomes straight (a must when the robot's trajectory isn't good enough after tuning K_P and K_I)

The following terms are determined by the set constants:

1. **Rise Time** = the time it takes for the error to become 0
2. **Over Shoot** = that time when the error becomes the same as the **Steady-State Error**, but then it overshoots
3. **Settling Time** = how long it takes for the error to become constant (for the robot's trajectory to become straight)
4. **Steady-State Error** = the difference between the error when the robot's path stabilizes and the desired error (in our case 0)

***Note:** that the constants affect the terms in the following ways:*

1. K_P is used to decrease the Rise Time
2. K_I is used to eliminate the Steady-State Error
3. K_D is used to reduce the Settling Time and Over Shoot



b. The rest of the parameters

The speed, the object stall detection frequency and threshold, the acceleration scale, and the easing method can be tuned using a ***Trial-and-Error Method*** to see which values work the best.

To simplify this process we have some preset values, that are modified if the result isn't what we wanted. The values are:

1. The Speed - in the range of [200, 1000], usually around 400.
2. The object stall detection frequency and threshold -
 - a. For pushing against a mission = (freq. 2000, threshold 5)
 - b. For stopping when colliding = (freq. 500, threshold 5)
3. The easing method - Ease Out In for most actions and Ease Out Quad for returning in the base
4. The acceleration scale - usually 100, 200 for big distances

Note: Some of these values may change, depending on your robot's structure (ex. the object stall detection threshold).

9. Old code vs. New Code

The new code uses the same principles as the old code, but they are now far more developed and optimized.

