Implementing PID Controller on a hand follower robot using Numerical Methods

Abdallah Amr Mohamed-120200011, Ahmed Hatem Aborehab-120200012, Mohamed Alfateh Shabaan-120200160, Mohamed Khaled Mohamed-120210346, Mostafa Ahmed Mostafa-120200214, Mostafa Osama Mohamed-120200018, Tarek Mahmoud Shohdy-120200030

Submitted To Prof. Waheed Zahra

Abstract—Overall, in any system, the proportional term, integral term, and derivative term combined to produce a fast response time, less overshoot, no oscillations, increased stability, and no steady-state errors. Eliminating the steady state errors connected to typical PID systems is crucial for achieving stability. To plot the transfer function's responses with various integrator gains for auto tuning, a MATLAB M-file was developed. Auto tuning techniques were then applied to PID systems to eliminate steady state defects. this paper analyzes and tests the improvement of PID controller over the regular P-controller taking a hand follower robot as a system example using methods with simulation and numerical analysis study.

Index Terms—control methods, PID controller, Numerical analysis, Stability, Steady-state error, simulation

I. INTRODUCTION

Proportional-Integral-Derivative (PID) control is the most widely used control algorithm in industry and has become the mainstream of industrial control. The popularity of PID controllers is due in part to their robust performance over a wide range of operating conditions and in part to their functional simplicity, which allows engineers to operate them in a simple and straightforward manner. the settings for this controller should be known to tune it correctly to produce the preferred output. Here, tuning is the process of obtaining the ideal response of the controller by setting the optimum proportional gain, integral and derivative factors. One type of action used in PID controllers is proportional control. Proportional control is a form of feedback control. It is the simplest form of continuous control that can be used in closed loop control systems. Pure P control minimizes process variable fluctuations but does not always bring the system to the desired set point. It provides a faster response than most other controllers, which makes the P-only controller initially responsive a few seconds faster. However, as the system becomes more complex (i.e. more complex algorithms), the response time differences may accumulate such that the P-controller may respond even minutes faster. Although a pure P controller has the advantage of a faster response time, it introduces deviations from the setpoint. This deviation is called offset and is usually not needed in the process. The presence of an offset means that the system cannot maintain the desired set point in steady state. It is analogous to bias in a calibration curve, where there is always a fixed, constant error that prevents the line from passing through the origin. The offset can be minimized by combining pure P control with another form of control, such as B. I or D control, is combined. However, it is important to note that it is not possible to eliminate the offset. [1]
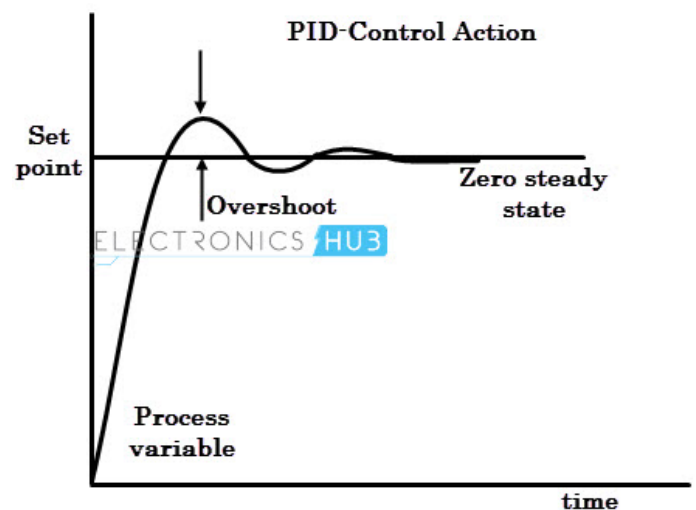

Fig. 1. PID control Action

This type of PID controller combines proportional control with integral and derivative control to automatically help equipment compensates for changes within the system. These modifications, integrals, and derivatives are expressed in time-based units. The integral function eliminates offset by summing up the input signal over time keeping running total so it has the memory of what happened before or the past information on other meaning when the system being in steady state under the desired point it creates a nonzero error so the integration will increase the output value, after a few iterations of increasing the output overshooting over the desired value at this point(the point above the desired value) utility of the derivative mode comes to provide fast response and prevent overshooting by calculating the rate of change of how the error growing or shrinking to provide the future information to the system and take a decision for next move depends on that. However, PID controllers are difficult to tune. but when properly tuned, provide the best control system. [4]
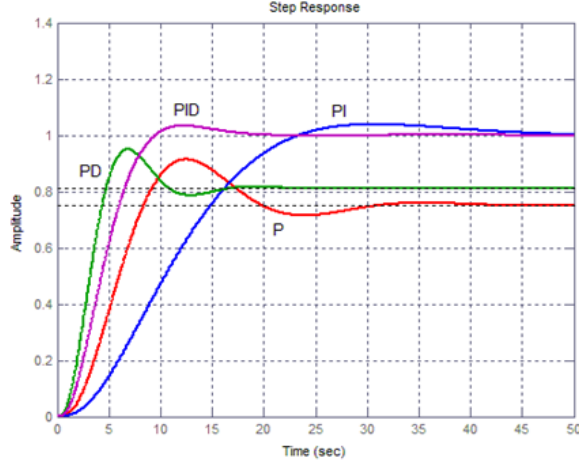
Fig. 2. PID controller output signal

II. OBJECTIVES

- Apply the numerical methods of differentiation in the part of the "D" (differential) in the PID controller which anticipates the future errors.
- Reflect our studies in the integration using numerical methods in the "I" (Integral) part of the PID controller that calculates the past errors.
- Keep the robot at the same distance from the target with the highest accuracy possible using the numerical methods studied in different lessons.

III. ANALYSIS

In this section, we are going to explain the math behind the PID controller and how we are going to numerically approximate the integral and differential values. Also, the MATLAB implementation for the controller on a differential drive robot to keep him on a certain distance from a wall, object, hand, etc...

A. Theory

First, we have to see the mathematical formula for PID controller. For an Error function given by:

$$E(t) = R - x(t) \quad (1)$$

where $E(t)$ is the Error function of the system, $x(t)$ is the distance the robot moved at time t, R is the reference point (wall, hand, ...). Reference R can be constant like the case we are studying or can be also a function of t Example: $sin(t)$, $cos(t)$, e^t.

The input for the PID controller is the Error function value at time t, and then we use the integral of the function to calculate the previous error and compensate that error in the form of overshooting from the desired point, to decrease this overshoot, we have to use the derivative of the Error function to make the output speed starts to decrease as we

are approaching the desired point. At the end, can get the linear speed of the robot as:

$$v = K_p E(t) + K_i \int_0^t E(t)\, dt + K_d \frac{dE(t)}{dt} \quad (2)$$

where K_p, K_i, K_d are parameters to be tuned to get the best response from the robot.

1- Evaluating definite Integral using Simpson's rule:[3] Using Simpson's rule to evaluate the definite integral of the function. As $E(t)$ values are recorded each h sec, so we can apply the rule to evaluate the integral $I = \int_0^t E(t)$:

$$I = \frac{h}{3}[E_1 + 2\sum_{i=3}^{m-1} E_{2i-1} + 2\sum_{i=2}^{m} E_{2i} + E_{2m}] \quad (3)$$

.

Where h is the time step size, $i = 1, 2, 3, ..., n$, $n = 2m$ and is equal to the number of points we have. The evaluated integral have a truncation error $O(h^5)$ which is accurate enough for our application.

For even number of points, we cannot apply Simpson's rule, so we have to evaluate Simpson integral for $n - 1$ points and the last 2 points to be evaluated using trapezoidal area rule, then we can evaluate I as:

$$I = I_{simpson} + \frac{E_n + E_{n-1}}{2}h \quad (4)$$

2- Evaluating Derivative using backward difference: [2] Primary task for our problem is to get the derivative of the last Error value, we have to use the backward difference method between the last to points:

$$E'(t) = \frac{E_n - E_{n-1}}{t_n - t_{n-1}} \quad (5)$$

but, also we have to consider the accuracy and truncation error of the method used. For the backward difference the truncation error is $O(h)$ which is significant for our problem. However, we can handle this problem by evaluating a form for derivative with higher accuracy using the last 3 points. The Taylor expansion for the Error function of $E(t_{n-1}) = E(t-h)$ and $E(t_{n-2}) = E(t - 2h)$ is:

$$E(t - h) = E(t) - hE'(t) + \frac{h^2}{2}E''(t) + \frac{h^3}{6}E'''(t) \quad (6)$$

$$E(t - 2h) = E(t) - 2hE'(t) + 2h^2 E''(t) + \frac{8h^3}{6}E'''(t) \quad (7)$$

By summation of equations 6,7, we can get rid of the h^2 term and decrease the truncation error.

$$-4E(t - h) = -4E(t) + 4hE'(t) - 2h^2 E''(t) - \frac{4h^3}{6}E'''(t)$$

$$E(t - 2h) = E(t) - 2hE'(t) + 2h^2 E''(t) + \frac{8h^3}{6}E'''(t)$$
$$(8)$$

The 2 equations in 8 add up to formulate:

$$E(t-2h) - 4E(t-h) = -3E(t) + 2hE'(t) + \frac{4h^3}{6}E'''(t) \quad (9)$$

Finally, we can approximate $E'(t)$ as:

$$E'(t) = \frac{E(t-2h) - 4E(t-h) + 3E(t)}{2h} - \frac{h^2}{3}E'''(t)$$
$$E'(t) = \frac{E(t-2h) - 4E(t-h) + 3E(t)}{2h} + O(h^2) \qquad (10)$$

In terms of the error values, we can say that for n points, the backward difference equation is:

$$E'_n = \frac{E_{n-2} - 4E_{n-1} + 3E_n}{2h} \qquad (11)$$

B. MATLAB Implementation

After we understood the mathematical concept for the PID controller and how to interpret the values of the integral and the derivative, now we can implement all the work as MATLAB codes to simulate and test the controller.

First, we must consider the kinematic model for the differential drive robot we are studying. The kinematic model for the robot is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2}cos(\theta) & \frac{r}{2}cos(\theta) \\ \frac{r}{2}sin(\theta) & \frac{r}{2}sin(\theta) \\ \frac{r}{2b} & \frac{-r}{2b} \end{bmatrix} \begin{bmatrix} \dot{\phi_r} \\ \dot{\phi_l} \end{bmatrix} \qquad (12)$$

where $\dot{x}, \dot{y}, \dot{\theta}$ are the velocities of the robot, θ is the orientation of the robot from the x-axis, ϕ_r and ϕ_l are the angular velocities of the wheels, r is the radius of the wheels and b is the track width (half the distance between the 2 wheels). All as shown in fig. 3. As we just need the x position - the
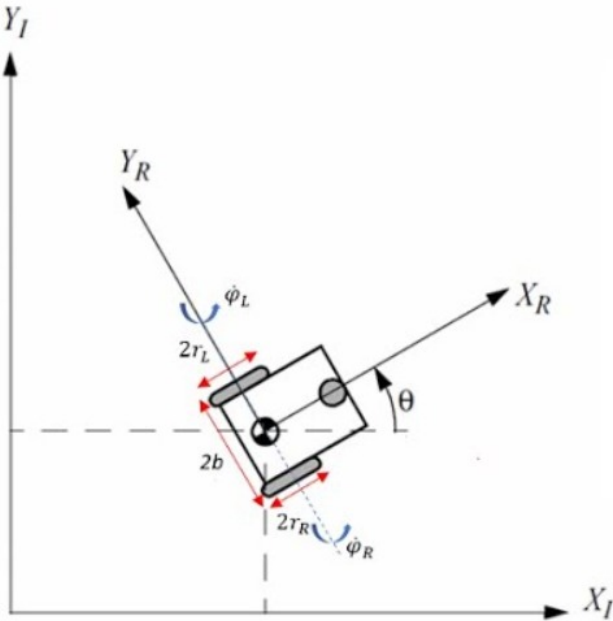

Fig. 3. Differential drive robot

robot will move just forward and backward so the y position and θ is - so we can summarize equation 12 into:

$$x_{n+1} = x_n + R\dot{\phi}t \qquad (13)$$

At the end, we add all the x values of the robot into array and return the array x

$$x = [x_1, x_2, x_3, ..., x_n] \qquad (14)$$

Following code 1 is the MATLAB code for the kinematic model

```
function x = Robot_Kinematics(w,x,t)
% x is an array for all the distances the robot
    travelled
% w is the angular velocity of the wheels in rad/sec
% t is the time step size
% r = 0.05
    x = [x  x(end) + w*0.05*t];
```

Listing 1. Differential Drive kinematic model MATLAB code

Next, we are going to implement the Simpson's rule in MATLAB. We check first is the number of points odd or even, to classify whether to use Simpson rule equation 3 or Simpson and trapezoidal rule equation 4. Applying the rule, we can use a for loop to sum all the intervals and then multiply by $\frac{h}{3}$. Code represents the MATLAB code for Simpsons's rule.

```
function integral_x = Simpson_integral(x,t)
% x is the array of Error values
% t is the array of time
    if length(t) == 1
        % if the time array is still have only one
    value return 0
        integral_x = 0;
        return
    end
    h = t(2)-t(1); % h value
    % for odd number of points, even number of
    intervals we apply simpson
    if mod(length(t),2) == 1
        integral_x = x(1) + x(length(x)); % add
    first and last value
        for i = 2:length(x)-1
            if mod(i,2) == 1
                % for odd points, we multiply by 2
                integral_x = integral_x + 2*x(i);
            else
                % for even points, we multiply by 4
                integral_x = integral_x + 4*x(i);
            end
        end
        integral_x = integral_x * h/3;
        % multiply by h/3 and return the value
    else
        % for even points, odd number of points we
    cannot apply only simpson, but take the last 2
    points and apply trapezoidal + Simpson
        integral_x = x(1) + x(length(x)-1);
        for i = 2:length(x)-2
            if mod(i,2) == 1
                integral_x = integral_x + 2*x(i);
            else
                integral_x = integral_x + 4*x(i);
            end
        end
        integral_x = integral_x * h/3 + h/2 * (x(
    length(x)-1) + x(length(x)));
        % trapeziodal rule
    end
```

Listing 2. Simpson Integral MATLAB code

Then, the implementation of the derivative using the 3 points backward difference method. Applying equation 11 on the MATLAB simply by calling the last 3 points from the Error array and dividing by $2h$. Code 3 is the implementation of the method for derivative.

```matlab
function dx = Derivative_Backward_difference(x,t)
% x is the array of Error values
% t is the array of time
    if length(x) == 1
        % if there is only 1 point return 0
        dx = 0;
        return
    end
    if length(x) == 2
        % for only 2 points apply normal 2 points
    slope
        dx = (x(2)-x(1))/(t(2)-t(1));
        return
    end
    % return dx vale for by using 3 point backward
    difference
    dx = (x(end-2) -4*x(end-1) + 3*x(end))/(2*(t(end
    ) - t(end-1)));
```

Listing 3. Derivative backward difference MATLAB code

PID controller depend on the last 2 codes and how accurately they evaluate the integral and derivative, apparently all we need is to apply equation 2 and give the main function the parameters K_p, K_i, K_d. Code 4 is the MATLAB function takes input the parameters and error array and time array to pass them to integral and derivative functions.

```matlab
function output = PID_Numerical(E,t,Kp,Ki,Kd)
% Kp, Ki, Kd are tuning parameters for the
    controller
% E is the array of Error values
% t is the array of time
    output = Kp*E(end) + Ki*Simpson_integral(E,t) +
    Kd*Derivative_Backward_difference(E,t);
```

Listing 4. PID controller MATLAB code

Finally, we can put all into work to get the whole system code 5. We have to define the time array t and the h for the system, then the initial position and the reference we are trying to reach. Subsequently, we define the tuning parameters K_p, K_i, K_d.

Last, the for loop that execute the system, as first we build the error array by subtracting the reference from the position values equation 1, pass the error array and the tuning parameters to the PID controller and get the output velocity and pass it to the kinematic model to get the new position array x and loop for the time span we have defined. At the end we plot the x vs t and fine tune the controller using induction.

```matlab
% defining time array and define h
t = 0:0.01:30;
% define the reference and x initial = 0
reference = 1;
x = 0;
% define the tuning parameters
Kp = 10.8;
Ki = 17.7;
Kd = 3.2;
```

```matlab
% the loop that excute and simulate the control
    system
for i = 2:length(t)
    % E is the error array obtained by subtracting
    all the x values from the reference
    E = reference - x;
    % w is the velocity of the robot obtained by the
     PID controller
    w = PID_Numerical(E,t(1:i-1),Kp,Ki,Kd);
    % the new value of x from the kinematic model
    x = Robot_Kinematics(w,x,0.02);
end
% plotting the distance the robot moved fine tune
    the parameters by induction
plot(t,x)
title("PID controller on Follower Robot using
    numerical methods")
hold on
plot(t,reference*t./t,'LineStyle','--')
legend('x(t)','Reference')
hold off
```

Listing 5. system simulation MATLAB code

IV. RESULTS

To validate our work we had to run multiple tests, observe their results and try to give an explanation to what we have observed. This section gives the test results of all the functions and parameters we tested.

First of all we tested the derivative and integration functions we implemented on MATLAB to make sure they are giving true answers when we input common functions like trigonometric functions, and the two functions proven success and accuracy as shown in fig. 4 where we tried to get the derivative and integral of sin(x).
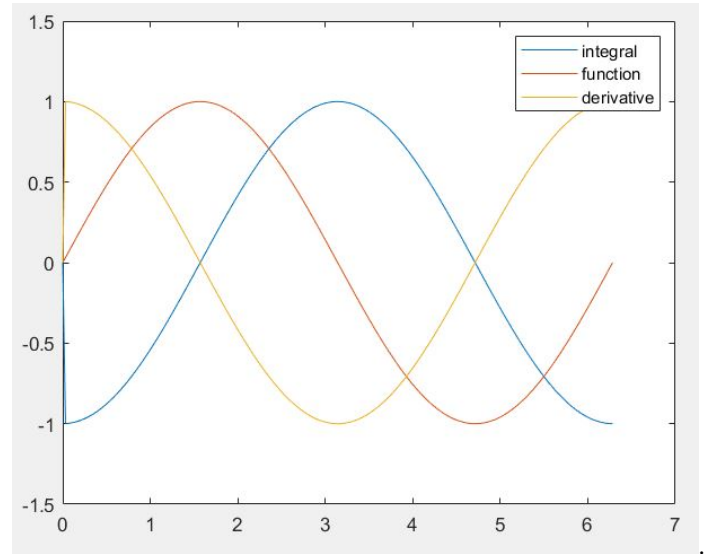

Fig. 4. Derivative and Integral test for sin(x)

Then we started changing the KP, KI and KD parameters to see how they affect the output signal. Starting with the KP when we kept the KI at 17.7 and KD at 3.2 we started with a KP of 5 which made great oscillation and increased the settling time as shown in fig. 5. While increasing the KP we noticed that the

controller is more aggressive, the oscillation and the settling time is smaller. We settled on a KP value of 10.8 at last with a conclusion that KP accounts for instantaneous deviation of the set point.
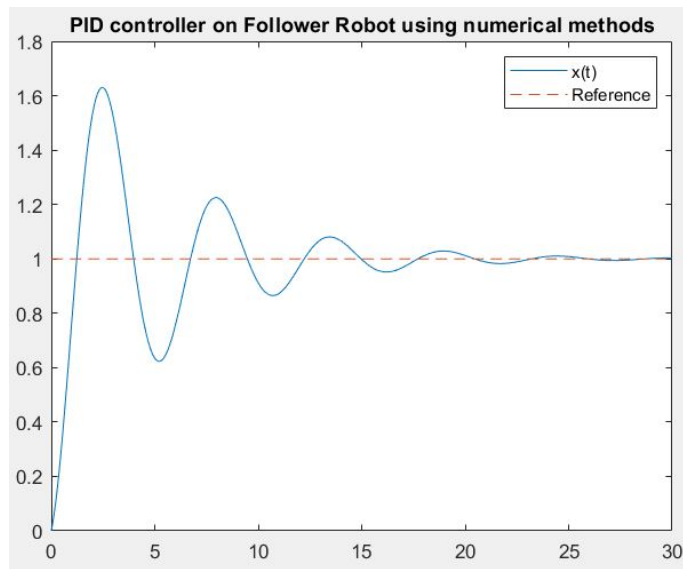

Fig. 5. Controller output signal at KP = 5, KI = 17.7 and KD = 3.2

After that we left the KP at 10.8 and KD at 3.2 and started to change the KI and observe the output signal. We noticed that increasing the KI eliminates the steady-state error but increases overshoot and oscillations as in fig. 6 where we set the KI = 30. At last we found that a KI value of 17.7 is descent and gives a satisfying output signal.
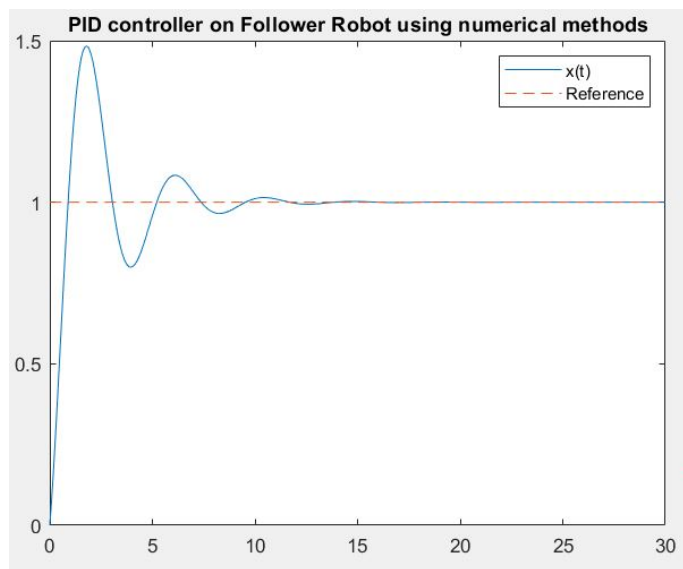
Fig. 6. Controller output signal at KP = 10.8, KI = 30 and KD = 3.2

Finally we set the KP at 10.8 and KI at 17.7 and started changing the KD value while observing the output signal. when we increased the KD value over 4.9 we noticed that a

very high oscillation occurs and the signal does not reach the set point over time. Increasing the KD makes the oscillation more smooth, however high KD value -above 4.9 in our example- can make the system unstable as shown in fig. 7.
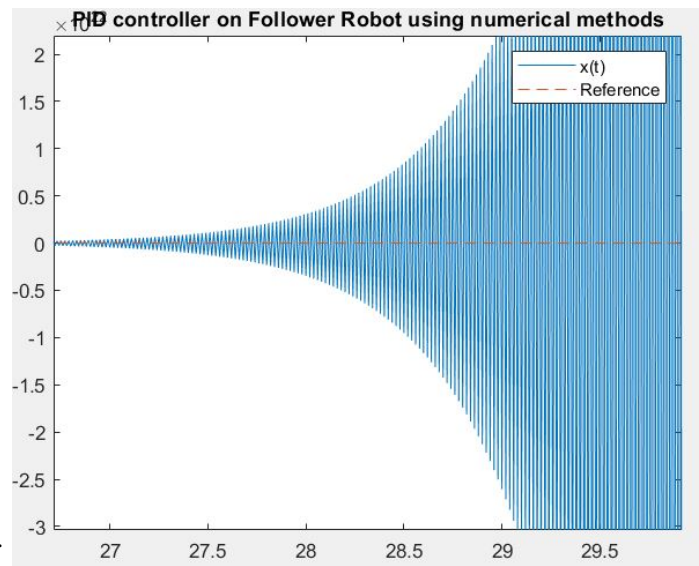

Fig. 7. Unstable output signal at KP = 10.8, KI = 17.7 and KD = 5.1

At last we settled on a KD value of 3.2 to get the controller output signal shown in fig. 8 by that we settle on the set point in a good time and the oscillation is not considered high.
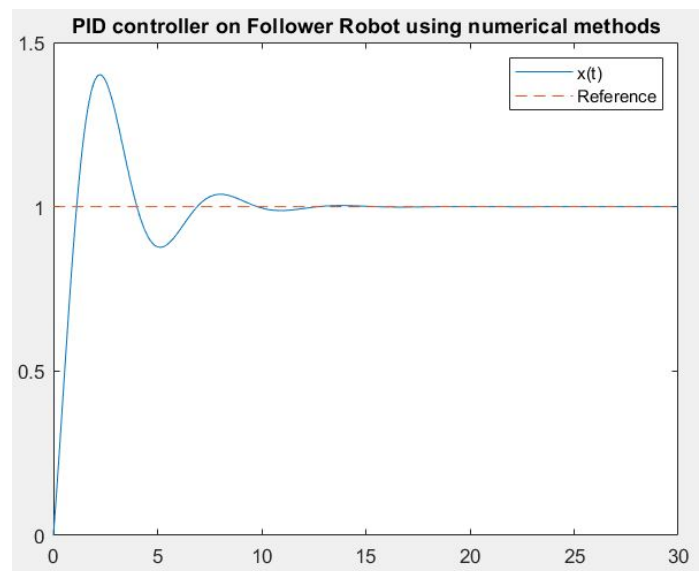

Fig. 8. Controller output signal at KP = 10.8, KI = 17.7 and KD = 3.2

V. CONCLUSION

As a conclusion for our progress in this report, we applied our knowledge and study of the applied numerical methods to perform a PID controller. So, we used Simpson's rule in order

to evaluate our accumulative integral error. Also, we used the backward difference method with two points to evaluate the differential accumulative error. We used this produced PID system to tune a mobile follower robot's motion to keep it at specific distance from an object -can be human, wall or even an obstacle-. As shown above in our analysis and results, we achieved our goal to tune the motion as accurate as possible.

VI. ACKNOWLEDGMENT

REFERENCES

[1] Steven C Chapra. Applied numerical methods with matlab for engineers and scientists. mcgraw-hill international edition. 2005.

[2] Taroh Matsuno. Numerical integrations of the primitive equations by a simulated backward difference method. *Journal of the Meteorological Society of Japan. Ser. II*, 44(1):76–84, 1966.

[3] William Marshall McKeeman. Algorithm 145: Adaptive numerical integration by simpson's rule. *Communications of the ACM*, 5(12):604, 1962.

[4] C CHAPRA STEVEN. *Applied Numerical Methods with Matlab: For Engineers and Scientists*. Tata McGraw Hill Education Private Limited, 2007.