Path Following with Obstacle Avoidance

EE615 Control and Computational Laboratory

Tejas Sanjaykumar Pagare | 190070067

Rathour Param Jitendrakumar | 190070049

Spring Semester 2022-23

Contents

1 Introduction

1.1 Aim

The purpose of this experiment is to understand and implement two most important algorithms in mobile robot kinematics called Pure Pursuit and Vector Field Histogram on a Differential Drive Robot.

1.2 Background

With the development of automation, robots have quickly replaced many industrial jobs especially at the assembly line where these robots can repeatedly do tasks as programmed without changing their location. But development of mobile robots with large range of motion comes with its own problems such as unknown environments, motion planning, tracking and control.

For the purpose of this experiment, we have a localised environments and the motion is pre-planned using a set of waypoints. Hence, the task of robot is to track the path while avoiding obstacles.

Coordinate Systems

We follow two coordinate systems as shown in 1. In inertial frame the robot's pose is given by $[x, y, \theta]$ where θ is the counterclockwise orientation of the robot. In robot's frame the same robot pose is $[0, 0, \theta]$ as robot is now at the origin. The coordinate conversion (of position/velocity) from inertial frame to robot frame can be done by multiplying

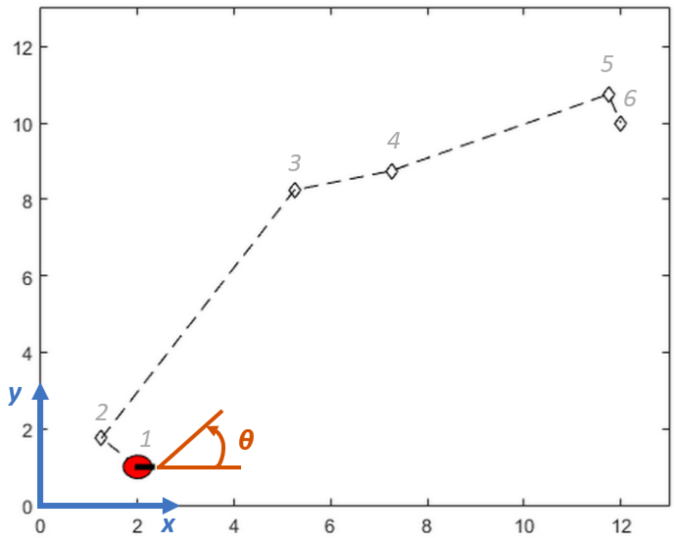


Figure 1: Coordinate System (From [3])

the inertial coordinates with the following rotation matrix

$$R(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

In a Differential Drive robot, the two standard wheels can move at different velocities $\dot{\phi}_1$ and $\dot{\phi}_2$.
We denote the linear velocity of robot as u and the angular velocity as ω and define them as follows

$$u = \frac{r(\dot{\phi}_1 + \dot{\phi}_2)}{2} \quad \text{and} \quad \omega = \frac{r(\dot{\phi}_1 - \dot{\phi}_2)}{2l} \tag{2}$$

where r is the radius of a wheel and l is the distance of centre of robot to a wheel.

2 Methodology

For simulation, Robot Operating System (ROS) is used to send and receive data from the MATLAB simulator. ROS Toolbox need to be installed along with Simulink. The simulator receives and sends messages at the following ROS topics:

- Velocity commands on the /mobile_base/commands/velocity topic
- robot pose information at /ground_truth_pose topic
- laser range data at the /scan topic

After receiving the respective information the data is fed to the simulink blocks of Pure Pursuit and Vector Field Histogram algorithm which uses several MATLAB functions.

3 Pure Pursuit Algorithm

3.1 Algorithm

Input and Output The Pure Pursuit simulink block accepts two inputs, which are pose and waypoints. Pose denotes the current robot position given as a tuple (x, y, θ) where which corresponds to the $x - y$ position and the orientation of the robot. Again, positive angles are measured counterclockwise from the positive x-axis. Waypoints is a n-by-2 array consisting of the waypoints the robot should follow, where the last waypoint is its final goal position. Output of the block is linear and angular velocity calculated using radius of curvature and the target direction, the direction to the goal point from the current location. The forward direction of the robot is 0 radians with positive angles measured counterclockwise.

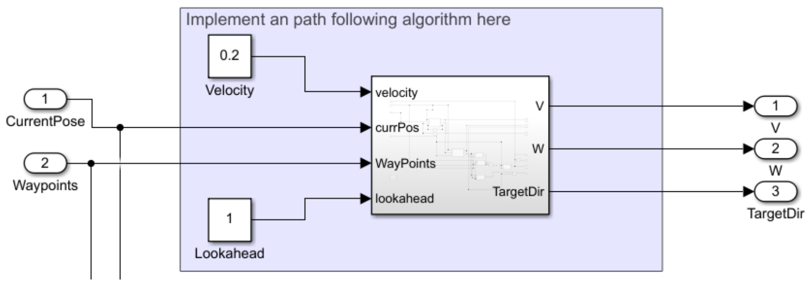


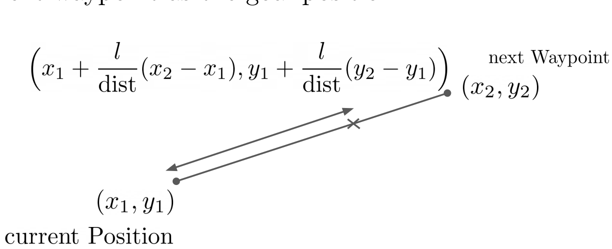
Figure 2: Pure Pursuit Implementation

Next waypoint The algorithm can be divided into 2 stages the initialization and the update.

In the initialization step, we find the waypoint closest to the robot. We want the robot to go through all the next waypoints on the path from this waypoint. This ensures smooth functioning with any valid random start location.

In the update step, we output the closest next waypoint from the current location.

Goal Point Here, we calculate the goal point as the point along the direction of next waypoint at a lookahead distance (l). If the distance between the next waypoint and the current position is less than the lookahead distance (i.e. dist$< l$), then we take the next waypoint as the goal position.



$$\text{dist} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Figure 3: Goal Position

Target Direction Target direction is the angle in radians the goal position makes with the robot's axis, which is positive counterclockwise and negative clockwise. For goal position (x_g, y_g), current location (x_1, y_1) and robot angle θ this can be calculated as follows

$$\theta_{\text{targ}} = \tan^{-1}\left(\frac{y_g - y_1}{x_g - x_1}\right) - \theta \tag{3}$$

Following figure illustrates the idea

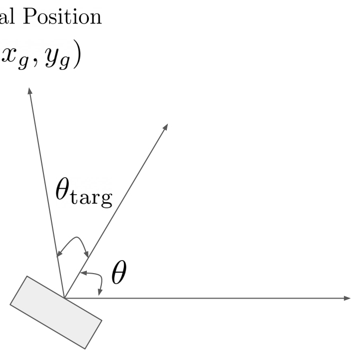


Figure 4: Target Direction

Radius of Curvature To calculate radius we first convert the goal position (x_g, y_g) and current position (x_1, y_1) of the robot from the inertial frame to robot frame to get (x'_g, y'_g) and (x'_1, y'_1) resp.

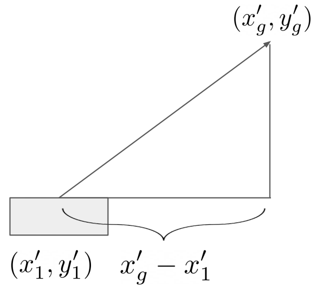


Figure 5: Radius of Curvature

From here we calculate the radius of curvature r as follows

$$r = \frac{l^2}{2|x'_g - x'_1|} \tag{4}$$

where x'_g and x'_1 are as defined in the Fig. (5) and l is the lookahead distance.

Linear Velocity We use a constant linear velocity of $v = 0.2\ ms^{-1}$.

Angular Velocity Using linear velocity v and radius of curvature r we calculate angular velocity ω as follows

$$\omega = v \times r \tag{5}$$

3.2 Implementation

Next waypoint The above two ideas were implemented by same matlab function by using feedback i.e., passing its output as its next input. For this we used a memory block which stores the feedback value and also helps during initialisation.

```
function [nextWayPoint, idx_next, init_next] = nextWayPoint(currPos,WayPoints,idx, init)
init_next = init
if init == 0
    distances = pdist2(currPos(1:2)',WayPoints(idx,:));
    [minDistance, minidx] = min(distances);
    idx = minidx
    init_next = 1
end
idx_next = idx
if norm(currPos(1:2)'-WayPoints(idx,:))<0.1
    idx_next = min(idx + 1, size(WayPoints, 1))
```

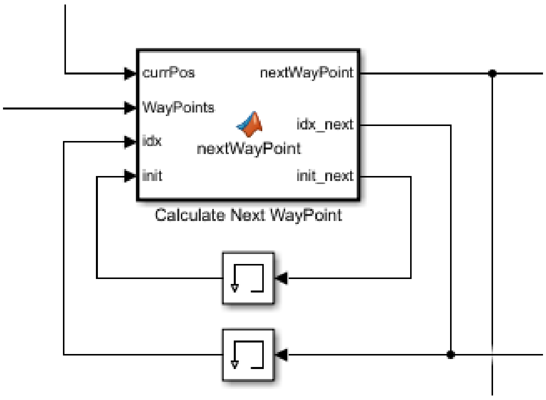


Figure 6: Calculate Next Waypoints Block with Memory elements

```
end
nextWayPoint = WayPoints(idx_next,:);
end
```

Goal Point

```
function [xout,yout] = getGoalPos(currPos,nextWayPoint,lookahead)
x1 = currPos(1);
y1 = currPos(2);
x2 = nextWayPoint(1);
y2 = nextWayPoint(2);
dist = pdist([x1,y1;x2,y2],'euclidean');

if dist<lookahead
    xout = x2;
    yout = y2;
else
    xout = x1+(lookahead/dist)*(x2-x1);
    yout = y1+(lookahead/dist)*(y2-y1);
end
```

Radius of Curvature

```
function x_R  = rotation(x_I,y_I,theta_I)
x_R = x_I*cos(theta_I)+y_I*sin(theta_I);
```

Target Direction

```
function theta = targetDir(xg,yg,xc,yc,theta_c)
theta = angdiff(theta_c,atan2(yg-yc,xg-xc));
```

We use the simulink blocks of subtraction, division, absolute and square to calculate the radius of curvation.

4 Vector Field Histogram

Input and Output The input received to the obstacle avoidance block is the ranges and angles where the corresponding range and angle are the distance and angle of the obstacle from the robot's current location. We also receive target direction as calculated by the pure pursuit algorithm.
The output of the block should be the obstacle-free angle of rotation from the robot's current position in radians where the forward direction of robot is zero with positive angles measured counterclockwise.

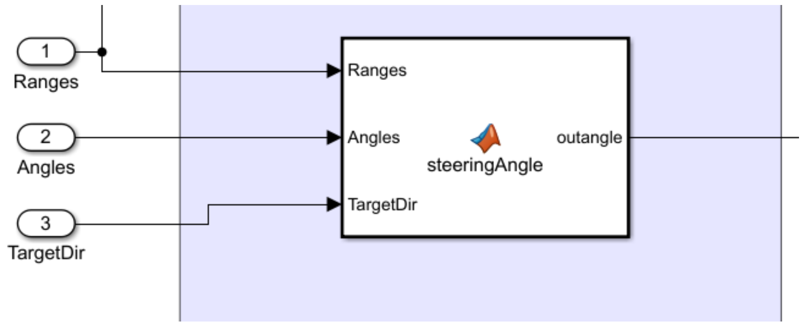


Figure 7: Vector Field Histogram Implementation

We use the simpler version of Vector Field Histogram algorithm from [1] as Approach 1 and a modification of it in Approach 2. Given the ranges and angles both approaches first calculate magnitude of the obstacle vector for all the cells in the $w_s \times w_s$ window. For the (i, j) th cell, $m_{i,j}$ is calculated as

$$m_{i,j} = (c_{i,j})^2(a - bd_{i,j}) \tag{6}$$

where a, b are constants and $c_{i,j}$ is the certainty value and $d_{i,j}$ is the distance of (i, j) th cell from the center of the robot.
[1] suggests to set a and b such that $a - bd_{\max} = 0$ where $d_{\max} = \sqrt{2}(w_s - 1)/2$. We consider 5×5 window around the robot, hence $d_{\max} = 2\sqrt{2}$, taking $b = 1$, we get $a = d_{\max} = 2\sqrt{2}$. For simplicity we take $c_{i,j} = 1$ for all the cells. However, [1] proposes an algorithm which increments certainty value for a single cell along the acoustic axis at the measured distance, making it efficient and computationally inexpensive. Thus $m_{i,j}$ is then calculated as

$$m_{i,j} = 2\sqrt{2} - d_{i,j} \tag{7}$$

After having calculated we make a histogram for *polar obstacle density* with bin width $\alpha = 360/n$ and n bins, where kth bin has value H_k calculated as

$$H_k = \sum_{i,j} \mathbb{1}_{i,j} m_{i,j} \tag{8}$$

where $\mathbb{1}_{i,j} = 1$ if angle corresponding to the (i, j)th cell lies in the kth bin else 0.

Histogram Smoothing The histogram might have errors due to missing or inaccurate data and thus may appear ragged. To solve this we use histogram smoothing to get the *smoothed polar obstacle density* H'_k as

$$H'_k = \frac{H_{k-l} + 2H_{k-l+1} + \cdots + lH_k + \cdots + 2H_{k+l-1} + H_{k+l}}{2l + 1} \tag{9}$$

We took $l = 3$, which gave sufficient smoothing for 60 bins.

We now describe the two approaches used

4.1 Approach 1

4.1.1 Algorithm

After completing the common procedure till the first smoothing, we follow the below approach

Valley Detection Having made the histogram, we now detect valleys. We describe valleys as the bins having H_k below some threshold. We call a valley a *candidate valley* if the valley width is wide enough. To do this, we simply iterate over all the bins, and identify the candidate valleys.
After getting all the candidate value, we choose the one which is nearest to the target direction θ_{targ}.

Steering Angle Once we find the nearest valley, we set the steering angle as the linear combination of the boundary angles of that valley. The higher weight $p = 0.8$ is given to the boundary closest to the target direction.

In case, we don't find any valley or candidate valley, we set the steering angle to some arbitrary angle close enough to 0 radians.

4.1.2 Implementation

```
function outangle = steeringAngle(Ranges,Angles,TargetDir)
% Filtering Invalid Data
badindices = bitor(isnan(Ranges),isnan(Angles));
Ranges = Ranges(~badindices);
Angles = Angles(~badindices);
% Calculating Histogram Values
a = 2*sqrt(2);
b = 1;
Ranges = a-b*Ranges;
numBins = 180;
edges = linspace(-pi,pi,numBins+1);
[~,edges,bin]=histcounts(Angles,edges);
B = accumarray(bin,Ranges,[numBins,1]);
[~,~,idx]=histcounts(TargetDir,edges);
B = conv(B,1/7*[1 2 3 2 1], 'same')
thresh = 2.5
minbins = find(B<thresh); % bin numbers of the valleys
if isempty(minbins) % in case of no valleys
    outangle = pi/10;
else
    % Valley Calculation
    C = {};
    x = [];
    for i=1:size(B,1)
        if B(i)<thresh
            x = [x,i];
            if i==size(B,1)|B(i+1)>=thresh
                if length(x)>=4
                    C{end+1}=x;
                end
                x = [];
            end
        end
    end
    % Valley Detection
    if isempty(C) % in case of no candidate valleys
        outangle = pi/10;
    else
        [y1,~] = min(abs(C{1}-idx));
        closevalley = 1;
        for j=2:length(C)
            [y2,~] = min(abs(C{j}-idx));
            if y2<y1
                y1 = y2;
                closevalley=j;
            end
        end
        p = 0.8;
        q = 10;
        if (C{closevalley}(1)-idx)*(C{closevalley}(end)-idx)<0
            outangle = TargetDir;
        elseif abs(C{closevalley}(1)-idx)<abs(C{closevalley}(end)-idx)
            if length(C{closevalley})>=q
                outangle = (p*edges(C{closevalley}(1))+(1-p)*edges(C{closevalley}(q)))/2;
            else
                outangle = (p*edges(C{closevalley}(1))+(1-p)*edges(C{closevalley}(end)))/2;
            end
        else
            if length(C{closevalley})>=q
```

```
60                  outangle = ((1-p)*edges(C{closevalley}(1))+p*edges(C{closevalley}(q)))/2;
61              else
62                  outangle = ((1-p)*edges(C{closevalley}(1))+p*edges(C{closevalley}(end)))/2;
63              end
64          end
65      end
66  end
```

If statement at line 49 sets steering angle to the target direction if the target direction lies in the valley. p is the linear combination parameter and q is a parameter which sets the maximum valley width while calculating the target direction.

4.2 Approach 2

After completing the common procedure till the first smoothing, we follow the below approach

4.2.1 Algorithm

Valley Detection One way is to not go from the middle of valley as that was resulting in the vehicle going too much away from target direction. So, the robot goes inside the valley in an uneven fashion which is closer to target direction. We took the convex combination of the selected valley boundary with parameter 0.1. This means the robot might be closer to boundaries of valley and thus obstacles.

We use a threshold for obstacle detection, as obstacles can change rapidly, by smoothing out the histogram again, the resulting valleys will be smaller and thus we ensure that the boundaries of valleys are clear to pass.

Now, we find the valley that is closest of target direction. First, we find the point which is closest to target direction and then find the valley this point belongs to by incrementing/decrementing (suitably) indices of this point.

If there is no valley, we always rotate by $pi/10$ in hope of finding new valleys.

Also, we take p of approach $1 = 0.1$ for our case.

4.2.2 Implementation

```
function outangle = steeringAngle(Ranges,Angles,TargetDir)
% Filtering Invalid Data
badindices = bitor(isnan(Ranges),isnan(Angles));
Ranges = Ranges(~badindices);
Angles = Angles(~badindices);
% Calculating Histogram Values
a = 15;
b = 1;
Ranges = a-b*Ranges;
numBins = 60;
edges = linspace(-pi/2,pi/2,numBins+1);
validindices = bitand(Angles >= -pi/2, Angles <= pi/2);
Ranges = Ranges(validindices);
Angles = Angles(validindices);
[~,edges,bin]=histcounts(Angles,edges);
B = accumarray(bin,Ranges,[numBins 1]);
% Smoothing Histogram Values
B = conv(B,1/7*[1 2 3 2 1], 'same');
% Considering edgecases of target direction
[~,~,idx]=histcounts(TargetDir,edges);
idx = max(1,idx);
idx = min(idx,numBins);
% Valley Calculation
threshold = 15
Bnew = conv(B,1/7*[1 1 1 1 1 1 1], 'same');
indices = find(Bnew<=threshold);
if isempty(indices)
    outangle = pi/18
```

```
else
    % Valley Detection
    [minval, minidx] = min(abs(indices-idx));
    startx = 1;
    endx = size(indices,1);
    if indices(minidx)>=idx
        startx = indices(minidx);
        i = indices(minidx);
        while (Bnew(i)<=threshold & i<size(Bnew))
            endx = i;
            i = i+1;
        end
        outangle = (0.9*edges(startx)+0.1*edges(endx+1))/2;
    else
        endx = indices(minidx);
        i = indices(minidx);
        while (Bnew(i)<=threshold & i>1)
            startx = i;
            i = i-1;
        end
        outangle = (0.1*edges(startx)+0.9*edges(endx+1))/2;
    end
end
```

5 Simulation and Modeling

5.1 Simulation Environment and Model structure

We use MATLAB and Simulink version R2020a and the ROS, Navigation toolboxes. Here we describe the Simulink blocks, the overall model is divided into 4 subsystems

```
open_system('pathFollowingWithObstacleAvoidanceExample');
```

1. Process Inputs The input is received at two subscribers. The first subscriber receives the data sent on the /scan topic which is used to extract the ranges and angles to be used in the VFH block. The second subscriber receives message on the /ground_truth_pose topic consisting of the robot (x, y) position and the orientation.

```
open_system('pathFollowingWithObstacleAvoidanceExample/Inputs','tab');
```

2. Compute Velocity and Heading for Path Following This subsystem is aimed to calculate the linear and angular velocity along with the target direction the robot shows move towards from the pure pursuit algorithm. This also consists of a block which is used to stop the robot when it is close to the final goal position.

```
open_system('pathFollowingWithObstacleAvoidanceExample/
    Compute Velocity and Heading for Path following','tab');
```

3. Adjust Velocities to Avoid Obstacles The linear and angular velocity computed by the path follower is used to calculate the obstacle-free steering angle closest to the target direction so that we still follow the path.

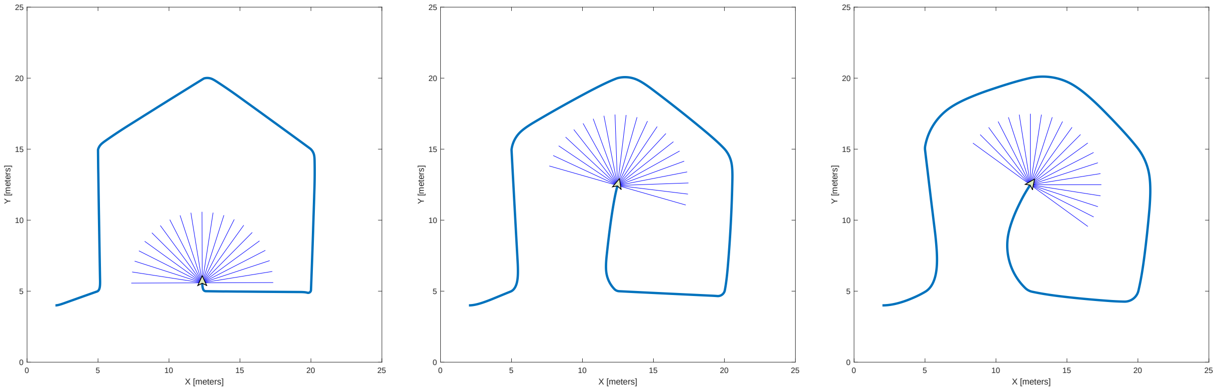
```
open_system('pathFollowingWithObstacleAvoidanceExample/
    Adjust Velocities to Avoid Obstacles','tab');
```

4. Send Velocity Commands The 'Output' subsystem is publishes the linear and angular velocity are published on the /mobile_base/commands/velocity ROS topic. This subsystem also has a special property that it publishes the velocity only when a new sensor information is available which prevents hitting obstacles in case of delay in receiving information.

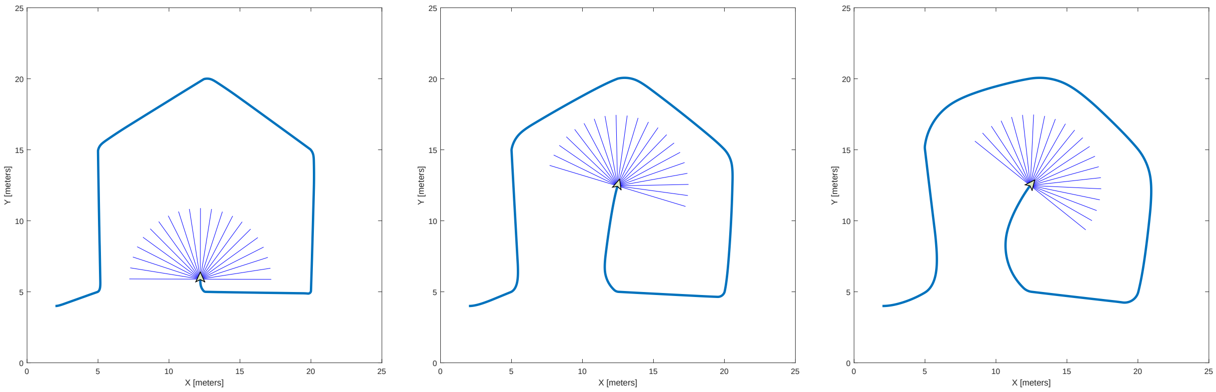
```
open_system('pathFollowingWithObstacleAvoidanceExample/Outputs','tab');
```

5.2 Simulation Results

5.2.1 Pure Pursuit



(a) $l = 0.2$m and $v = 0.2$ms^{-1} (b) $l = 0.2$m and $v = 0.5$ms^{-1} (c) $l = 0.2$m and $v = 1$ms^{-1}



(d) $l = 1$m and $v = 0.2$ms^{-1} (e) $l = 1$m and $v = 0.5$ms^{-1} (f) $l = 1$m and $v = 1$ms^{-1}

Figure 8: Variation of Paths with Lookahead Distance l and Linear Velocity v
The path followed is [5 5; 5 15; 12.5 20; 20 15; 20 5; 12.5 5; 12.5 12.5]

For smaller velocities, the path is more accurate and as we increase velocities the path is not followed as accurately as previous. The curvature has increased as the robot faces difficuly in turning for each lookahead point.

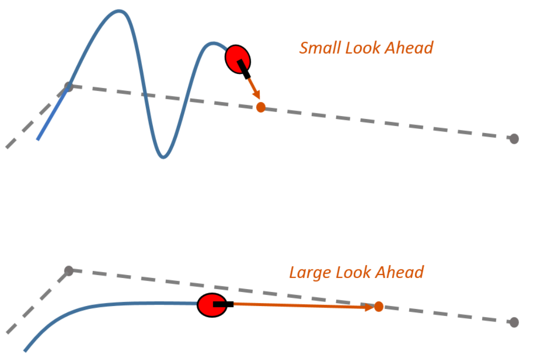


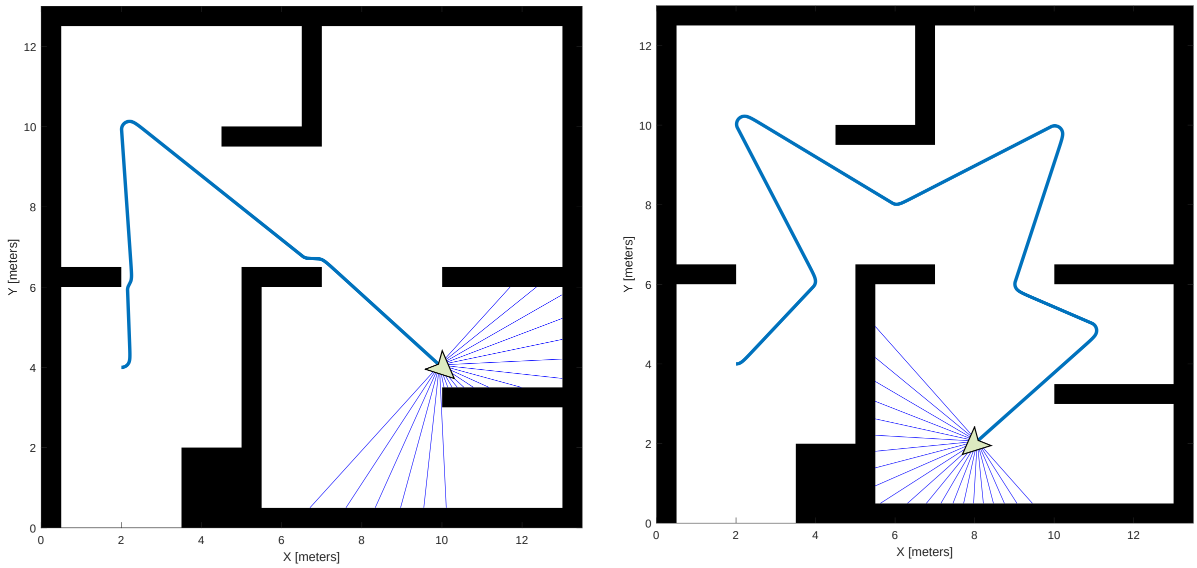
Figure 9: Variation with Lookahead distances

The change in lookahead distance is not making any significant changes as the distance between waypoints is large, so for both the above lookahead distances values the goal point actually lie on path.

But the major explanation is that even though lookahead distance is large, we are not fixing a goal point and going towards it and then recalculating goal point; the goal point is just an instantaneous point so if we stray away from path, the algorithm with self correct before reaching the calculated goal point by calcualaing new goal point.

5.2.2 VFH

Inbuilt VFH Module

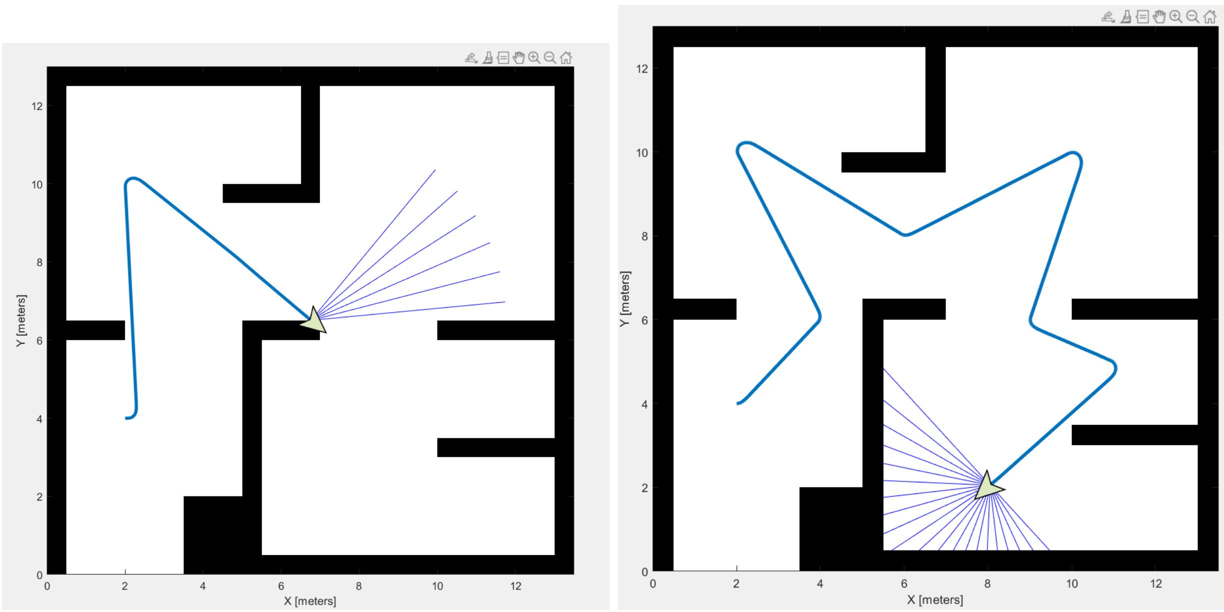


(a) For path [2 4; 2 10; 10 4]

(b) For path [2 4; 4 6; 2 10; 6 8; 10 10; 9 6; 11 5; 8 2]

Figure 10: Inbuilt Vector Field Histogram $VFH+$ block for different paths with our pure pursuit algorithm

Approach 1



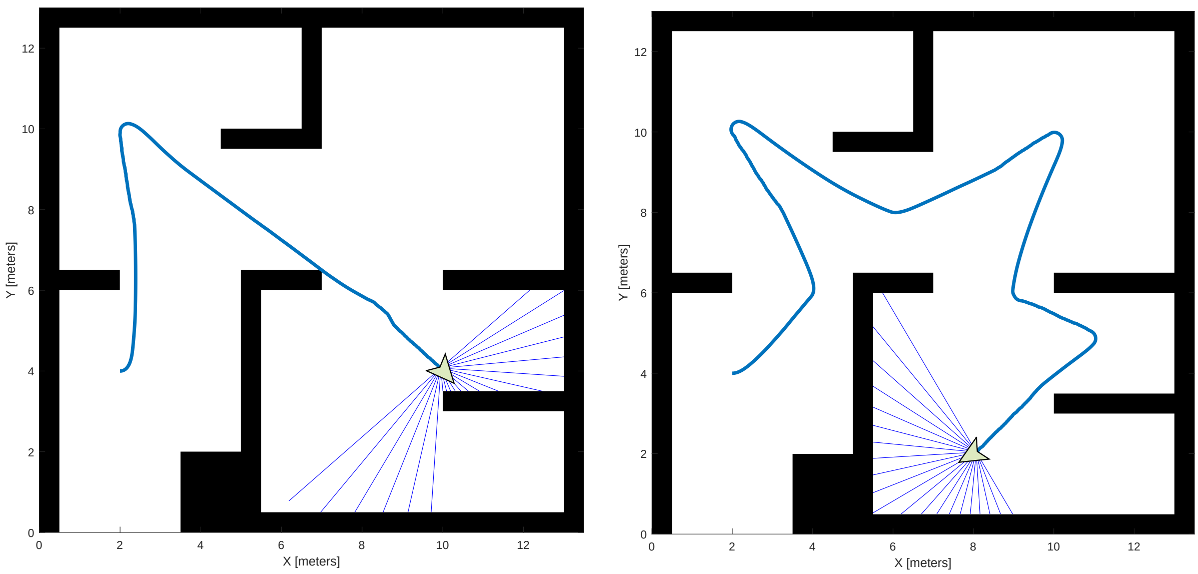
(a) For path [2 4; 2 10; 10 4]

(b) For path [2 4; 4 6; 2 10; 6 8; 10 10; 9 6; 11 5; 8 2]

Figure 11: Vector Field Histogram for different paths using approach 1

Here we observe that for first path that the robot collapses with the obstacle, the possible reasons of which can be sensitivity towards the threshold value used to infer the candidate values.

Approach 2



(a) For path [2 4; 2 10; 10 4]

(b) For path [2 4; 4 6; 2 10; 6 8; 10 10; 9 6; 11 5; 8 2]

Figure 12: Vector Field Histogram for different paths using approach 2

Inbuilt system avoids obstacles accurately, whereas approach 1 couldn't and in approach 2 the robot just turns enough to avoid obstacle. This was expected as we are not considering safety direction parameter. Approach 2 though works but the robot flickers a lot during its motion. This can be seen as long straight paths between waypoints are not straight. Approach 1 is better more stable in this regard.

6 Conclusion

6.1 Summary

We implemented path following using pure pursuit algorithm and obstacle avoidance using a simpler version of VFH. From the experimentation we found high sensitivity towards various parameters like linear velocity, lookahead distance in pure pursuit and the threshold in VFH.

6.2 Limitations and Future Work

Both VFH approaches are threshold sensitive and appropriate tuning is required.

Linear velocity is constant in our approach, appropriate velocity control must be added to gain advantage of long straight paths while still slowing down at turns and obstacles.

In future, we would like to apply the algorithm on a real mobile robot to face the real world issues. We would also implement the VFH+ or VFH* described in the papers [6] and [7].

References

[1] J. Borenstein and Y. Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[2] R. Craig Coulter. Implementation of the pure pursuit path tracking algorithm. 1992.

[3] MathWorks. Pure pursuit controller. URL: `https://in.mathworks.com/help/nav/ug/pure-pursuit-controller.html`.

[4] MathWorks. Vector field histogram. URL: `https://in.mathworks.com/help/nav/ug/pure-pursuit-controller.html`.

[5] Illah R. Nourbakhsh Roland Siegwart. *Introduction to Autonomous Mobile Robots*. Intelligent Robotics and Autonomous Agents. The MIT Press, 1st edition, 2004.

[6] I. Ulrich and J. Borenstein. Vfh+: reliable obstacle avoidance for fast mobile robots. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, volume 2, pages 1572–1577 vol.2, 1998.

[7] I. Ulrich and J. Borenstein. Vfh/sup */: local obstacle avoidance with look-ahead verification. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 3, pages 2505–2511 vol.3, 2000.

[8] xLAB for Safe Autonomous Systems. F1tenth l10 - pure pursuit, Feb 2022. URL: `https://youtu.be/x9s8J4ucgO0`.