PID Digital Controller Implementation for Wall Following Robot Technical Report

Author, *Ibrahim Ahmethan P2699379.* Supervisor, *Dr. Chigozirim Justice Uzor.*

I. INTRODUCTION

IN the previous assignment (Lab4), the Finite State Machine architecture or Binary controller was applied for wall following purpose, the output of the binary controller had two positions either to apply torque on wheels or not, and the robot was able to follow the wall in the given environment, however, due to the binary control logic, the wheels had to start and stop rotating continuously while following the wall since the wheels have two possible states either on or off, of course, such a system does not function properly for continuously fluctuating parameters, the sudden change in the binary state caused the robot having no gradual increase or decrease in the wheels speed hence, the Pioneer robot shook continuously and was not able to maintain a smooth operation of the robot while following the wall, such a drawback could affect the functioning of mechanical and electronical aspects alongside the efficiency of the design. The binary controller was successfully applied and the aim was achieved; however, there is always room for improvement. In this assignment (Lab6), a more efficient and reliable system is introduced to enhance the operational performance of the robot in order to follow the wall for the same environment used in the previous task. Closed loop system architecture with Proportional Integral Derivative (PID) controller has been implemented to achieve the wall following aim, the main idea behind the closed loop PID system is to check and correct the error which is considered as the variation between the desired output and the actual input, this process has been executed continuously during the run time until a steady state condition is satisfied without overshooting and the shortest possible settling period. The objectives of this assignment are determined as implementing the PID controller instead of the binary one to maintain the stability of the robot, testing different combinations of PID to determine pros and cons, tuning the PID controller's gain values to appropriate levels by keeping the side effects of each assigned value in mind and developing a whole software implementation system with a continuous feedback mechanism to ensure reliable and smooth operation of the robot to follow the wall in the given environment[1].

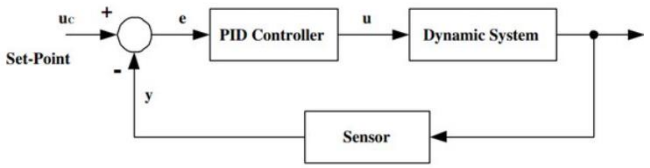


Figure 1. Diagram of PID with Feedback Mechanism

II. PID CONTROLLER IMPLEMENTATION

In this assignment, PID technology is used to regulate the Motors' speed; thus, it can be concluded that any plant inside automatic control systems can control and adapt to changes under different conditions, the following equation shows us the genius mathematical model behind the first PID digital controller developed by Elmer Sperry to automate ships steering mechanism [2].

$$u(t) = K_p\left(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{d}{dt}e(t)\right) \quad \text{eq (1)}$$

- $u(t)$ is the feedback arriving from the controller at time t
- $e(t) = y_{sp}(t) - y(t)$ is the error which is the difference between the setpoint and obtained value process at specific time t
- K_p, K_i *and* K_d respectively are P, I, and D gains

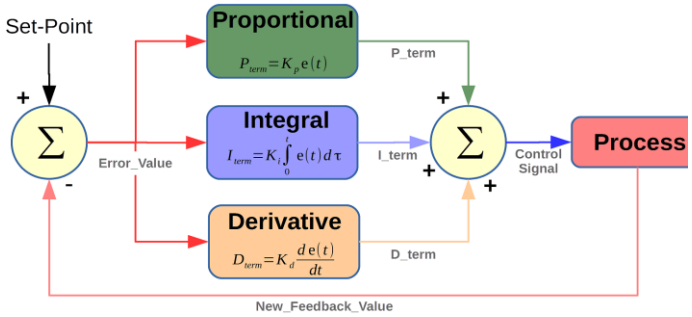


Figure 2. Closed Loop PID Controller

A. PID Design and Software Implementation

Python programming language has been utilized along with the CoppeliaSim simulation to develop and test the proposed PID method, the implementation of the PID has been built on top of the previous implementation of the binary controller, in addition, the current design has been updated with multiple methods under the Robot class. Furthermore, the conditions under the while loop modified according to the needs of the PID controller.

In the following code chunk, in addition to a general overview of the PID Python implementation and data structure, the

previous methods and attributes written to perform binary control, more 4 methods, several constants, and attributes are added to the Robot class to represent and perform the PID logic.

```
def error (self, Sp, Pv):
     ……..
        ……..
            ……..
    return value
def proportional (self, error1):
    ……..
        ……..
            ……..
    return value

def integral (self, total_error):
    ……..
        ……..
            ……..

    return value

def derivative (self,previous_error,
current_error):
    ……..
        ……..
            ……..
    return value
```

As shown above, error, **P** proportional, **I** integral, and **D** derivative parts of the controller take place separately and independently of each other as separate methods. This approach helps us to organize the code more efficiently to make it more manageable and easier in terms of detecting bugs and errors. Furthermore, it allows a user to modify the combination of **PID** such as **PI or PD**; as a result, having a flexible and manageable structure will contribute to enhancing the execution of the proposed software algorithm.

- **proportional P** component takes the difference between the **setpoint Sp** (the input value which is the desired maintained distance between the robot and wall) and the **process value Pv (**the output obtained value which is the right middle side sensor 7 readings of the Pioneer robot 3DX); thus, the typical error in the system is interpreted as the following.
  error* = K_p ∗ *(setpoint – process value)

- **integral I** component is a cumulative error, the main purpose of the integral error is to remove the static error that will be resulted from the **P** component, once the error is reduced to zero P = 0 the motor of the robot stops promptly from rotating, in this case introducing the I component will prevent such malfunction to happen and ensure the stability of the system [3]; thus, the integral component of the controller system will be interpreted as the following.
  integral = $(Ki * \sum pervioue\ errors)$

- **derivative D** component is the last one in PID, it is typically known that the derivative is the rate of change of a function with respect to time t. The purpose of using the D component is to limit the influence of overshooting "dampening" to converge to the setpoint more quickly, in another word, the derivative will reduce the settling time [4], so the derivative component can be interpreted as the following.
  Derivative* = K_d ∗ *(current error – pervious error)

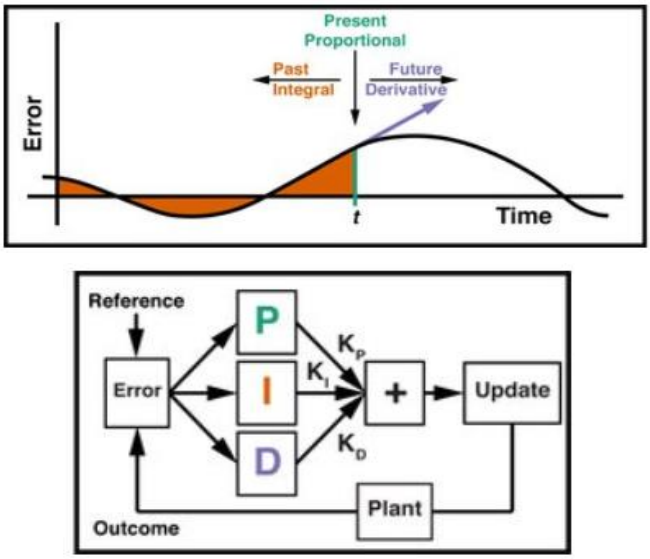


Figure 3. PID Graph and Diagram

The above *Figure 3* shows us the graph and diagram of PID; the purpose of representing the graph and diagram together is to help us visualize the theory as mentioned in eq(1) the mathematical or explicit model of PID model, we know from calculus that the integral is the area of two points under a curve and the derivative is the rate of change of a function with respect to time t (how fast our function is increasing or decreasing); therefore, by considering the properties of I and D parts in the PID system, it will give us the insight to understand the theory, visualize and implement the PID to optimize the motor speed in this assignment[5].

Another objective of this assignment, alongside the PID and its implementation, is to apply the PID to the robot to follow the wall by maintaining a predetermined setpoint, the following code chunk displays adapting the PID controller to the robot's left and right motors to substitute the obtained result of PID controller directly to the wheels.

```
elif robot.distance(s_values[7]) < Sp
and robot.distance(s_values[7]) != 0:
    …..
        ……..
    robot.turn(2 + abs((PID)), 2)
```

The previous code condition will be satisfied while the right

middle sensor [7] readings are smaller than the setpoint Sp, initially, both right and left wheels are set to a default speed of 2 m/s which is equal to 7.2 km/h, so when the reading of sensor is 7 below the setpoint, it means that the robot is converging to the wall; in this case, the absolute value of PID is added to the initial speed of the right motor. As a result of this addition, the robot starts to change its direction gradually away from the wall due to increase in the speed of the right wheel while the speed of the left wheel is constant at 2 m/s. On the other hand, while the sensor 7 reading is larger than the setpoint Sp, the robot is heading away from the wall (out of the setpoint); in this case, the opposite implementation has been carried out to increase the speed of the left wheel by adding the PID values while the speed of the right motor is constant at 2 m/s. Accordingly, the robot starts to head back to maintain the position of the setpoint. One more thing to mention is that with regarding to the absolute values used while adjusting the speed of the wheels, to change the direction of the robot there are different approaches as the following.

1- Decreasing the speed of a wheel.
2- Increasing the speed of a wheel.
3- Turning the speed of the wheels opposite to each other.

In this assignment, to steer the robot to maintain its position at the setpoint, approach number 2 is implemented, thus the absolute value of PID is added to the corresponding wheel to change the heading. It is worth noting that the reason behind taking the absolute value of PID is due to the fluctuations of the obtained values in the PID controller between negative and positive values. Therefore, the absolute value will always be guaranteed to produce positive values that will be added to the desired wheel. Lastly, the reason for picking approach 2 for steering is the smoothness of this algorithm under the condition of steering.

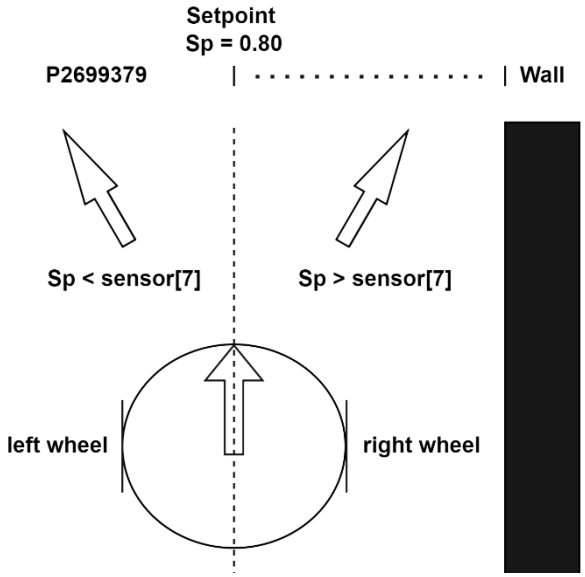
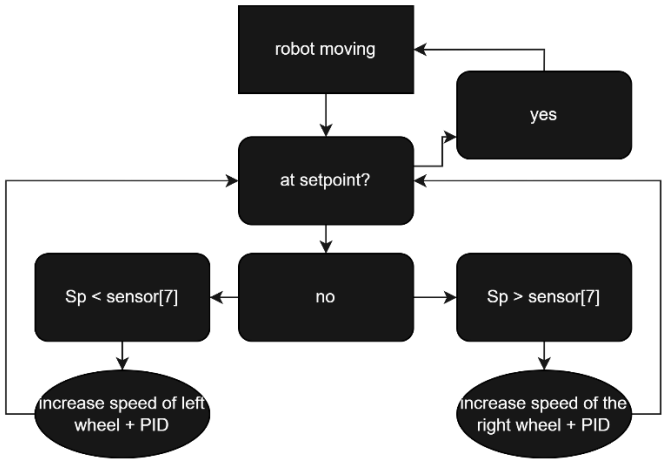


Figure 4. Top view of robot heading and Setpoint Sp



B. Design Limitations

Of course, there is no human-made system that has perfect functioning without any limitations, in this assignment (Lab 6) many technical errors and coding bugs overcame by trying and developing new solutions and approaches, It is worth noting that there are two main limitations regarding this assignment; the first one is, PID limitation, this limitation is due to the nature and principles of working in the PID system. In this case of limitation, the solution was to develop extra new functioning algorithms to optimize and minimize drawbacks related to the PID limitations, the second limitation is more likely a man-made limitation, such as being not able to implement specific task or algorithm due to lack of some experience or knowledge, it is surely true that there are no limits of learning.

As mentioned before, the PID controller has three components, proportional, Integral and Derivative respectively, the P and D components were easy to deal with since they are working with current errors, especially the D component was mostly implemented to eliminate overshooting by estimating the upcoming error and reduce the settling time; however, the integral I component is known for dealing with accumulative errors, so it is working as a memory that registers every error occurred in the past; therefore, the error summation is the total of the difference between the process value Pv and the setpoint value Sp throughout time. The only way the controller may change the error total is by altering the process value Pv since the Setpoint is predetermined. Overshoot occurs when the controller shifts the process value passing the setpoint and going beyond it to the other side.

the total error will be either dropping or growing, depending on whether the process value is larger than or less than the setpoint; as a result, there will always be some overshoot, the PID values are not utilized to eliminate the overshooting; rather, they are merely utilized to assist lessening the severity of overshooting [6]. Thus, we can conclude that a good PID controller might continue to overshoot to the opposite direction back and forth, with each subsequent oscillation getting smaller and smaller until it converges to the setpoint. The process values not only determine the degree of overshoot, but the starting position of the robot also has an impact on the overshooting, the overshoot will be larger if the process values start far from the setpoint,

for this reason, the front sensors 3 and 5 are set to detect wall and turn left when these sensors detect a wall at distance equal to the setpoint. In conclusion, once the robot detects a wall from the front sensors 3 and 5, it will turn left and the readings of sensor 7 will be close to the setpoint which is 0.60, setting the starting position close to the setpoint will play an important role in reducing overshoot; however, to enhance the proposed system additional algorithms is considered to limit the influence of the cumulative effect of the I part, 3 different approaches have been implemented considered as the following.

1- Deactivating the I component (the sum of errors) when the output of the controller is saturated
2- Deactivating the I component until the process value within a specific region of the setpoint(boundaries)
3- Readjust the sum of errors to zero while the process value exceeds the setpoint.

For this case, approach number two has been applied, it has been the best choice among the other approaches due to its simplicity. Hence allowing the I component to perform within a particular region, other than that region the integral will return the value of zero.

```
def integral(self, total_error):
    sumation = total_error
    I = i_gain * sumation
    # print(f" sumation is {sumation}")
    if I > -0.009 and I < 0.009:
        return I
    else:
        return 0
```

The code above displays the implementation of algorithm to prevent the accumulative effect of the I component [7].

C. Tuning the PID Gains

This takes up the most time and requires the greatest labor. The Kp, Ki, and Kd can be tuned using a variety of techniques, The PID constants can be adjusted manually, by mathematical calculations, or using computer programs, before starting tuning, it's critical to understand the guidelines for tuning a PID controller, for example, it must be thought what changes occur when each of the gains is increased or decreased; furthermore, how the fact the reaction of each parameter such as settling time, steady-state error, rise time, overshoot and stability of the system, for successful PID tuning all the mentioned parameters will be should be taken into consideration and examine each of them carefully after any update or change in the gain values. In this assignment, the tuning technique used is manual tuning, in addition to manual tuning, e Ziegler-Nichols method was considered; however, according to my observation and tuning experience [8], manual tuning offered a more flexible way to manipulate the gain constant. A future tuning technique using computational intelligence optimization could be a very efficient way to tune and optimize the gain values by using some genetic algorithms approach for the given solution space[9].

What happens when each of the constants is increased:

Constant:	Rise time:	Overshoot:	Settling time:	Steady-state error:	Stability:
Kp	decrease	increase	Small change	decrease	degrade
Ki	decrease	increase	increase	decrease	degrade
Kd	minor change	decrease	decrease	No effect	Improve (if small enough)

III. Result and Conclusion

Finally, the Pioneer 3DX robot is able to follow the wall by utilizing the PID controller to adapt its wheel speed according to the obtained sensor readings. Several combinations of PID, such as PI, PD and P, have been applied to examine the performance of the robot under a variety of conditions. According to the observation, the ideal performance has been carried out by PID, PD and PI respectively. Although the robot succeeds to follow the wall, there are some limitations when the robot makes a U-turn; However, the robot conducted satisfactory performance. Future work could be done to enhance the wall-following robot by trying different approaches for tuning the gains and introducing more parameters to the PID controller such as time t (which was neglected in this assignment), lastly thanks to the PID controller, the PID performance compared to the binary one, it is obvious that PID is the winner of this challenge by offering more stable and reliable robot operation for wall following.

References

[1]https://www.electrical4u.com/settling-time/#:~:text=Increase%20proportional%20gain%20KP,KD%2C%20settling%20time%20decreases.

[2] https://www.controleng.com/articles/understanding-pid-control-and-loop-tuning-fundamentals/

[3] https://arrow.tudublin.ie/cgi/viewcontent.cgi?article=1088&context=engscheleart

[4] https://www.researchgate.net/publication/318779589_Modified_PID_Implementation_On_FPGA_Using_Distributed_Arithmetic_Algorithm_Comparison_with_traditional_implementation/figures?lo=1

[5] https://www.researchgate.net/publication/345051167_A_control_theoretic_model_of_adaptive_behavior_in_dynamic_environments/figures?lo=1

[6] https://smithcsrobot.weebly.com/uploads/6/0/9/5/60954939/pid_control_document.pdf

[7] https://www.wescottdesign.com/articles/pid/pidWithoutAPhd.pdf

[8] https://www.mstarlabs.com/control/znrule.html

[9] https://pidexplained.com/how-to-tune-a-pid-controller/

APPENDIX

```
import numpy as np
import random
import sim as vp

print("Hello Dr.Uzor!")
s_values = [] ## sensor 7 readings will be appended to this list
class Robot(): # the main Robot class with several methods
    def __init__(self):
        for i in range(1, 17): ## this loop used to activate all the sensors in the
robot by looping through the sensors
            error, self.i = vp.simxGetObjectHandle(clientID,
'Pioneer_p3dx_ultrasonicSensor' + str(i),
                                                   vp.simx_opmode_blocking)
            error = vp.simxReadProximitySensor(clientID, self.i,
vp.simx_opmode_streaming)
            s_values.append(self.i)

        code, self.ibo = vp.simxGetObjectHandle(clientID, "Dummy",
vp.simx_opmode_blocking)

        # activating left and right motors
        code, self.R_motor = vp.simxGetObjectHandle(clientID,
"Pioneer_p3dx_rightMotor",
                                                    vp.simx_opmode_blocking)
        code, self.L_motor = vp.simxGetObjectHandle(clientID,
"Pioneer_p3dx_leftMotor",
                                                    vp.simx_opmode_blocking)
    def move(self): ## method created to move the robot with default speed 1.5 for
both motors
        code = vp.simxSetJointTargetVelocity(clientID, self.R_motor, 1.5,
vp.simx_opmode_blocking)
        code = vp.simxSetJointTargetVelocity(clientID, self.L_motor, 1.5,
vp.simx_opmode_blocking)

    def turn(self, turn_right, turn_left):## method to steer the robot with two
attributes left and right steering
        code = vp.simxSetJointTargetVelocity(clientID, self.R_motor, turn_right,
vp.simx_opmode_blocking)
        code = vp.simxSetJointTargetVelocity(clientID, self.L_motor, turn_left,
vp.simx_opmode_blocking)

    def position(self):  #  used to identify specific points or reference frames in
the scene
        code, position = vp.simxGetObjectPosition(clientID, self.ibo,-
1,vp.simx_opmode_blocking)
        y = np.array(position)
        return np.linalg.norm(y)

    def wandering(self): ## method to allow the robot wander if no wall detected
        x = random.randint(0, 6)
        y = random.randint(0, 6)
        code = vp.simxSetJointTargetVelocity(clientID, self.R_motor, x,
vp.simx_opmode_blocking)
        code = vp.simxSetJointTargetVelocity(clientID, self.L_motor, y,
vp.simx_opmode_blocking)

    def distance(self, sensor): ## method to measure the distance with one attribute
sensor
```

```
        returnCode, detection, detection_point, detection_object_handle,
surface_normal  = vp.simxReadProximitySensor(
            clientID, sensor, vp.simx_opmode_buffer)
        x = np.array(detection_point) ## converting the x, y and z values to array.
in order to linalg.norm function
        if detection == True:
            return np.linalg.norm(x) ## normalizing the x, y and z coordinates to
have a single return value
        else:
            return 0

    def error(self, Sp, Pv): ## error = setpoint - process value "sensor[7]
readings"
        error = Sp - Pv
        return float(error)

    def proportional(self, error): ## applying the proportional component
        P = p_gain * error
        return P

    def integral(self, total_error): ## applying the integral component
        sumation = total_error
        I = i_gain * sumation
        # print(f" sumation is {sumation}")
        if I > -0.65 and I < 0.65: ## limiting the I component to avoid constant
accumulation effect
            return I
        else:
            return 0

    def derivative(self,previous_error, current_error): ##applying the derivative
component
        error_difference = current_error - previous_error
        D = d_gain * error_difference
        return D

## PID constants (gains)
p_gain = 6
i_gain = 0.5
d_gain = 6
Sp= 0.6
vp.simxFinish(-1)
clientID = vp.simxStart('127.0.0.1', 778, True, True, 5000, 5)
if clientID != -1:
    print('Connected to API Coppelia server')
    robot = Robot()
    emp_list = [0]
    while True:

        E = robot.error(Sp, robot.distance(s_values[7])) ## error in the system from
sensor 7
        emp_list.append(E) ## appending the error the the emp_list to use it later
in the D component
        P = robot.proportional(E)
        x = sum(emp_list)
        I = robot.integral(x)
        D = robot.derivative(emp_list[-2], E) ## since D component is the current
error - previous error
                                              ## for that reason emp_list[-2] is
will give the previous error
```

```
                                                    ## and E is the current error
        # print(emp_list)
        ## different combinations of PID controller
        PI = P + I
        PD = P + D
        PID = P + I + D

        if robot.distance(s_values[4]) < 0.9 and robot.distance(s_values[4]) != 0:
## avoid wall turn to the left
            robot.turn(1.2, -1.2)

        elif robot.distance(s_values[3]) < 0.7 and robot.distance(s_values[3]) !=
0:## avoid wall turn to the left
            robot.turn(1.2, -1.2)

        elif robot.distance(s_values[7]) < Sp and robot.distance(s_values[7]) != 0:
## left steering
            robot.turn(2 + abs(round(PID,20)), 2)

        elif robot.distance(s_values[7]) > Sp and robot.distance(s_values[7]) != 0:
## right steering
            robot.turn(2, 2 + (abs(round(PID, 20))))

        elif robot.distance((s_values[8])) < 2 and robot.distance(s_values[8]) != 0:
## make a U turn
            robot.turn(-2, 2)
            robot.turn(-2, 2)
        elif robot.distance(s_values[9]) < 2 and robot.distance(s_values[9]) != 0:
            robot.turn(-2, 2)
        elif robot.distance(s_values[10]) < 2 and robot.distance(s_values[10]) != 0:
            robot.turn(-2, 2)
        elif robot.distance(s_values[11]) < 2 and robot.distance(s_values[11]) != 0:
            robot.turn(0.4, 1.5)

        else:
                robot.wandering()
else:
    print('Not able to connect to API!')
```