2021/2022

Mobile Robots Assignment #2

Student Name: Mohammed Albardawil

Student ID Number: P2661204

Table of Contents

Introduction

The main aim of this assignment is to apply the basic of PID controllers into robotics. The Proportional-Integral-Derivative (PID) controller is a basic but adaptable feedback compensator construction [1]. The PID controller is commonly used since it is simple to comprehend and extremely efficient. One advantage of the PID controller is that all engineers understand differentiation and integration theoretically, thus they can construct the control system even if they do not have a comprehensive grasp of control theory. Furthermore, despite its simplicity, the compensator is highly clever in that it records the system's history (by integration) and predicts the system's future behavior (through differentiation). We will explain the effects of each PID parameter on the dynamics of a closed-loop system and show how to utilize a PID controller to enhance system performance.

In this assignment, both implementing and tuning a PID controller are examined and discussed.

Methodology

A system may be composed of many components grouped in a variety of ways; nevertheless, we will begin by examining the components and functions of a traditional closed-loop system (Figure 1) [2].

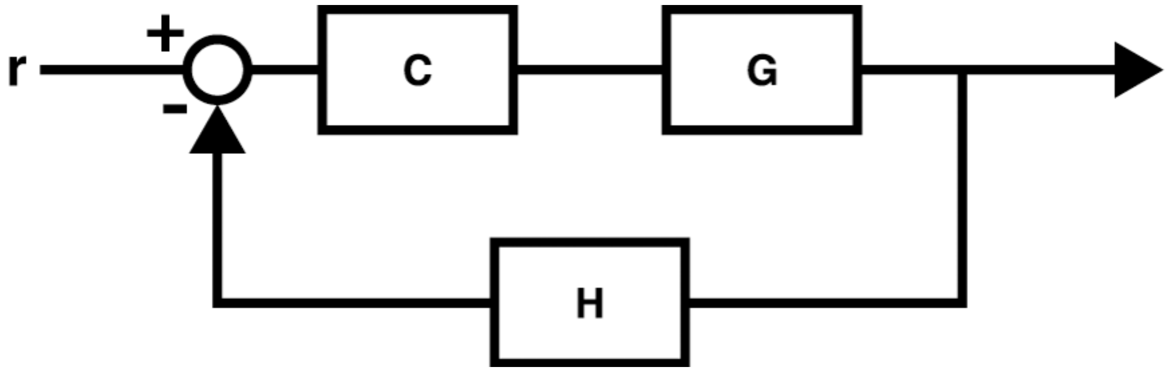


Figure 1. A single-input single-output system closed loop system [2]

1. Input: r - An input is a reference value that should be directly related to the system's output. This does not always translate to a voltage/power source, but rather to a configuration or switch that connects a voltage/power source to the system. In the system example that follows, our input signal will be a unitary step.
2. Controller: C - This is the PID controller that we will create in this situation. It is located immediately before the plant for which we are reimbursed and just after the intersection of the input signal and feedback.
3. Plant: G - This is a mathematical expression of all of your subsystems as a transfer function. If you're trying to manage a DC motor, then the plant is your DC motor. Your input will force the plant to respond in such a manner that the output value is ideally near to your input.
4. Output: y - This reading represents the real system's reaction to our intended response (the input) after it has been processed by our built system (plant). The system's performance is measured by comparing the output to the input, assuming that some mistakes will occur and have been accepted as acceptable.
5. Feedback: H - The feedback line in a system's equivalent block diagram (see Figure 1) introduces the system's output into the input. The error of the system is thus equal to: input - output x H; if the system has unitary feedback (meaning that H = 1), then our error is just input minus output. This will be a crucial concept to know as we describe how each aspect of the PID controller operates.

It is crucial to notice that the closed loop transfer function may further simplify the transfer function for the whole loop in Figure 1 into just one block with a single input and single output:

$$Closed - Loop(s) = \frac{C(s) \cdot G(s)}{1 + C(s) \cdot G(s) \cdot H(s)}$$

Hence, a PID controller is a three-part system:

1. Proportional compensation: The proportional compensator's primary role is to add a gain that is proportional to the erroneous reading created by comparing the system's output and input. Its function reduces system’s stability, however, it is the most important part as it enhances the accuracy of the system and lowers steady-state error.
2. In a unitary feedback system, the derivative compensator will inject the derivative of the error signal multiplied by a gain KD. In other words, the slope of the waveform of the erroneous signal will be injected into the output. Its primary goal is to improve the overall closed-loop system's transient responsiveness.
3. The integral compensator in a unitary feedback system introduces the integral of the error signal multiplied by a gain KI. This implies that the area under the curve of the error signal will have an effect on the output signal. We will demonstrate this later, but it is crucial to note that this aspect of the controller will enhance the entire closed-loop system's steady-state error. In addition, it reduces the fluctuations caused by the differentiator part of the controller itself. Below is a figure that demonstrates the PID controller

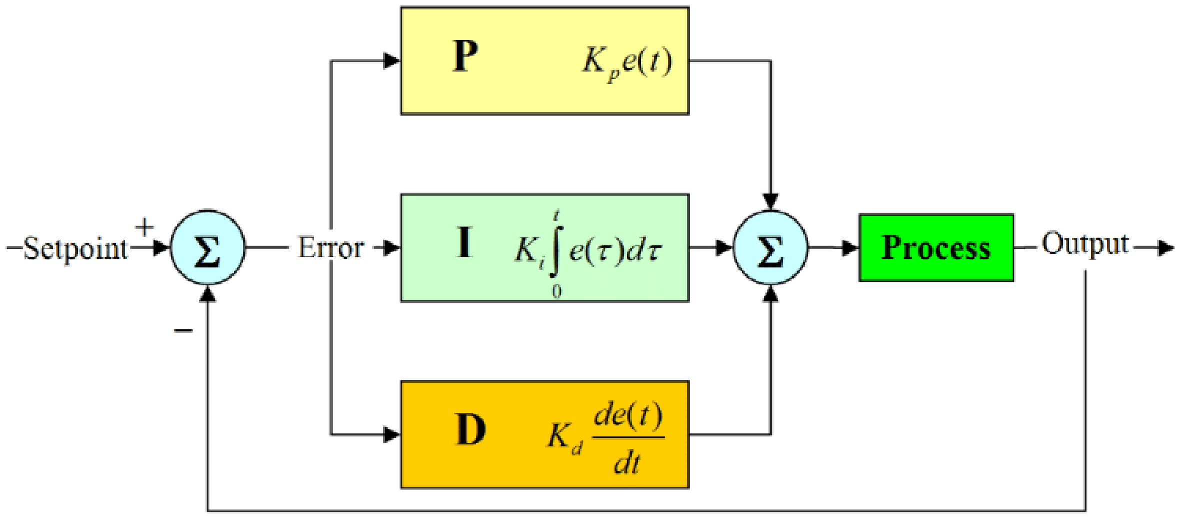


Figure 2. A PID controller Architecture (adapted from [2])

Let us explain more about the PID Control system’s structure. There are three parts as usual, input, process and output. The input is the error produced by the machine (in our case, the distance between the robot and the wall to follow and the desired distance which is the threshold that the robot must not travel beyond). The error’s symbol is denoted by e(t). Next, Proportional part is calculated by multiplying the error and the proportional gain denoted by Kp. The gain for all parameters (P, I and D) must assumed and tuned (at a later stage) by us (the system’s designer) so that it starts to control the motion of the robot. However, the issue with the P controller itself is that it overshoots if the gain determined value is high and vice versa. There should be some tuning to be applied by decreasing and increasing by small numbers the value of the Kp until reaches a satisfactory result. Although the P parameter can control the system by itself, the resulted response may become too low. In order to improve the response, here it comes the role of the derivative part of the controller. As it applies to all of the three parameters, the gain Kd is multiplied by the derivative of all errors (current – previous over an interval of (t)) as follows:

$$D = K_d \frac{d_e(t)}{d(t)}$$

Unfortunately, the D part would produce some fluctuations which affects the smoothness of the controller's response. For example, in our case the motors would not move approximately in a straight-line by the controller, it is going to move left and right as this movement is totally noticeable. To resolve this issue, the Integral part must be also involved. It also has a gain to be determined which is denoted by Ki, and it is multiplied by the integration of all errors (current + previous over an interval of (t)) during the system's process as follows:

$$I = Ki \int_0^t e(\tau)\, d\tau$$

To give the final equation as follows:

$$\boldsymbol{PID} = Kp * e(t) + Ki * \int_0^t e(\tau)\, d\tau + K_d * \frac{d_e(t)}{d(t)}$$

System's Design and Implementation

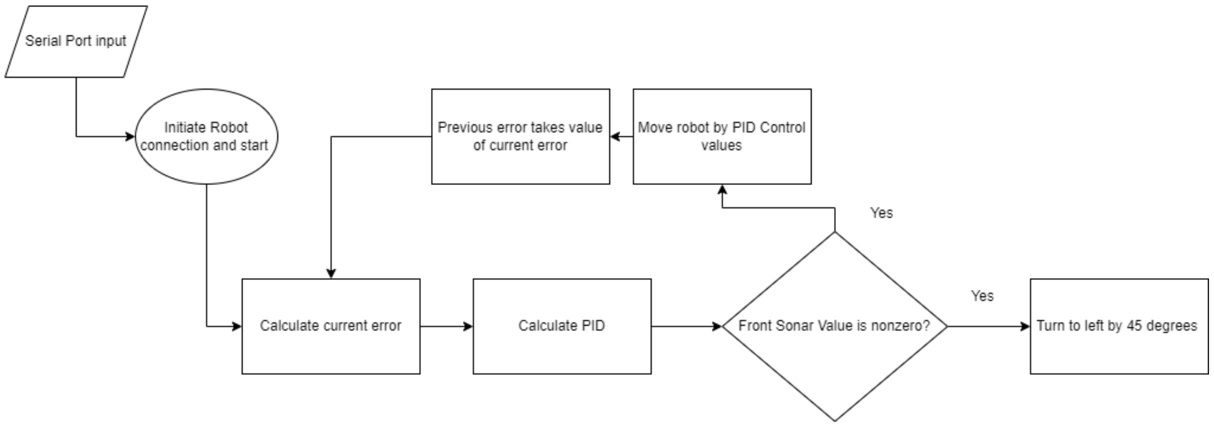
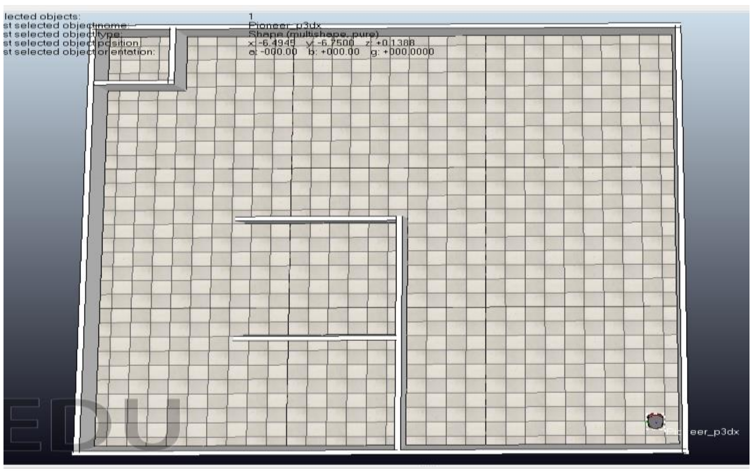

Serial Port input
Initiate Robot connection and start
Previous error takes value of current error
Move robot by PID Control values
Yes
Calculate current error
Calculate PID
Front Sonar Value is nonzero?
Yes
Turn to left by 45 degrees

Figure 3: system workflow

Before explaining the whole workflow, let us take a look at the technology used to implement the system. Unlike the last assignment which was implemented using Matlab as the programming language and the IRobot Create Toolbox for Simulation, this time I chose the Python programming language, Coppeliasim/VREP as the simulator environment and the Prioneer P3Dx robot. The robot has 16 sonar sensors, and the only sensors used in these experiments are the right and front sensors. The reason why I chose the right sonar instead of the left one is that the Robot itself must start beside the right wall of the map to follow the wall for a full cycle as described in the below image:



```
lected objects:                   1
st selected object name:          Pioneer_p3dx
st selected object type:          Shape (multishape, pure)
st selected object position:      x: -6.4945    y: -6.7500    z: +0.1388
st selected object orientation:   a: -000.00    b: +000.00    g: +000.0000
```

Figure 4: Coppeliasim and the map for robot's movement environment

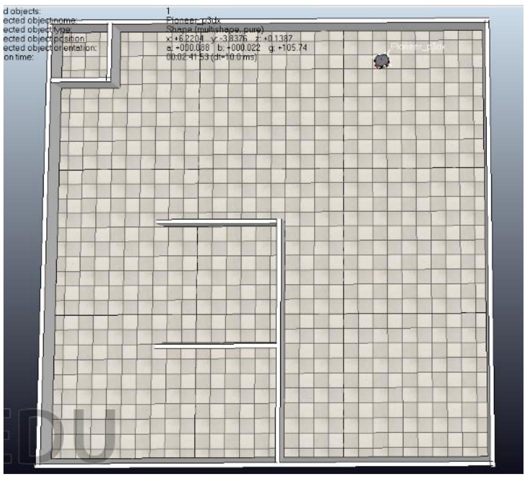
Now let us demonstrate the whole workflow that is implemented by the system. The robot's sensors used are the front and right sonar sensors. After the initial start of robot using the function sim.simxStart(), as it connects to the client part of the simulator in Coppeliasim simulator software, it starts to read the Sonar sensor value from the right side of the robot. Next, the current error is calculated by Current Error = RightSonarValue – Desired Value (which is determined as 0.5). the desired value is the threshold value in which the maximum difference between robot and the wall to return back to the right. The best thing about the PID controller compared to the Finite System Machine (FSM), is that it does not have several if and nested if statements in the code, which results in high programming performance. Once the current error is calculated, the PID controller method declared inside the code receives it as argument along with the previous error (which is initialized with 0) and the PID parameters gains as follows: robort.PID(current, previous, [0.2, 0.006, 0.005]). To make it simple, the integral part is calculate as follows: I = (current_error+previous_error)*dt, where dt is the time interval value. The same applies to the derivative part, but subtraction:

D = (current_error-previous_error)/dt

After calculating the PID, the right motor will receive a value of 1 + PID and the left motor 1 – PID, where 1 it the threshold for Motors moving velocity. As mentioned earlier, there is only two conditions, if the front sonar receives a signal, then the robot is going to turn to the left by using the method robot.turnArc(0, 2.5), supposing the value of 2.5 is 45 degrees. Otherwise, the PID effect starts to take it place by using the same method (robot.turnArc) for each motor. In the end, the previous error is assigned with the current error and then the motion of the starts to move every time with the same workflow by updating the current and previous errors values, without the need of any conditional statement (automatic linear control). Note that the time step chosen (dt) is 10ms, that is why the value chosen for dt=0.01.

Analysis and Results

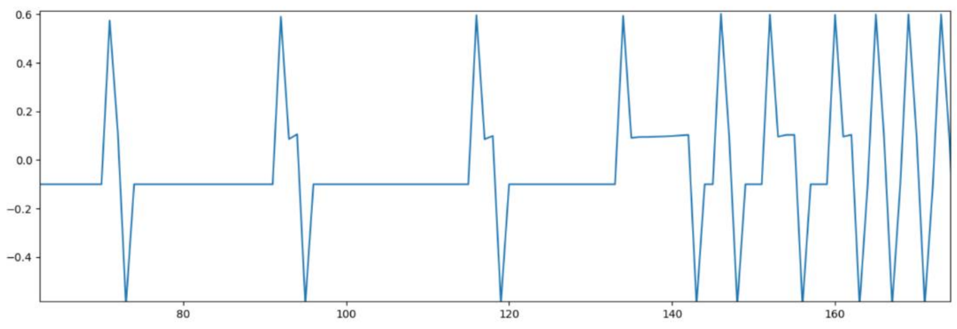
As mentioned earlier that there are two parts of this project. The first is to build a PID controler for a wall following robot, and the second is to tune the controller by changing the parameters of the PID controller. Changing parameters means that each time the gain for each part has to be changed. Starting at the value of Kp, the first value was 1.5. However, it has a high overshoot and a noticeable difference between the velocity of values of each motor which cannot be reduced or controlled unless the value has reduced to a smaller number. Then I made several experiments until reached the value of 0.2. Unfortunately, this made a very bad movement as it reached. See the image below to understand the curve made by the robot's movement with the use of the P controller only. Then, I added the value of the D, and also tried to tune it along the value of the Integral part several times until it reached the smooth movement.



```
d objects:                    1
ected object name:            Pioneer_p3dx
ected object type:            Shape (multishape, pure)
ected object position:        x: +6.2204   y: -3.8376   z: +0.1387
ected object orientation:     a: +000.088   b: +000.022   g: +105.74
on time:                      00:02:41.53 (dt=10.0 ms)
```

Figure [5]: Movement of Robot based on PID controller

Below is a graph that shows the PID behaviour for a half cycle.



As the above paragraph explains, the response of the PID is almost stable excepts that the front sonar would have been tuned better such that the right turn must have a smaller value.

Conclusion and Learning Outcomes

In the end, the whole system performs the wall following smoothly PID controller. This achieved some analytical results by differentiating distance and velocity for 2 different robotics positions inside the map. To conclude, the robot moved a full cycle with the speed of 1 m/s again using differential system, with help of the distance read by the Right Sonar to ensure the robot moving approximately on a virtual straight line following all walls inside the provided map in this assignment. I have learned a lot in this assignment. I have understood how the PID controller works and how it deals with wall following as an robot motion application. I also have learned to find and make use of other useful functions in Python and Coppliasim simulator, as it was a bit hard to understand to the lack of resources. It is worth to mention that I have been introduced to a new concept in Control Systems engineering, which made it possible to solve and apply applications with less code achieving high performance and accuracy.

Appendix

```
try:
    import sim
except:
    print ('--------------------------------------------------------------')
    print ('"sim.py" could not be imported. This means very probably that')
    print ('either "sim.py" or the remoteApi library could not be found.')
    print ('Make sure both are in the same folder as this file,')
    print ('or appropriately adjust the file "sim.py"')
    print ('--------------------------------------------------------------')
    print ('')

import time, math, random, matplotlib.pyplot as plt


class Robot():

    def __init__(self):

        # Setup Motors
        res, self.leftMotor = sim.simxGetObjectHandle(clientID, 
'Pioneer_p3dx_leftMotor',sim.simx_opmode_blocking)
        res, self.rightMotor = sim.simxGetObjectHandle(clientID, 
'Pioneer_p3dx_rightMotor',sim.simx_opmode_blocking)

        # Setup Sonars
        res, self.frontSonar = sim.simxGetObjectHandle(clientID, 
'Pioneer_p3dx_ultrasonicSensor5',sim.simx_opmode_blocking)
        res, self.leftSonar = sim.simxGetObjectHandle(clientID, 
'Pioneer_p3dx_ultrasonicSensor1',sim.simx_opmode_blocking)
        res, self.rightSonar = sim.simxGetObjectHandle(clientID, 
'Pioneer_p3dx_ultrasonicSensor8',sim.simx_opmode_blocking)
        res, self.rightFrontSonar = sim.simxGetObjectHandle(clientID, 
'Pioneer_p3dx_ultrasonicSensor7',sim.simx_opmode_blocking)

        # Start Sonars
        res,detectionStateF,detectedPoint,detectedObjectHandle,detectedSurface
NormalVector = 
sim.simxReadProximitySensor(clientID,self.frontSonar,sim.simx_opmode_streaming
)
        res,detectionState,detectedPoint,detectedObjectHandle,detectedSurfaceN
ormalVector = 
sim.simxReadProximitySensor(clientID,self.leftSonar,sim.simx_opmode_streaming)
        res,detectionState,detectedPoint,detectedObjectHandle,detectedSurfaceN
ormalVector = 
sim.simxReadProximitySensor(clientID,self.rightSonar,sim.simx_opmode_streaming
)
```

```
        res,detectionState,detectedPoint,detectedObjectHandle,detectedSurfaceN
ormalVector =
sim.simxReadProximitySensor(clientID,self.rightFrontSonar,sim.simx_opmode_stre
aming)
        time.sleep(2)


    def getDistanceReading(self, objectHandle):
        # Get reading from sensor
        res,detectionState,detectedPoint,detectedObjectHandle,detectedSurfaceN
ormalVector =
sim.simxReadProximitySensor(clientID,objectHandle,sim.simx_opmode_buffer)

        if detectionState == 1:
            # return magnitude of detectedPoint
            return math.sqrt(sum(i**2 for i in detectedPoint))
        else:
            # return another value that we know cannot be true and handle it
(use a large number so that if you do 'distance < reading' it will work)
            return 9999

    def sonar(self, son):
        if son == 1:
            return self.getDistanceReading(self.frontSonar)
        elif son == 2:
            return self.getDistanceReading(self.rightSonar)


    def stop(self):
        res = sim.simxSetJointTargetVelocity(clientID, self.leftMotor, 0,
sim.simx_opmode_blocking)
        res = sim.simxSetJointTargetVelocity(clientID, self.rightMotor, 0,
sim.simx_opmode_blocking)

    def turnArc(self, leftMotorVelocity, rightMotorVelocity):
        res = sim.simxSetJointTargetVelocity(clientID, self.leftMotor,
leftMotorVelocity, sim.simx_opmode_blocking)
        res = sim.simxSetJointTargetVelocity(clientID, self.rightMotor,
rightMotorVelocity, sim.simx_opmode_blocking)

    #Calculate PID
    def PID(self, current_error, previous_error, arr):
        dt = 0.01
        sum_error = 0
        sum_error = (previous_error + current_error)*dt
        current_error_derivative = (current_error - previous_error)/dt
        PID = arr["Kp"] * current_error + arr["Ki"] * sum_error + arr["Kd"] *
current_error_derivative
        right_PID = 1 + PID
        left_PID = 1 - PID
        return (PID, right_PID, left_PID)


print ('Program started')
sim.simxFinish(-1) # just in case, close all opened connections
clientID=sim.simxStart('127.0.0.1',19999,True,True,5000,5) # Connect to V-REP

curr_err_arr = []
PID_arr = []
if clientID!=-1:
    print ('Connected to remote API server')
    robot = Robot()
    previous_error = 0

    while 1:
        right_sonar_val = robot.sonar(2)
        if right_sonar_val == 9999:
            right_sonar_val = 0
        current_error = right_sonar_val - 0.5   #0.6 is the reference distance
value (threshold)

        PID, right_PID, left_PID = robot.PID(current_error, previous_error,
{"Kp":0.2, "Ki":0.006, "Kd":0.005})

        print("Current Error: ", current_error)

        if robot.sonar(1) != 9999 or
robot.getDistanceReading(robot.rightFrontSonar) != 9999:      # threshold
value
            robot.turnArc(0, 2.5)
            print(right_PID/2)
        else:                                                     #near
right wall
            robot.turnArc(left_PID, right_PID)

        previous_error = current_error

    # Before closing the connection to V-REP, make sure that the last command
sent out had time to arrive. You can guarantee this with (for example):
    sim.simxGetPingTime(clientID)
    # Now close the connection to V-REP:
    sim.simxFinish(clientID)
else:
    print ('Failed connecting to remote API server')
print ('Program ended')
```

References :

[1] ‘Control Tutorials for MATLAB and Simulink - Introduction: PID Controller Design’. [Online]. Available: https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID. [Accessed: 25-Nov-2021]

[2] https://www.researchgate.net/figure/The-general-structure-of-PID-controller_fig4_335716292