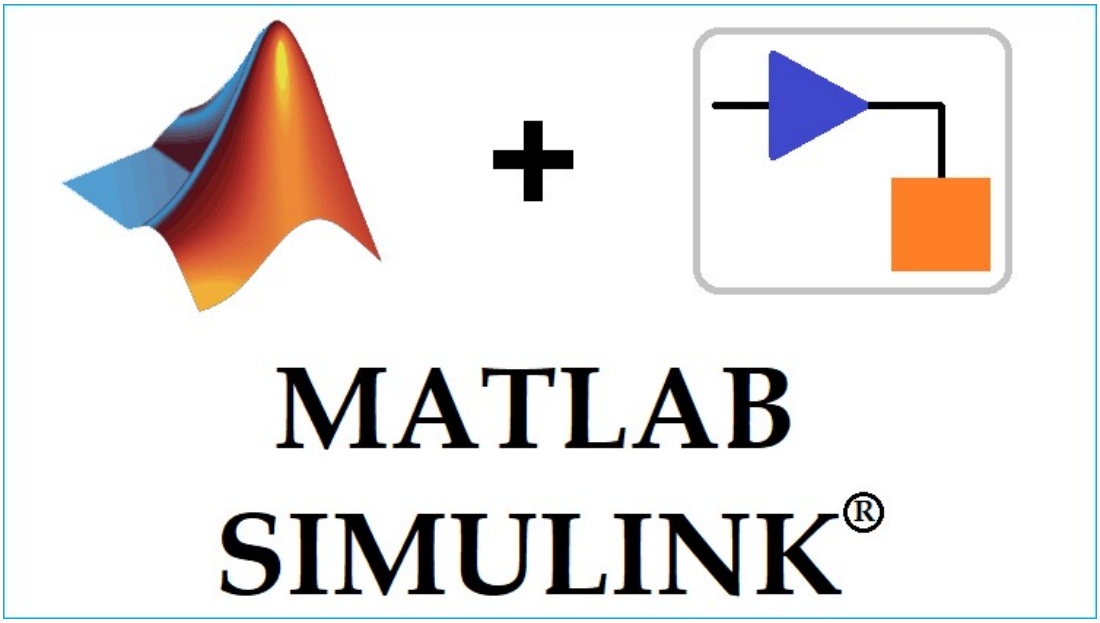
FCS assignment



Semester 3 Embedded Systems

Assignment 1

Sept 10, 2022.

Student: Andre Sanao

Course: Technology

Contents

Abstract

In this assignment, the objective is to design PID feedback loop in MathLab Simulink. After the designing, the performance of the PID I simulated to the correct values to have a smooth and robust graph. These readings will be later used in further assignments of the project.

Introduction

This assignment is applying the learning outcome of the closed loop system. PID is just a form of feedback. It is a type of controller that corrects error from past, present and future. This feature is to satisfy most of the control problem. That is why PID is the most common form of feedback system control across a wide range of hardware applications. The design of the PID Feedback Controller is shown in figure 1.

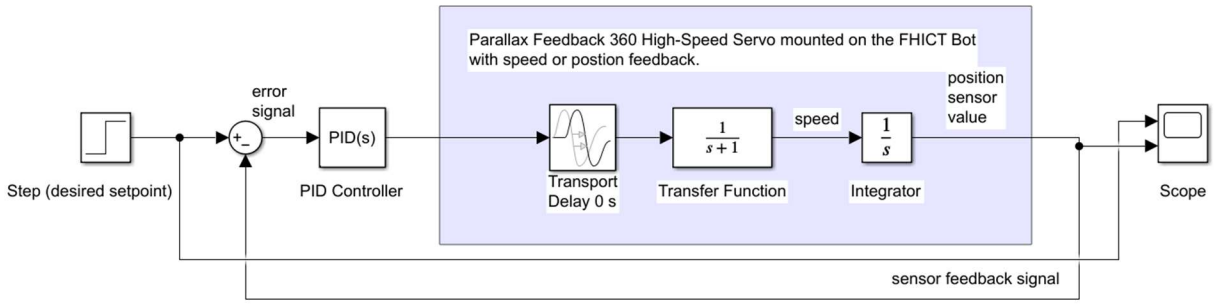


Figure 1 PID Feedback Controller design

Assignment

The first step is to design the PID feedback loop in Simulink and determine P, I and D values in Simulink. Using the Excel sheet 'ParallaxStepResponse.xlsx` and the step response data are needed to determine the values of τ and θ of the Parallax Feedback. For these 2 values to be determined figure 2 is used as way to visualize the formula.

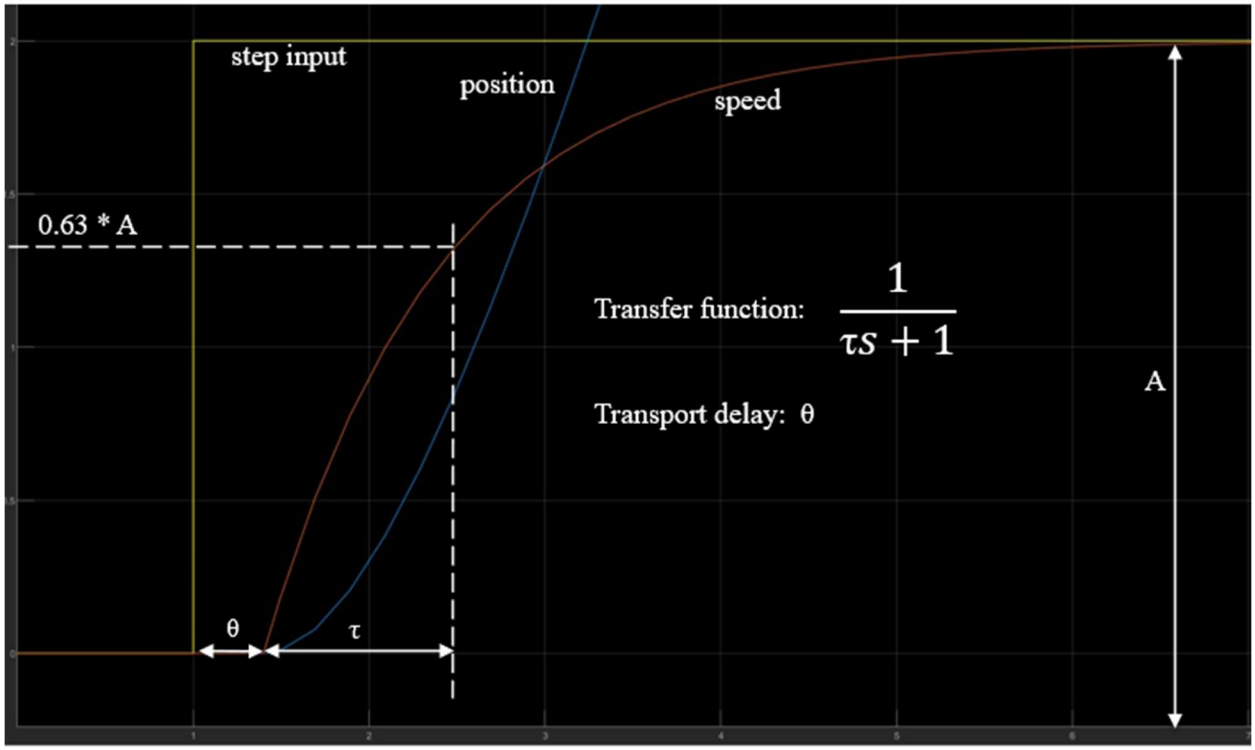


Figure 2 Step response to transfer function

For τ which is a value that is use in the Transfer function, one of the steps is $0.63 \times A$ is. A is the height of the position unloaded (Max Speed) in figure 3 which is roughly around 2.1. So the answer is $0.63 \times 2.1 = 1.323$ ticks. With this value in the spreadsheet, the position of the ms is founded which is 128ms. τ is 128ms minus the speed loaded(0) where its starts gaining speed which is 41ms. $\tau = 128 - 41 = 87ms$.

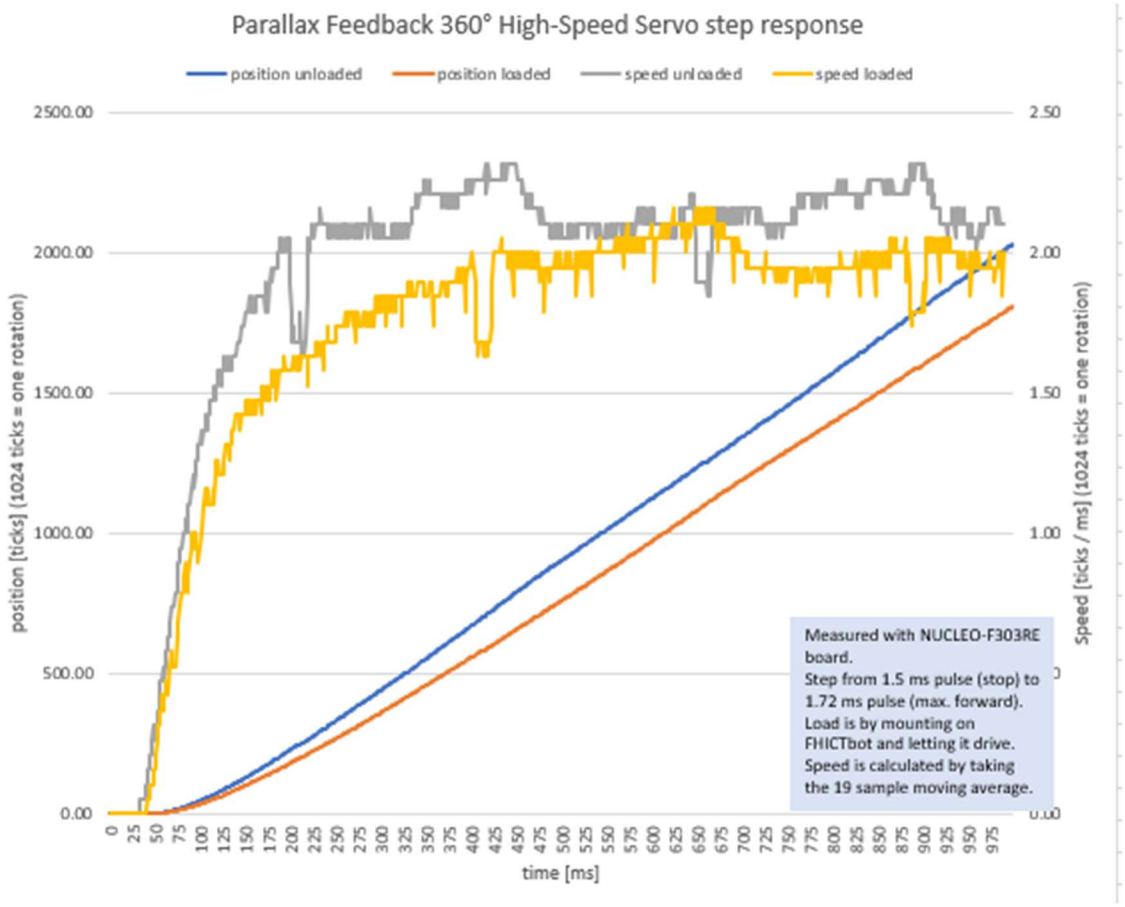


Figure 3 Parallax Feedback 360° High-Speed Servo step response

Now that the Transfer function and the Transfer delay has been determine, the values of the P, I and D of the PID Controller will be calculated using Ziegler-Nichols heuristic closed loop method to find appropriate P, I and D values manually as explained in the presentation (figure 4).

	kP	kI	kD
P	0.5*kU	0	0
PI	0.45*kU	0.54*kU/pU	0
PD	0.8*kU	0	0.1*kU*pU
PID	0.6*kU	1.2*kU/pU	0.075*kU*pU

Figure 4 Ziegler-Nichols open loop method

Ziegler-Nichols method

For the calculations, PID values need to start at 0. First step is adjusting the Proportional(P) to get a stable sinus line. The kP value with stable sinus(see figure 5) is 21.038 and this will be recorded as kU for the formula 0.6*kU which is 0.6*21.038.

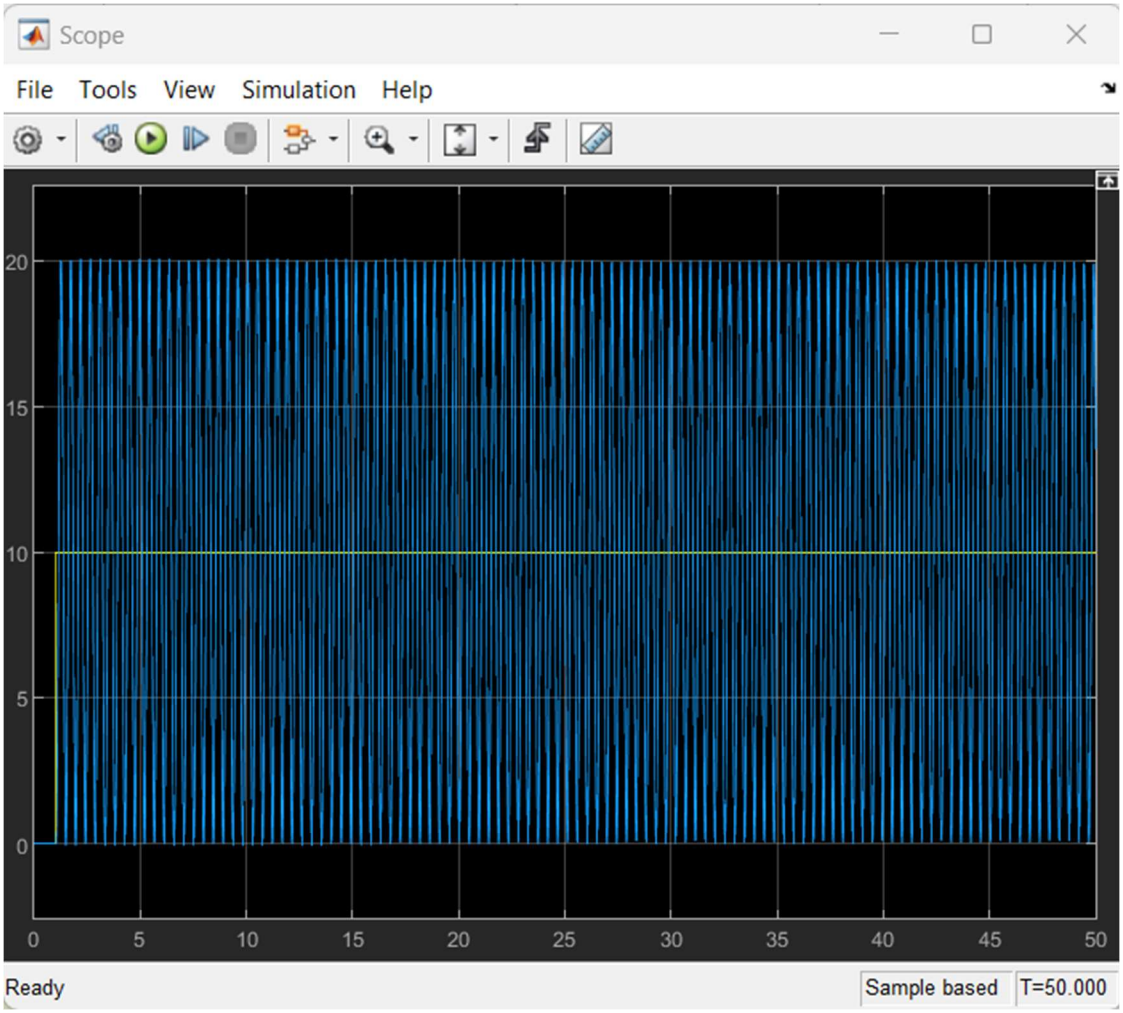


Figure 5 Proportional value

Second step is tuning the Integral(I). The formula for integral is 1.2*kU/pU=kI. kU is already calculated during the tuning of the Proportional which is 21.038. To get pU, the oscillation period needs to be measured. That is the time it takes to do a full cycle from a point back to itself. The period can be measured in the scope simulation (see figure 6). The value of a full cycle is 468ms which can be converted to 0.468s. This value will be used in pU.

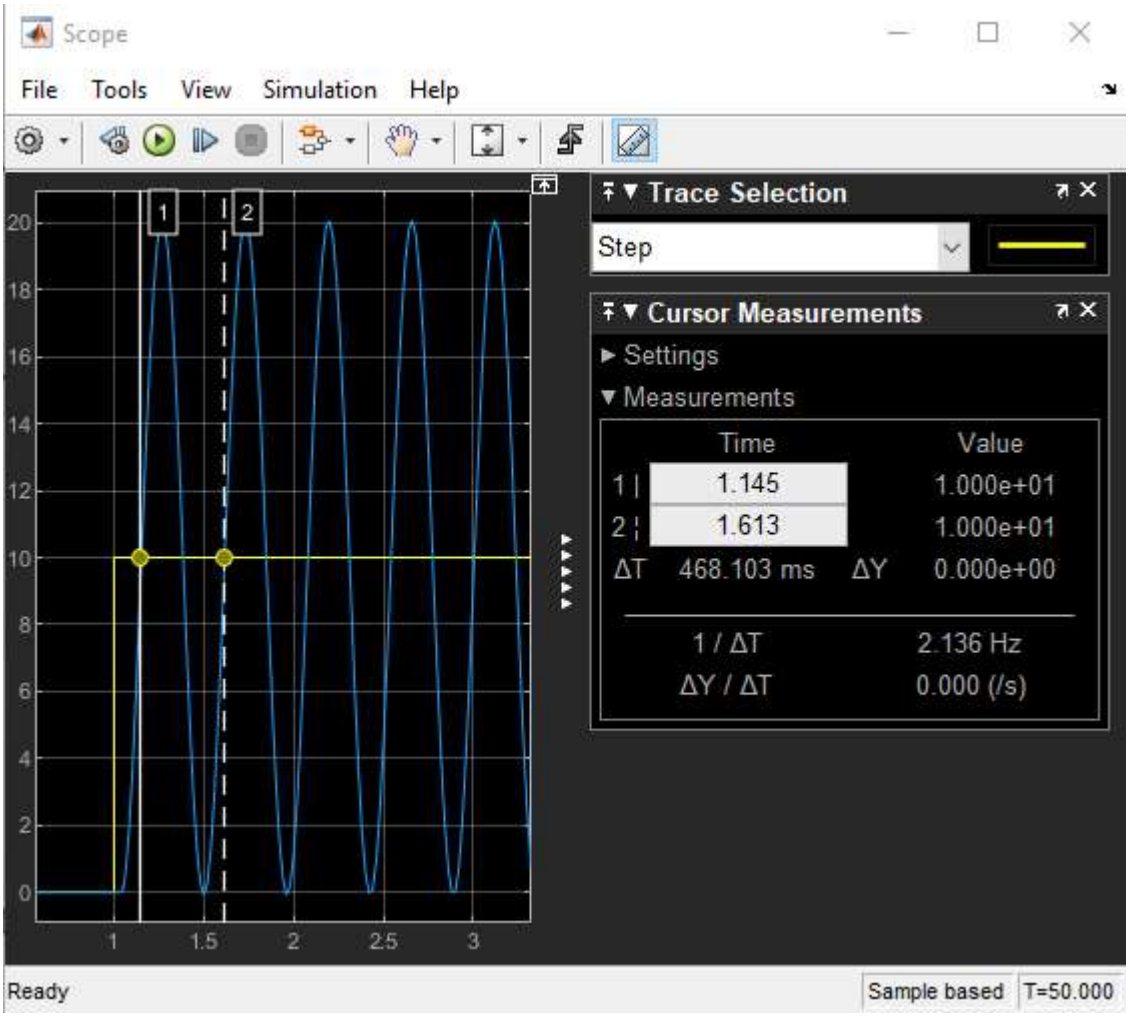


Figure 6 Oscillator full cycle measurement

Now that all the values have been taken, the formula can be calculated. Integral is 1.2*21.038/0.468= 53.944 and this value is set in the PID controller (see figure 7).

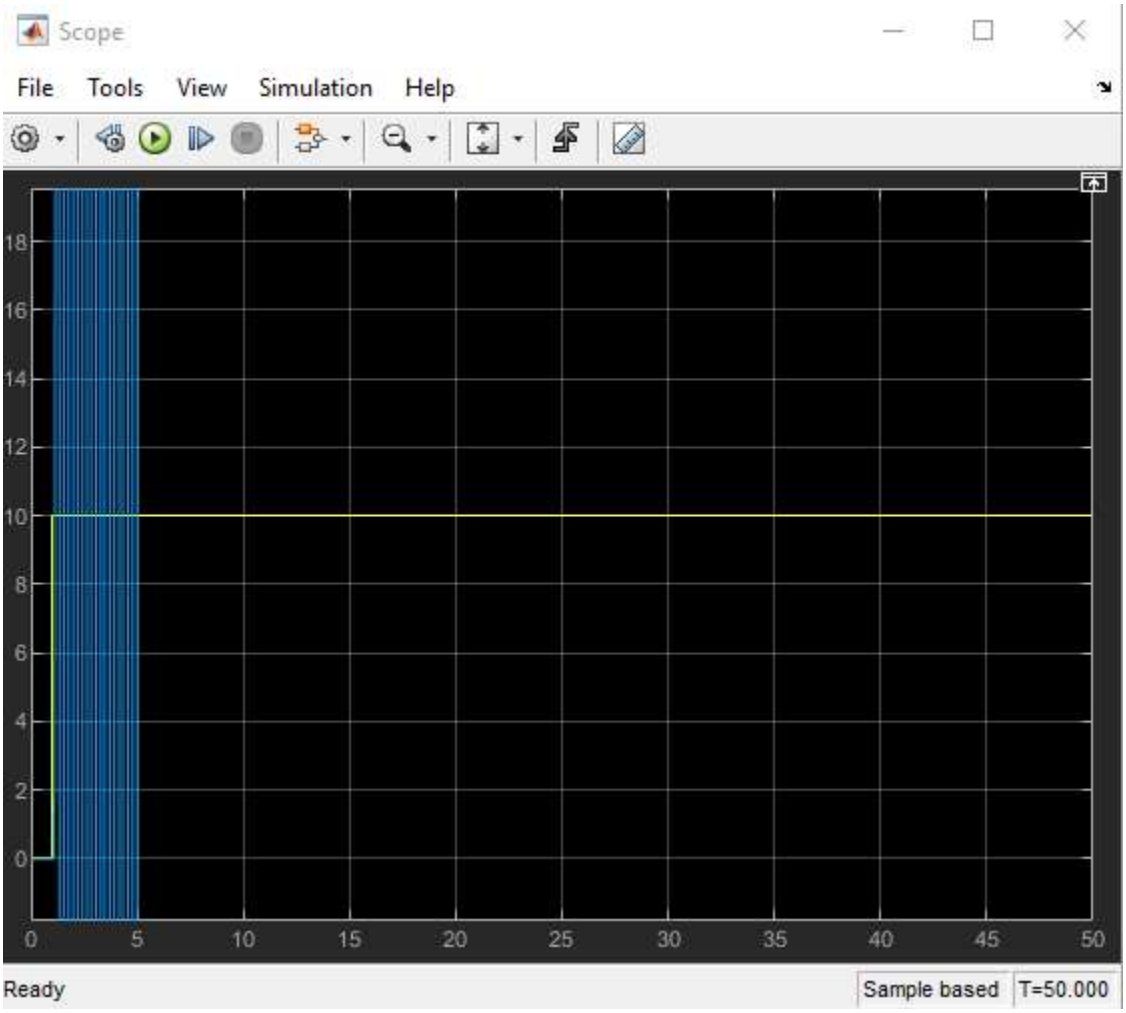


Figure 7 Integral value added to the PID Controller

The third part is the value for the Derivative(D). The formula for Derivative is 0.075*kU*pU=kD. Luckily these values are already taken and the formula can be calculated. Derivative is 0.075*21.038*0.468=0.738. Unfortunately the oscillator period stayed the same because the Ziegler-Nichols closed loop method calculation does not measure the speed.

Tuning option

While testing with the Tuning option, live data were captured in the scope that would give a smooth and steady graph that is shown in figure 8. In figure 9, the simulated tuned values of the PID Controller are shown.

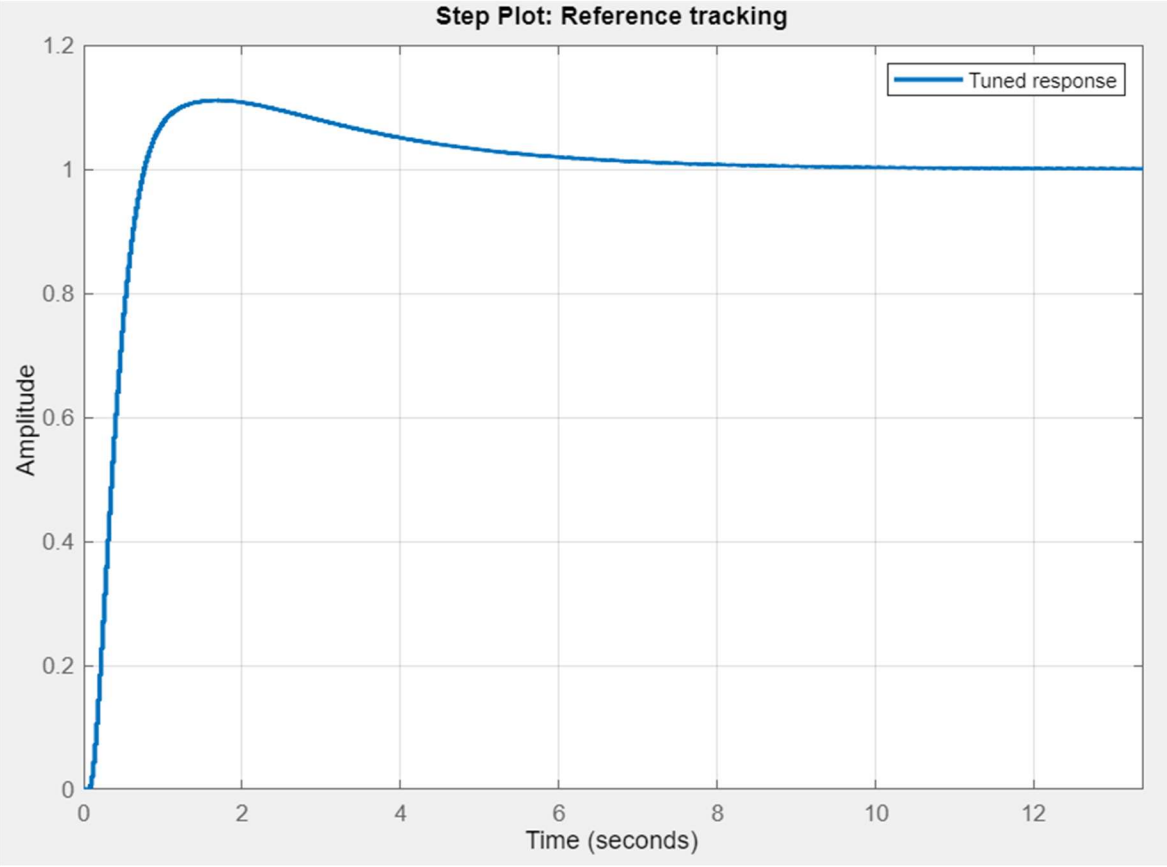


Figure 8 Auto tuned reading of the PID feedback controller

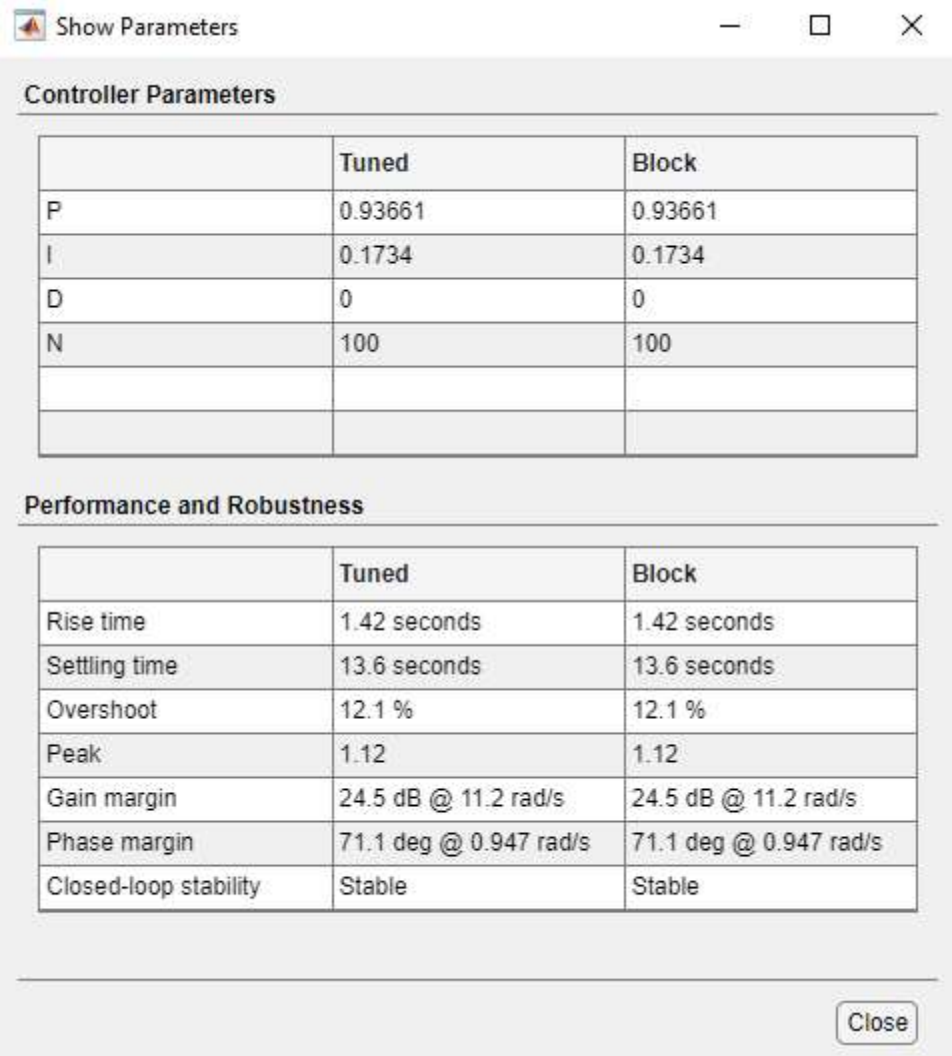


Figure 9 configurations of the PID Controller

During the testing simulation, some problems were encountered and better understanding of how the PID controller values. The Proportional(P) and the Integral(I) values were increasing the overshoot when their values risen but having a lower Derivative(D) will balance out the overshoot creating a fixed height in the scope. These values are far from the Ziegler-Nichols closed loop method values but the difference is the speed is removed from the equation because the method does not require it.

Conclusion

Further tests were not used but during the simulation, the stability was achieved without the use of the Ziegler-Nichols closed loop method. The PID Controller was tuned in a way that makes the system show clean graphs.

Pseudo code

The pseudo code will contain about each component and how they work together. These components will be put into practice later the project. The PID will contain a lot of constant calculating for using low level coding. The outcome of this project will be a robot calculating the distance with speed.

When we initialize stop action in programming, it normally executes right away but in the real world this logic does not work given the factors of physics. This project contains momentum influenced by speed, battery voltage an many other factors. The code needs to adapt with the factors of the real world to run without errors.

The following procedure will contain variable names that has no value because it is a pseudo code. It needs to build without errors and is going to be used in the future.

What is known in the PID controller is that it calculates the error of the current measured value and the goal for the measured value to reach.

So the code will start with something as simple as:

$$error = desired\_speed - measured\_speed$$

The outcome of the `error` can be positive and negative. If the robot creates an overshoot due to speed, then the `error` value will be negative. When having a negative value, the robot needs to decelerate to remove the overshoot. For a nice and smooth deceleration that the Proportional value provides, the `power` of the motor could be simply set as equal to the `error`.

$$power = error$$

But having this line will only create a loop of never-ending overshoot and overcorrect. We learned in the earlier section that we can use `kP` to increase until the oscillation is steady and continuous. We can use `kP` to multiply by error. For now this is the code and later it will be tuned.

```
error = desired_speed-measured_speed;
power = error*kP;
```

Now that the Proportional part of the code has been added, it will be nicer if the code is set into a function. This function will be continuous as the robot will need to calculate every other millisecond.

```
void myPID(int desired_speed)
{
    desired_speed = 0;
    int error;
    int measured_speed;
    int power;
    int kP;

    while(true)
    {
        error = desired_speed-measured_speed;
        power = error*kP;
    }
}
```

The Proportional will not be sufficient to control the power of the robot when the error becomes smaller. The Integral will remove this by increasing the power slowly. Integral will be looking into past values that the robot has created. There will be a delay in the loop for the current error. The Integral will be the sum of the previous cycles of the value and will be added to the error to create new Integral values for future errors.

$$integral \mathrel{+}= error$$

This line of code is still not perfect because having an external influence will not be calculated. This will create huge values that would be heavy for the system to control. A higher value in the `integral` will indicate an external influence that will slow down the system. To combat this problem, more extra power will be added to give some sort of boost to the system. To account for the scaling value of the `integral`, just like with the Proportional term, `kI` will be added to manage the huge values of the `integral`. So `kI` is expected to have small values to contain the fluctuations of the `integral`.

This is the pseudo code so far after adding the Integral portion:

```
while(true)
{
    error = desired_speed-measured_speed;
    integral += error;
    power = error*kP + integral*kI;
    HAL_delay(20);
}
```

A delay of 20ms is used because the motor that is going to be used is a servo motor and it only detects value changes as fast as 20ms.

Now that the pseudo code has made it this far, there will be situations where our `error` reaches 0. That means the `desired_speed` has been met and more added power will be a nuisance in a situation where it is not needed. To make up for this, the `integral` should be set to 0 every time the `desired_speed` has been met.

The next situation is `integral` windup. This is when the `desired_speed` is changed drastically from 0 to 100%, causing the `integral` to start calculating for huge `error` values. To fix this, the `integral` should reset to 0 when the `error` is too big.

The pseudo code should look like this when the solutions are added:

```
while(true)
{
    error = desired_speed-measured_speed;
    integral += error;
    if(error == 0 || error == desired_speed || error == huge_value)
    {
        integral = 0;
    }
    power = error*kP + integral*kI;
    HAL_delay(20);
}
```

So far the code for controlling the output power based on the current and past errors have been made. Now looking it is needed that the robot should be looking for future errors. This is the Derivative. The Derivative will be looking at the rate of the errors. It needs to know how fast the robot is getting to the `desired_speed`. The Derivative will contribute with larger values for greater speed. It is typically outweighed by Proportional and Integral components. If there is a situation where the robot is going faster for some other reason, the Derivative component will become larger and the output power will be reduced and the opposite will happen if the robot is going slower. To calculate this error the code will look as follows:

Just like the Proportional and the Integral components, the Derivative will have a `kD` to compensate for the issue with scaling. Values that is too high will cause instability with the power contribution from the derivative overpowering the power temporarily. Values that are too small will cause the Derivative component that m ay as well not exist. To add the output power of the Derivative component. The code will be as follows:

```
while(1)
{
    error = desired_speed-measured_speed;
    integral += error;
    if(error == 0 || error == desired_speed || error == huge_value)
    {
        integral = 0;
    }
    derivative = error - prev_error;
    prev_error = error;
    power = error*kP + integral*kI + derivative*kD;
    HAL_Delay(20);
}
```

With a little tweaking, the code is more flexible and can easily be used by inputting value in the function. The final code will be the following:

```
void myPID(int16_t desired_speed)
{
    desired_speed=0;
    int16_t error=0, prev_error=0;
    int16_t kP=0, kI=0, kD=0;
    int16_t integral=0, derivative=0;
    int16_t measured_speed=0;
    int16_t power=0;
    int16_t huge_value=0;
    uint32_t time_elapsed, delay_time;
    delay_time = HAL_GetTick() + 20;

    while(1)
    {
        time_elapsed = HAL_GetTick();
        if(time_elapsed >= delay_time)
        {
            error = desired_speed-measured_speed;
            integral += error;
            if(error == 0 || error == desired_speed || error == huge_value)
            {
                integral = 0;
            }
            derivative = error - prev_error;
            prev_error = error;
            executePID(&error, &kP, &kI, &kD, &integral, &derivative, &power);
            controlServo(&power);
            time_elapsed = HAL_GetTick();
        }
    }
}

void executePID(int16_t* error_value, int16_t* p_value, int16_t* i_value, int16_t* d_value, int16_t* integral_val, int16_t* derivative_val, int16_t* power_val)
{
    *power_val=(*error_value)*(*p_value) + (*integral_val)*(*i_value) + (*derivative_val)*(*d_value);
}
void controlServo(int16_t* ctrl_val){}
```

The data types have been fixed for lower memory usage and a system time has been added for the delay and will be initiated at startup.