# Mobile Robotics EE/CE 468

## Homework Assignment 01

Lyeba Abid - la07309
Ali Muhammad - aa07190

| Question: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | Total |
|---|---|---|---|---|---|---|---|---|---|
| Points: | 0 | 10 | 20 | 15 | 20 | 15 | 10 | 10 | 100 |
| Score: | | | | | | | | | |

**Problem 1** [**0 Points**] Setup

**Problem 2** [**10 Points**] **CLO-3/C-2**

**Solution:** Mobile Robot Control Box - kinematic Controller in the wheelSpeed function

```
function [phiDotL, phiDotR] = wheelSpeed(v, omega, waypoints,
    pose)

% Path constants
stopThreshold = 0.1;
slowThreshold = 0.3;

% Robot constants
trackWidth = 0.381;
wheelRadius = 0.195/2;

% Slow the robot down when it's near the threshold
distanceToEndpoint = norm(waypoints(end, :) - pose(1:2)');
if (distanceToEndpoint < slowThreshold)
v = distanceToEndpoint/slowThreshold*v;

% Stop the robot if it's inside the target threshold
if distanceToEndpoint < stopThreshold
    v = 0;
end
end

% Convert velocity and heading angular velocity to wheel
    speeds
phiDotL = (v - trackWidth/2*omega)/wheelRadius;
phiDotR = (v + trackWidth/2*omega)/wheelRadius;

end
```

Listing 1: wheelSpeed function

**Problem 3 [20 Points] CLO-3/C-3**

**Solution: PD Controller**

The PD Path algorithm we implemented is as follows:

```
function [LinVel, AngVel, phi_e, phi, phi_r, ye, x]  =
    pidPathFollower(Waypoints, Pose)
% A variable to keep track of the current waypoint
persistent k;
if isempty(k)
  k = 1;
end

x = Pose(1);
y = Pose(2);
phi = Pose(3);

% Update waypoint when it is near the existing waypoint
if (norm([x - Waypoints(k, 1), y - Waypoints(k, 2)]) <= 0.2)
  k = k + 1;
end

% Kp, Kd values for pd controller
Kp = 50;
Kd = 17;
LinVel = 0.3;

x1 = Waypoints(k - 1, 1);
y1 = Waypoints(k - 1, 2);
x2 = Waypoints(k, 1);
y2 = Waypoints(k, 2);

if (y1 == y2)
  yref = y1;
  xref = x;
elseif (x1 == x2)
  xref = x1;
  yref = y;
else
  % calculate slope and intercept for the path being followed
      between
  % waypoints and find the orthogonal line from the path to
      the robot
  m1 = (y2 - y1) / (x2 - x1);
  c1 = y1 - (m1 * x1);
  m2 = (-1 / m1);
  c2 = y - m2 * x;

  % calculate reference point at any instant of time
  xref = (c2 - c1) / (m1 - m2);
  yref = m1 * xref + c1;
end

% calculate error
xe = cos(phi) * (xref - x) + sin(phi) * (yref - y);
ye = -sin(phi) * (xref - x) + cos(phi) * (yref - y);

phi_r = atan2(Waypoints(k, 2) - Waypoints(k - 1, 2), Waypoints
    (k, 1) - Waypoints(k - 1, 1));
% phi_r = Waypoints(3); % used when heading is given in
    waypoints for
% circle and wave

% error in heading
phi_e = phi_r - phi;

% calculate kappa
curv = Kp * ye + Kd * sin(phi_e);
AngVel = LinVel * curv;
end
```

Listing 2: PD Controller Algorithm

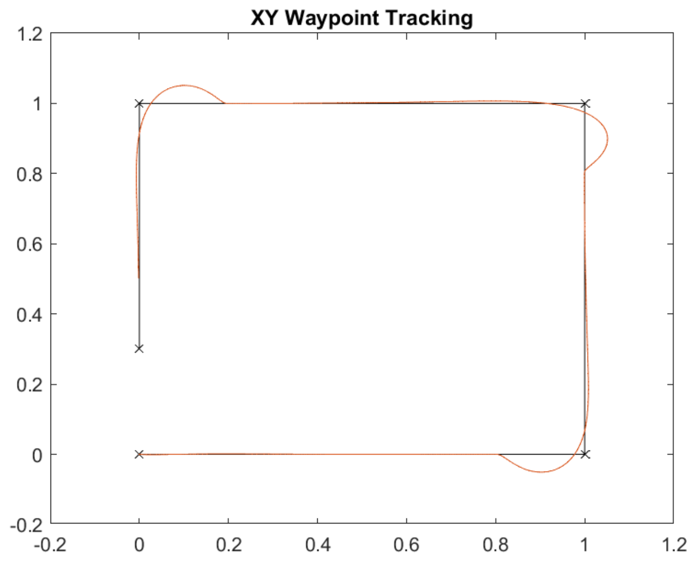


Figure 1: pidPathFollower on square path

The PID controller, with appropriately tuned $K_p$ and $K_d$ values, demonstrated remarkable performance on the provided trajectories - a circular path and a wave. For the circular trajectory, the $K_p$ value was set to a level that allowed precise tracking of the path, while the $K_d$ component aided in minimizing oscillations and ensuring stability. On the more complex wave trajectory, the balance between $K_p$ and $K_d$ allowed the controller to effectively follow the path, with $K_d$ playing a significant role in damping oscillations and maintaining accuracy. The PID controller's adaptability to these diverse trajectories showcases the importance of fine-tuning $K_p$ and $K_d$ to achieve responsive and stable control, albeit the need for potential adjustments in response to environmental dynamics or system changes.

**Problem 4 [15 Points] CLO-1/C-3**

**Solution: Path Following**

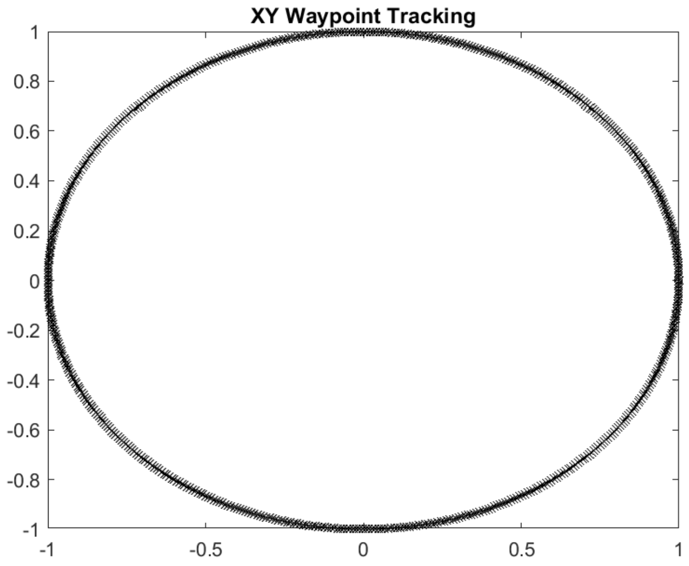


Figure 2: pidPathFollower on Circular path

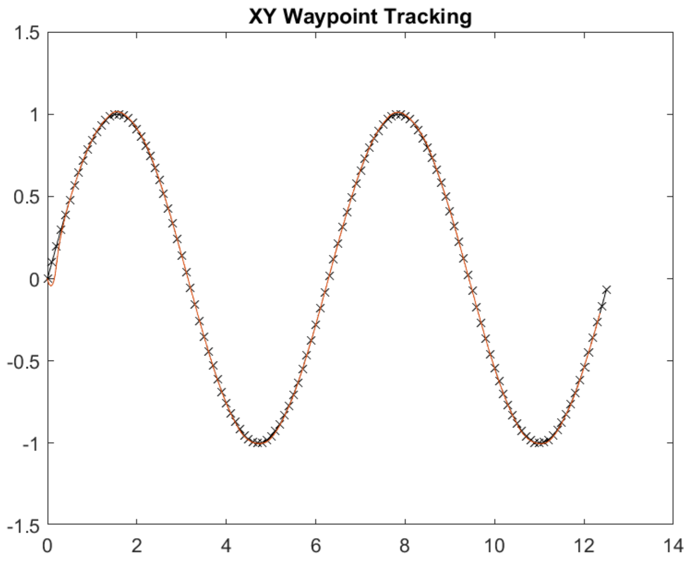


Figure 3: pidPathFollower on Sinusoidal wave

The PID controller, with appropriately tuned $K_p$ and $K_d$ values, demonstrated remarkable performance on the provided trajectories - a circular path and a wave. For the circular trajectory, the $K_p$ value was set to a level that allowed precise tracking of the path, while the $K_d$ component aided in minimizing oscillations and ensuring stability. On the more complex wave trajectory, the balance between $K_p$ and $K_d$ allowed the controller to effectively follow the path, with $K_d$ playing a significant role in damping oscillations and maintaining accuracy. The PID controller's adaptability to these diverse trajectories showcases the importance of fine-tuning $K_p$ and $K_d$ to achieve responsive and stable control, albeit the need for potential adjustments in response to environmental dynamics or system changes.

**Problem 5 [20 Points] CLO-1/C-3**

**Solution: Pure Pursuit Controller**

The Pure Pursuit Path algorithm we implemented is as follows:

```
function [LinVel, AngVel] = purePursuit(Waypoints, Pose)
  % Persistent variables to retain state information
  persistent lastGoal;
  persistent i;

  if (isempty(i))
    i = 1; % Initialize 'i' if it's empty
  end

  % Initialize 'lastGoal' with the robot's current position if
     it's empty
  if (isempty(lastGoal))
    lastGoal = Pose(1:2);
  end

  x = Waypoints(i, 1);
  y = Waypoints(i, 2);
  currentGoal = [x; y];

  LinVel = 1.0; %  Linear velocity value can be changed

  % Calculate the desired angular velocity using a custom
    controller
  w = customPurePursuitController(Pose, lastGoal, currentGoal,
     LinVel);

  % Check if the robot has reached the current goal point
  if (atGoalPoint(Pose, currentGoal))
    lastGoal = currentGoal;
    i = i + 1;
    x = Waypoints(i, 1);
    y = Waypoints(i, 2);
    currentGoal = [x; y];
  end

  AngVel = w;
end
```

Listing 3: Pure Pursuit Path Algorithm

The pure pursuit algorithm takes in the waypoints and the current pose of the robot and returns the linear and angular velocities. The algorithm uses a custom controller and some helper functions to calculate the angular velocity. The helper functions are as follows:

```
function [reached] = atGoalPoint(robotPose, goalPoint)
  goalRadius = 0.25;
  dist = sqrt((robotPose(1) - goalPoint(1))^2 + (robotPose(2)
     - goalPoint(2))^2);
  reached = abs(dist) < goalRadius;
end

function [angular_velocity] = customPurePursuitController(
   current_pose, last_point, current_point, linear_velocity)
  lookahead_distance = 1;

  error = calculateCrossTrackError(last_point, current_point,
     current_pose);

  goal_point = calculateGoalPoint(current_pose, last_point,
     current_point, lookahead_distance);

  vector_to_goal = goal_point - current_pose(1:2);
  alpha = (atan2(vector_to_goal(2), vector_to_goal(1)) -
     current_pose(3));

  curvature = 2 * sin(alpha) / lookahead_distance;

  angular_velocity = curvature * linear_velocity;
end

function [point] = calculateGoalPoint(current_pose,
   line_segment_start, line_segment_end, radius)
  if (distanceToGoalPoint(current_pose(1:2), line_segment_end)
      < radius)
    point = line_segment_end;
  else
    delta = line_segment_end - line_segment_start;
    vector_f = line_segment_start - current_pose(1:2);

    a = dot(delta, delta);
    b = 2 * dot(vector_f, delta);
    c = dot(vector_f, vector_f) - radius^2;
    discriminant = b^2 - 4 * a * c;

    point = [];
    point1 = [];
    point2 = [];

    if (discriminant >= 0)
      discriminant = sqrt(discriminant);
      t1 = (-b - discriminant) / (2 * a);
      t2 = (-b + discriminant) / (2 * a);

      if (t1 >= 0 && t1 <= 1)
        point1 = line_segment_start + t1 * delta;
        point = point1;
      end

      if (t2 >= 0 && t2 <= 1)
        point2 = line_segment_start + t2 * delta;
        point = point2;
      end
    end

    if (~isempty(point1) && ~isempty(point2))
      if (distanceToGoalPoint(point1, line_segment_end) <
          distanceToGoalPoint(point2, line_segment_end))
        point = point1;
      else
        point = point2;
      end
    end
  end
end

function [error] = calculateCrossTrackError(segment_start,
    segment_end, pose)
  projection = projectOntoLineSegment(segment_start,
      segment_end, pose(1:2));
  error = sqrt((projection(1) - pose(1))^2 + (projection(2) -
      pose(2))^2);
end

function [projected_point] = projectOntoLineSegment(
    segment_start, segment_end, point)
  segment_vector = segment_end - segment_start;
  point_vector = point - segment_start;
  projected_point = (dot(point_vector, segment_vector) / dot(
      segment_vector, segment_vector)) * segment_vector;
  projected_point = projected_point + segment_start;
end

function [distance] = distanceToGoalPoint(current_point,
   goal_point)
  distance = sqrt((current_point(1) - goal_point(1))^2 + (
     current_point(2) - goal_point(2))^2);
end
```

Listing 4: Helper Functions for the Pure Pursuit Algorithm

The below images show the outputs of the built-in MATLAB Pure Pursuit Block, and our implemented Pure Pursuit Block respectively.

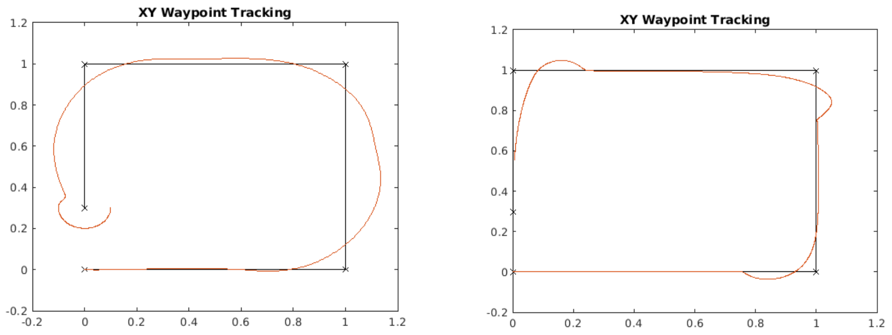


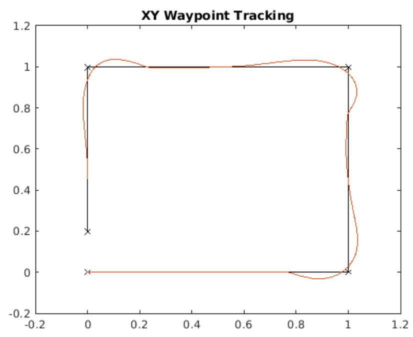
Figure 4: MATLAB Pure Pursuit (left) and Our Pure Pursuit Block (right)

For both the above figures, the linear velocity was 0.1 (for MATLAB, 0.1 was not working for some reason so this is of 0.5), angular velocity was 1.0, and look ahead distance had been set to 0.5. This clearly shows our pure pursuit algorithm to be performing better than MATLAB's algorithm. However, our algorithm performs better when linear velocity is low, on higher velocities it doesn't perform as much well.

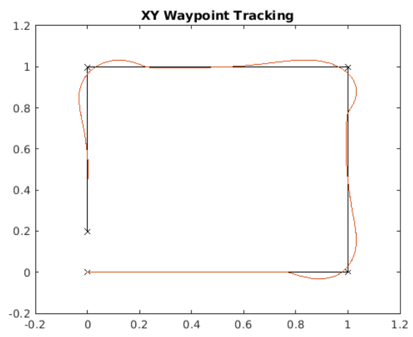
We varied the linear velocity and look ahead distance to see how the robot performs under different parameters. The below sections show the results of our algorithm for different linear and look ahead distances.

## Varying Linear Velocity

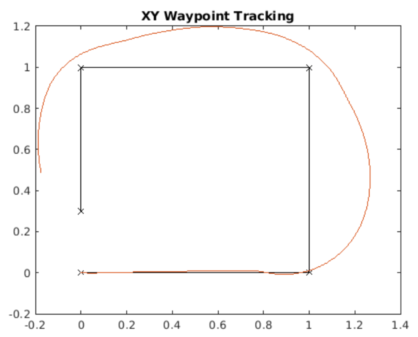
By keeping the lookahead distance constant (at 0.75) and varying the linear velocity to 0.2, 0.3, 0.5, and at 1.0 respectively, we get the following results:



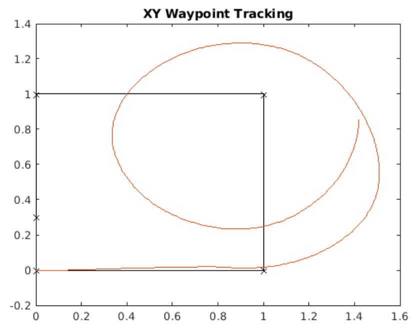
(a) Linear Velocity = 0.2



(b) Linear Velocity = 0.3



(c) Linear Velocity = 0.5



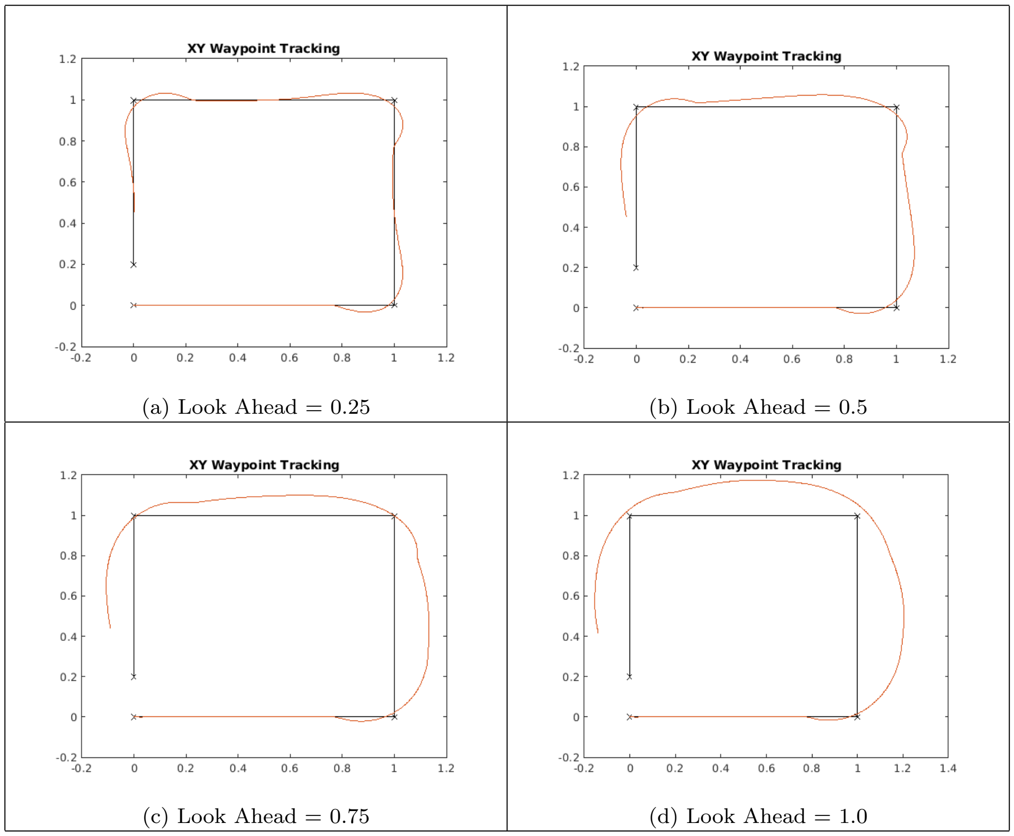
(d) Linear Velocity = 1.0

Figure 5: Pure Pursuit Algorithm with Different Linear Velocities

From the above, we concluded that we need to keep the linear velocity at an optimal value such that it is not too high; in which case the robot overshoots the path on turnings. The best linear velocity from the above was 0.2 and 0.3, which is the default value in our algorithm.

### Varying Look Ahead Distance

We varied the look ahead distance to 0.25, 0.5, 0.75, and 1.0 respectively, and kept the linear velocity constant at 0.3. The results are as follows:



(a) Look Ahead = 0.25 (b) Look Ahead = 0.5

(c) Look Ahead = 0.75 (d) Look Ahead = 1.0

Figure 6: Pure Pursuit Algorithm with Different Look Ahead Distances

From the above, we concluded that having larger look ahead distances resulted in the robot computing much longer arcs to get to the goal point. This resulted in the robot overshooting the path and going off the track. Thus, having smaller look ahead distances was optimal, but not too small either in which case the robot would oscillate a little bit around the path. So the optimal look ahead distance according to our algorithm is 0.5.

Keeping the above in mind, we concluded on linear velocity of 0.3 and look ahead distance of 0.5 to be the optimal values for our algorithm.

In comparison with the PD Controller, our Pure Pursuit did not follow the path as accurately or closely as the PD Controller. A reason for this could be since the PD Controller directly adjusts the control based on the error between the desired path and the actual robot position while the Pure Pursuit relies on geometry and a constant look ahead distance.

Tradeoffs while considering both algorithms would definitely include the amount of accuracy required for the path following that the robot requires. Smoothness can also be a factor as Pure Pursuit had a smoother path. Complexity to implement the algorithm could also be a tradeoff since Pure Pursuit required more calculations and geometry to implement while PD Controller was simpler, and shorter without those complex calculations.

**Problem 6 [15 Points] CLO-1/C-3**

**Solution: Odometry**

```
function [pose, x_next, y_next, phi_next] = Odometry(x_k, y_k,
    phi_k, wheelspeeds)
  display(wheelspeeds)

  % Robot constants
  trackWidth = 0.381;
  wheelRadius = 0.195/2;

  % Calculate v and omega
  v_sol = wheelRadius * (wheelspeeds(1) + wheelspeeds(2)) *
     0.5;
  omega_sol = -wheelRadius * (wheelspeeds(1) - wheelspeeds(2))
      * (1/trackWidth);

  Ts = 0.01;
  x_next = x_k + Ts * v_sol * cos(phi_k);
  y_next = y_k + Ts * v_sol * sin(phi_k);
  phi_next = phi_k + Ts * omega_sol;

  pose = [x_next, y_next, phi_next];
end
```

Listing 5: Odometry Algorithm

**Problem 7 [10 Points] CLO-1/C-3**

**Solution: Using Odometry**

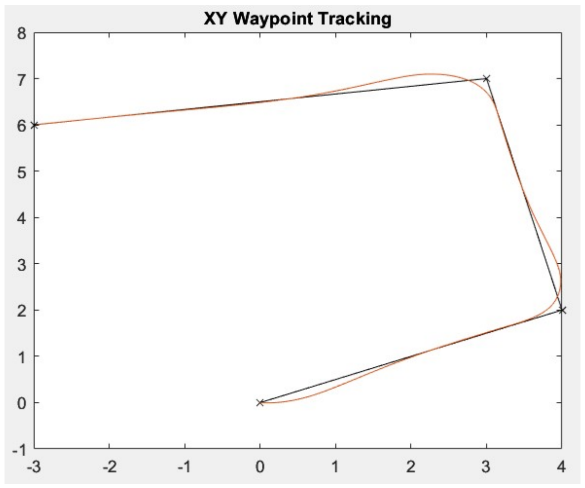


Figure 7: Pose obtained from Gazebo sensors

The depicted figure illustrates the path obtained when the pose data was acquired from Gazebo sensors. Below is the pose output as observed in the Gazebo sensor data.

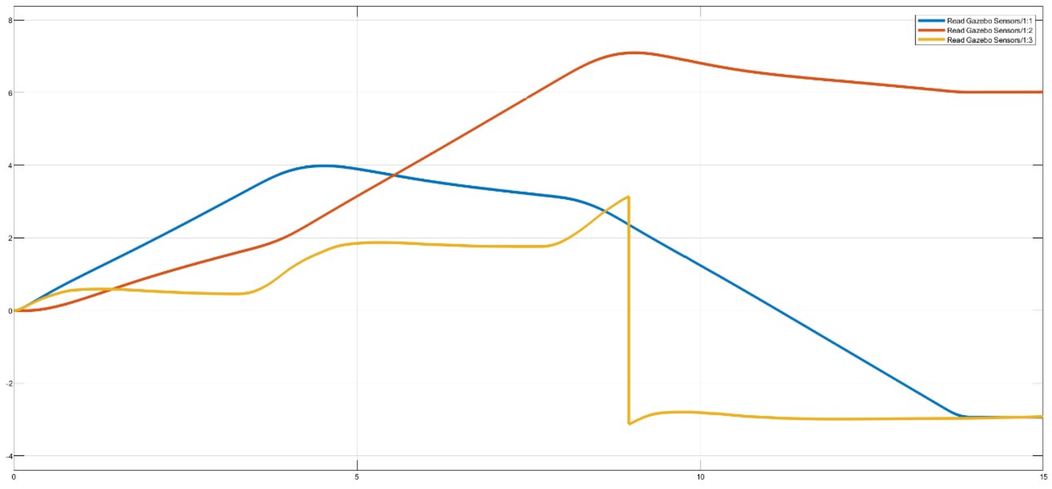


Figure 8: Pose obtained from Gazebo sensors

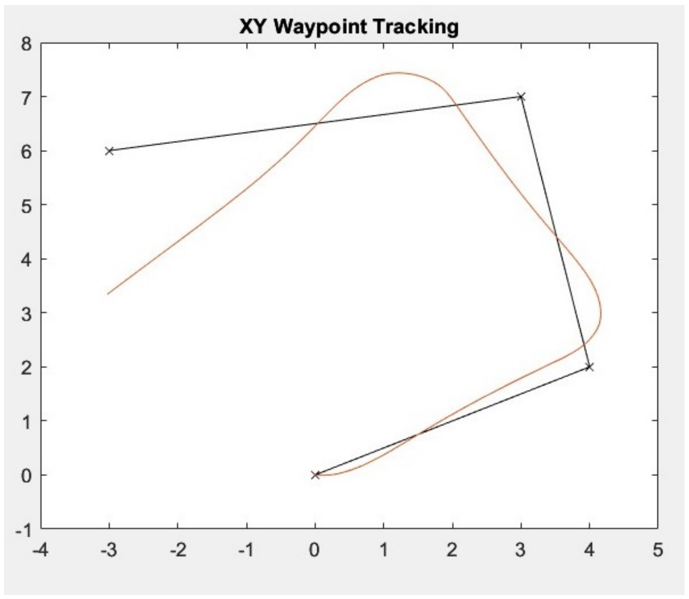


Figure 9: Pose obtained from Gazebo sensors

The figure above illustrates the path derived from odometry-based pose calculations. Below, we present the corresponding pose data obtained from the Odometry Block on the scope.

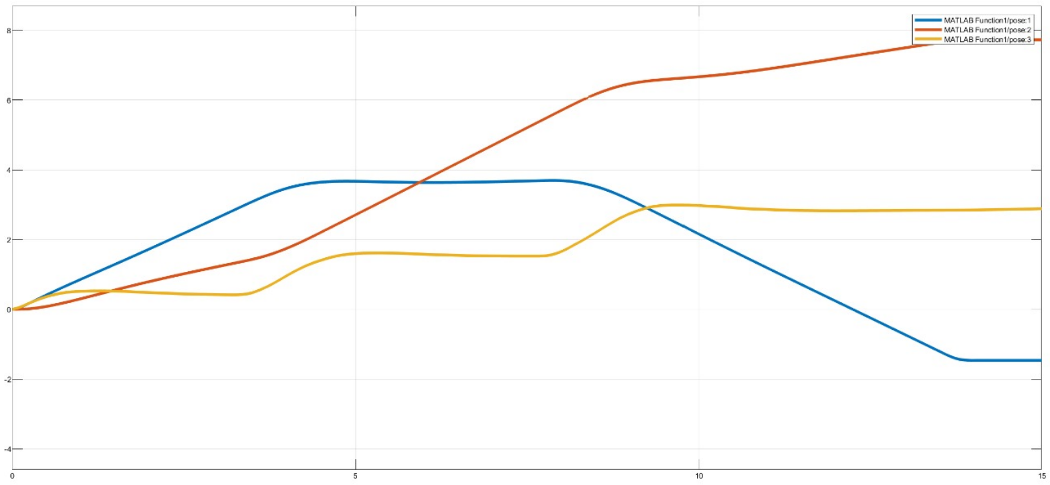


Figure 10: Pose obtained from Gazebo sensors

The discrepancy in odometry output, as evidenced by the two plots above, is more pronounced during turns compared to when the robot follows a linear path. Initially, the robot accurately adheres to its intended course. However, as it encounters significant changes in the heading angle, there is a noticeable increase in error. A thorough examination of the pose graph through the utilization of a Simulink scope has revealed that the most significant deviation is concentrated in the heading angle ($\phi$), while the $x$ and $y$ coordinates closely match those obtained from Gazebo sensors.

The reason for these differences in odometry calculations can be attributed to the recursive nature of the equations we employed:

$$\begin{aligned}
x_{\text{next}} &= x_k + Ts \cdot v_{\text{sol}} \cdot \cos(\phi_k) \\
y_{\text{next}} &= y_k + Ts \cdot v_{\text{sol}} \cdot \sin(\phi_k) \\
\phi_{\text{next}} &= \phi_k + Ts \cdot \omega_{\text{sol}}
\end{aligned}$$

In essence, we employed integration using the Riemann sum method, with a sampling time ($Ts$) of 0.01 seconds, the default value in MATLAB. As this time step is not infinitesimally small, discrepancies in odometry calculations are unavoidable, resulting in less precise results.

The challenge posed by these discrepancies during turns is a common issue in odometry calculations. Beyond the sampling time, other factors may contribute to differences in odometry outputs. These could include variations in wheel diameters, tire slippage, wheel calibration inaccuracies, and sensor noise. Each of these factors can introduce errors in the odometry process, particularly when the robot undergoes complex maneuvers.

**Problem 8 [10 Points] CLO-1/C-2**

**Solution:**

**Lyeba Abid**

(a) 40 hours

(b) I focused on refining the PD path algorithm and implementing odometry for the assignment.

(c) In contrast to some of my peers, setting up the software didn't take much time; however, I faced a learning curve in grasping Simulink and comprehending the assignment requirements and implementation details. Once I got the hang of it, the implementation became easier. Throughout this assignment, I gained valuable insights into the significance of $K_p$ and $K_d$ in control systems, particularly their role in error correction. I also deepened my understanding of the practical implementation of the pure pursuit algorithm. I linked these concepts to my prior knowledge, recognizing their crucial role in achieving precise robot navigation.

**Ali Muhammad**

(a) 47 hours

(b) We both engaged in collective discussion over the questions and the topics, however, I mainly worked on Pure Pursuit and Odometry.

(c) A lot of time was spent on the setting up of the software, mainly Gazebo, and integrating it with the system. Although the guides were helpful, they usually only catered to a particular user. As someone using an Ubuntu-based Linux Distro, I was confident it would work on my machine, however, it later turned out that the GazeboPlugins wouldn't build on my distro, ultimately leading me to install a VMware and run Gazebo on a VM.

In addition, being a CS student, MATLAB and Simulink were also very new to me which further increased the overhead resulting in the huge amount of time taken for this homework.

As far as the questions are concerned, it took some time to get the hang of it, but once that was out of the way, the questions were not very difficult (they were still hard though). I mainly enjoyed the Pure Pursuit algorithm; the concept is quite simple, but the geometry took some time to get comfortable with, and also took reference to quite a few online resources. Odometry was also not that difficult to implement once the equations and formulae were sorted out - taken from the slides. It was interesting to see the robot taking sensor data and then using the sensor data to calculate its pose for computations later on.