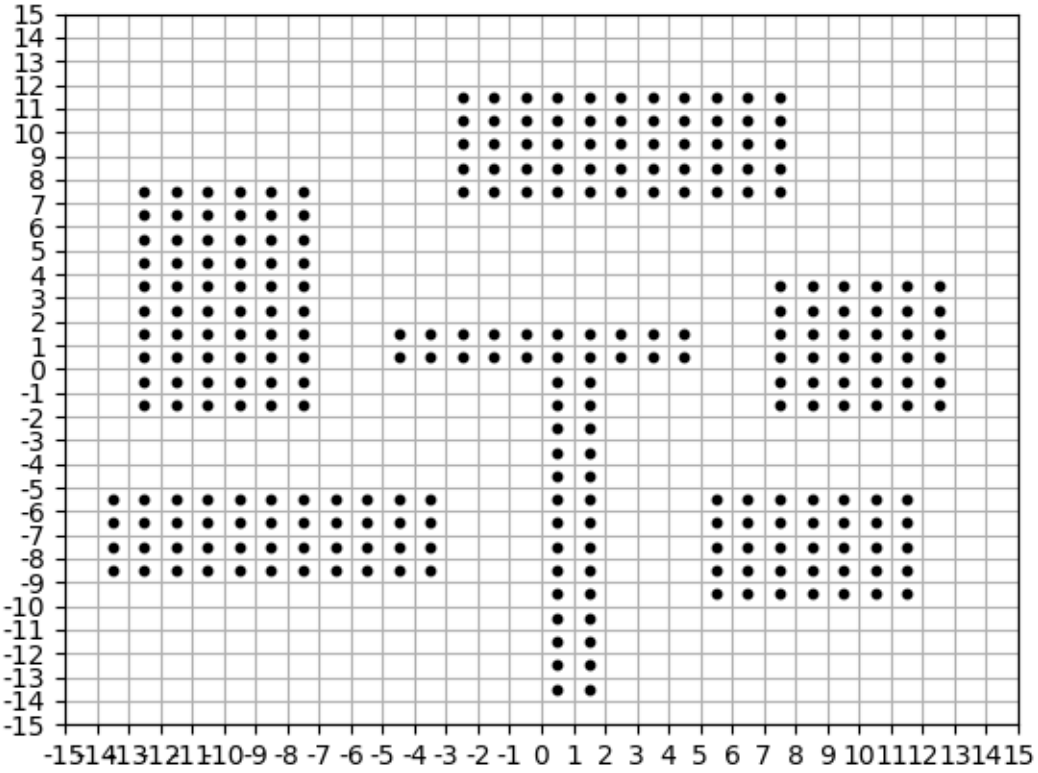
AE5335 Assignment 2

Submitted by: Purna Patel, Ritik Jain

Problem 1:

To Decompose the environment, we used square decomposition with the square size of 1 unit. In square decomposition we make a grid on the environment of size 1 unit and all the grids with obstacles in it are marked as obstacles and the rest are marked as free cells where the vehicle can traverse.

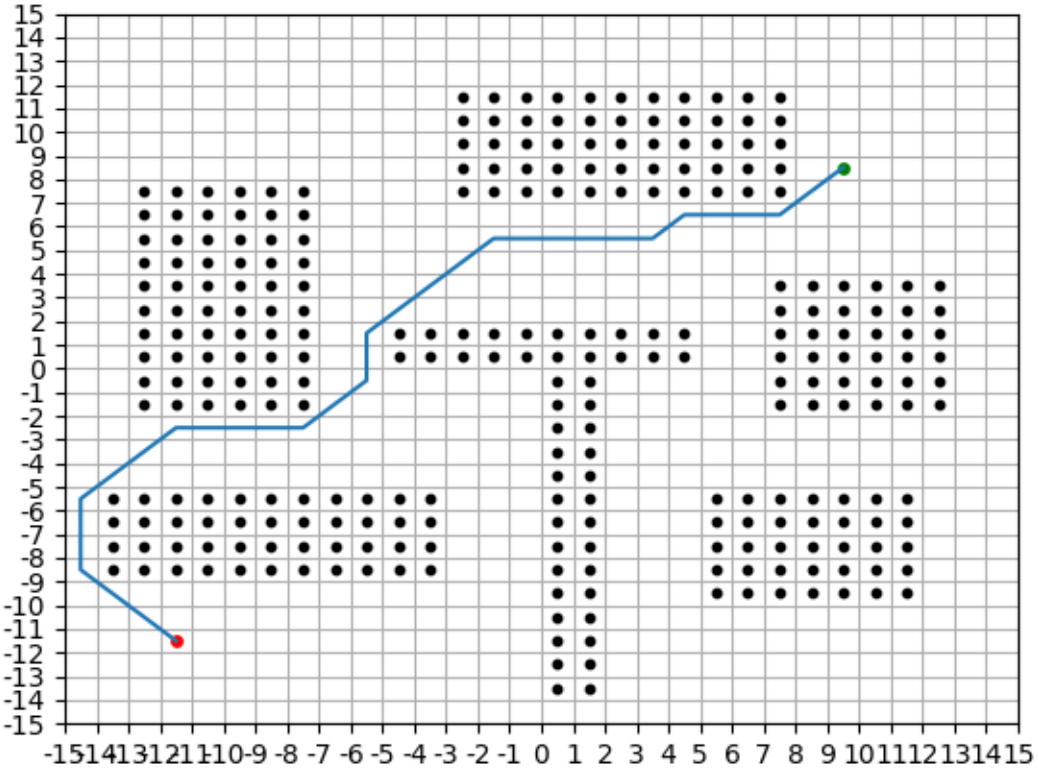
Figure of the cell decomposition plotted using Python



We used 8-connected neighbours therefore the edge is created in the free cell as follows

Vertices (i, j) has an edge with -> (i+1,j), (i,j+1), (i-1,j), (i,j-1), (i+1,j+1), (i-1,j+1), (i-1,j-1), (i+1,j-1) vertices

The output for the shortest path is as follow



The green point is the start point, red point is the goal, and the black points are the obstacles. The code for which is attached in the appendix.

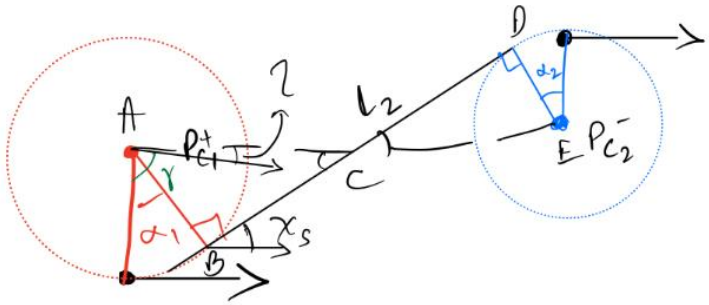
Problem 2:

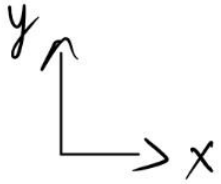
The dubins Connection function of MATLAB was used which gives the shortest path type and the distance between the start and the endPose provided the minTurning Radius

Start Pose $\Rightarrow P_0 = (0,0)$ $\chi_0 = 0$
End Pose $\Rightarrow P_1 = (1,1)$ $\chi_1 = 0$

$R_{min} = 0.15$ km

i) The shortest path is achived by C^+SC^- found using MATLAB





$$P_{C_1}^+ = P_0 + R_{min} \begin{bmatrix} \cos(\chi_0 + \pi/2) \\ \sin(\chi_0 + \pi/2) \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0.15 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.15 \end{bmatrix}$$

$$P_{c_2}^- = P_1 + R_{min}\begin{bmatrix}\cos(x_1 - \pi/2)\\ \sin(x_1 - \pi/2)\end{bmatrix}$$

$$= \begin{bmatrix}1\\1\end{bmatrix} + 0.15\begin{bmatrix}0\\-1\end{bmatrix} = \begin{bmatrix}1\\0.85\end{bmatrix}$$

In $\triangle ABC$ $\quad AC = \dfrac{\| P_{c_2}^- - P_{c_1}^+ \|}{2}$

$$= \sqrt{(1-0)^2 + (0.85-0.15)^2}\,/2$$

$$= \sqrt{1+0.7^2}\,/2$$

$$= \sqrt{1.49}\,/2$$

$$= 1.2207/2 = 0.6103$$

$$AC^2 = AB^2 + BC^2$$

$$BC = \sqrt{AC^2 - AB^2}$$

$$= \sqrt{0.6103^2 - 0.15^2}$$

$$= 0.5916$$

$BC = CD$ (The triangles are concurrent)

$$l_2 = BC + CD$$

$$= 1.1832$$

$$\gamma = \tan^{-1}\left(\frac{BC}{AB}\right) = 1.3225$$

$$\gamma = \tan^{-1}\left(\frac{P_{C_2}y - P_{C_1}y}{P_{C_2}x - P_{C_1}x}\right) = \tan^{-1}\left(\frac{0.85 - 0.15}{1 - 0}\right)$$

$$= 0.6107$$

$$(\gamma - \eta) + (\pi/2 + \chi_s) = \pi$$

$$\chi_s = \pi/2 + \eta - \gamma$$

$$= 0.859$$

$$\chi_0 + \alpha_1 = \chi_s \qquad \chi_s - \alpha_2 = \chi_1$$

$$\alpha_1 = 0.859 \qquad \alpha_2 = 0.859$$

Total Path length $= R_{min}\alpha_1 + l_2 + R_{min}\alpha_2$

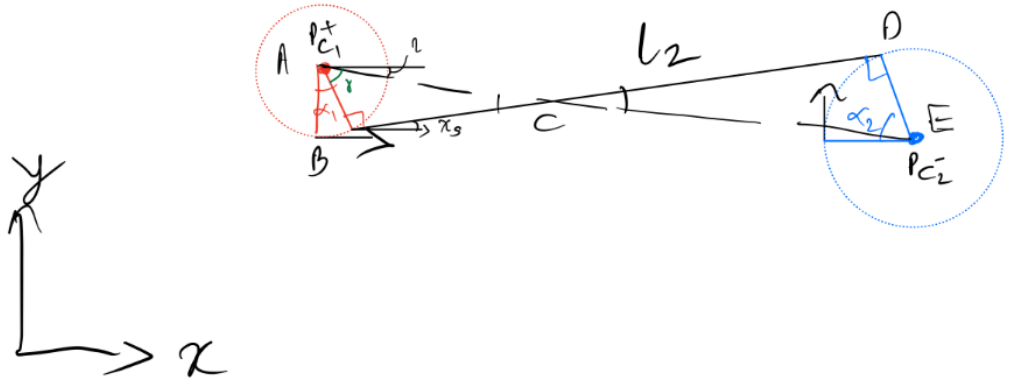
$= 0.15(0.859) + 1.1832 + 0.15(0.859)$

$= 1.4409$ km

ii) The shortest path is achived by C^+SC^- found using MATLAB

Start Pose $\Rightarrow P_0 = (1, 1)\ \chi_0 = 0$

End Pose $\Rightarrow P_1 = (1.25, 1)\ \chi_1 = \pi/2$



$$P_{C_1}^+ = P_0 + R_{min} \begin{bmatrix} \cos(\chi_0 + \pi/2) \\ \sin(\chi_0 + \pi/2) \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 0.15 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1.15 \end{bmatrix}$$

$$P_{C_2}^- = P_1 + R_{min} \begin{bmatrix} \cos(\chi_1 - \pi/2) \\ \sin(\chi_1 - \pi/2) \end{bmatrix}$$

$$= \begin{bmatrix} 1.25 \\ 1 \end{bmatrix} + 0.15 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.4 \\ 1 \end{bmatrix}$$

In $\triangle ABC$ $AC = \| P_{C_2} - P_{C_1} + \| / 2$

$= \sqrt{(1.4-1)^2 + (1 - 1.15)^2} / 2$

$= \sqrt{0.4^2 + 0.15^2} / 2$

$= \sqrt{0.1825} / 2$

$= 0.4272 / 2 = 0.2136$

$AC^2 = AB^2 + BC^2$

$BC = \sqrt{AC^2 - AB^2}$

$= \sqrt{0.2136^2 - 0.15^2}$

$= 0.1521$

$BC = CD$ (The triangles are concurrent)

$l_2 = BC + CD$

$= 0.3041$

$\gamma = \tan^{-1}\left(\frac{BC}{AB}\right) = 0.7923$

$$\gamma = \tan^{-1}\left(\frac{P_{c_2}^{-}y - P_{c_1}^{+}y}{P_{c_2}^{-}x - P_{c_1}^{+}x}\right) = \tan^{-1}\left(\frac{1-1.15}{1.4-1}\right)$$

$$= -0.3588 \quad = \quad 5.9244$$

$$(\gamma - \gamma) + (\pi/2 + \chi_s) = \pi$$

$$\chi_s = \pi/2 + \gamma - \gamma$$

$$= 6.7029 \quad = 0.4197$$

$$\chi_0 + \alpha_1 = \chi_s \qquad \chi_s - \alpha_2 = \chi_1$$

$$\alpha_1 = 0.4197 \qquad \alpha_2 = 0.4197 - \pi/2$$

$$= -1.1511 = 5.1321$$

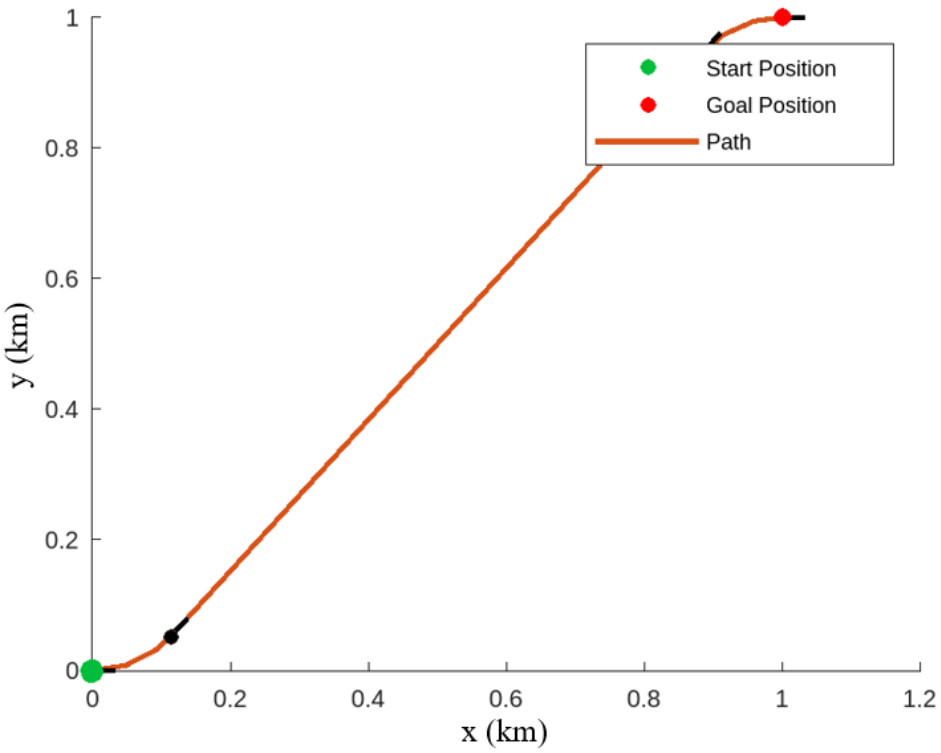
Total Path length $= R_{min}\alpha_1 + l_2 + R_{min}\alpha_2$

$$= 0.15(0.4197) + 0.3041 + 0.15(5.1321)$$

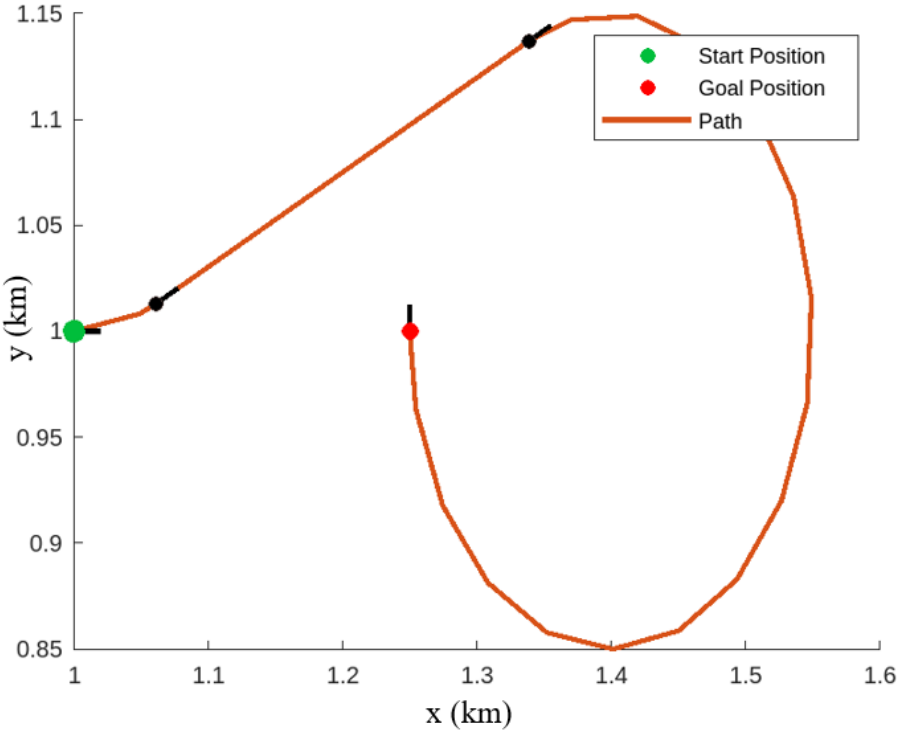
$$= \underline{1.1369 \text{ Km}}$$

Output plot:

1)



2)



Matlab Script attached in the appendix at the end of the report.

Problem 3:

Given waypoints and directions.

Waypoint 1: p1 = (0,0) and $\chi 1 = 10$.

Waypoint 2: p2 = (10,−5) and $\chi 2 = 50^{\circ}$.

Waypoint 3: p3 = (30,5) and $\chi 3 = 120^{\circ}$.

To make the problem easier, the trajectory is divided into 2 parts.

2 trajectories are generated between points 1 – 2 and 2 – 3 and stitched together.

The optimization is run twice to find two trajectories and then the trajectories are stitched together to form a single smooth path.

Problem formulation:

1) **Order of the trajectory:**
   4th order trajectory is chosen for optimising the path.
   General equation:

$$a_{x0} + a_{x1}t + a_{x2}t^2 + a_{x3}t^3 + a_{x4}t^4 = p_x$$
$$a_{y0} + a_{y1}t + a_{y2}t^2 + a_{y3}t^3 + a_{y4}t^4 = p_y$$

2) **Decision variables:**
   Considering t1 = 1 for making the formulation easier, there are total 10 decision variables, 5 for each dimension.

$$x = \left[a_{x0}, a_{x1}, a_{x2}, a_{x3}, a_{x4}, a_{y1}, a_{y1}, a_{y2}, a_{y3}, a_{y4}\right]^T$$

3) **Cost function:**
   For the problem we need to minimize the total arc length of the trajectory.
   Instead of minimizing the actual arc length, the squared arc length is to be minimized which provides similar result, but the squared root formulation is neglected which makes computation easier.

$$f(x) = \int_0^1 (a_{x0} + a_{x1}t + a_{x2}t^2 + a_{x3}t^3 + a_{x4}t^4)^2 + (a_{y0} + a_{y1}t + a_{y2}t^2 + a_{y3}t^3 + a_{y4}t^4)^2 \; dt$$

   Converting the equation in matrix form.

$$f(x) = \int_0^1 x^T \begin{bmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \\ 4t^3 \\ \mathbf{0} \end{bmatrix} [0 \quad 1 \quad 2t \quad 3t^2 \quad 4t^3 \quad \mathbf{0}] x$$

$$+ x^T \begin{bmatrix} \mathbf{0} \\ 0 \\ 1 \\ 2t \\ 3t^2 \\ 4t^3 \end{bmatrix} [\mathbf{0} \quad 0 \quad 1 \quad 2t \quad 3t^2 \quad 4t^3] x \, dt$$

$$f(x) = \int_0^1 x^T H(t) x \, dt$$

Where,

$$H(x) = \begin{bmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \\ 4t^3 \\ \mathbf{0} \end{bmatrix} [0 \quad 1 \quad 2t \quad 3t^2 \quad 4t^3 \quad \mathbf{0}] + \begin{bmatrix} \mathbf{0} \\ 0 \\ 1 \\ 2t \\ 3t^2 \\ 4t^3 \end{bmatrix} [\mathbf{0} \quad 0 \quad 1 \quad 2t \quad 3t^2 \quad 4t^3]$$

Applying trapezoidal rule to the integral.

$$f(x) = x^T \widetilde{H} x$$

Where,

$$\widetilde{H} = \delta t \left(\tfrac{1}{2} H(0) + \tfrac{1}{2} H(1) + H(\delta t) + H(2\delta t) + \cdots + H\big((M-1)\delta t\big) \right)$$

4) Equality constrains:

Equality constrains for trajectory between 2 points.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & tan\chi_0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & tan\chi_1 & 2tan\chi_2 & 3tan\chi_3 & 4tan\chi_4 & 0 & -1 & -2 & -3 & -4 \end{bmatrix} x = \begin{bmatrix} p_{x0} \\ p_{y0} \\ p_{x1} \\ p_{y1} \\ 0 \\ 0 \end{bmatrix}$$

5) Inequality constrains:

Normally there is only one inequality constrain, i.e. t1 >=0. But as we are considering t1 = 1, there is no need of this constrain.

When the optimization problem was run with inequality constrain g(x) <= 0, the algorithm seems to guess incorrect direction at the setpoints.
This is due to the tangent terms in the equality constrains.
To avoid this, following directional inequality constrains were implemented for the end points.

We know that,

$$\tan\chi_0 = \frac{a_{y1}}{a_{x1}} = \frac{\sin\chi_0}{\cos\chi_0} \qquad \text{and} \qquad \tan\chi_1 = \frac{a_{y1}+2a_{y2}+3a_{y3}+4a_{y4}}{a_{x1}+2a_{x2}+3a_{x3}+4a_{x4}} = \frac{\sin\chi_1}{\cos\chi_1}$$

$$\sin\chi_0 = \frac{a_{y1}}{\sqrt{a_{y1}^2+a_{x1}^2}} \qquad \cos\chi_0 = \frac{a_{x1}}{\sqrt{a_{y1}^2+a_{x1}^2}}$$

$$\sin\chi_1 = \frac{a_{y1}+2a_{y2}+3a_{y3}+4a_{y4}}{\sqrt{(a_{y1}+2a_{y2}+3a_{y3}+4a_{y4})^2+(a_{x1}+2a_{x2}+3a_{x3}+4a_{x4})^2}}$$

$$\cos\chi_1 = \frac{a_{x1}+2a_{x2}+3a_{x3}+4a_{x4}}{\sqrt{(a_{y1}+2a_{y2}+3a_{y3}+4a_{y4})^2+(a_{x1}+2a_{x2}+3a_{x3}+4a_{x4})^2}}$$

In the above equations, the denominators will always be positive.
Thus, the sign of the LHS terms matches the numerator.
Therefore, we can say that.

$$sign(\sin\chi_0)a_{y1} \geq 0 \qquad sign(\cos\chi_0)a_{x1} \geq 0$$
$$sign(\sin\chi_1)(a_{y1}+2a_{y2}+3a_{y3}+4a_{y4}) \geq 0$$
$$sign(\cos\chi_1)(a_{x1}+2a_{x2}+3a_{x3}+4a_{x4}) \geq 0$$

If we evaluate closely, the above equations constrain the first derivatives of the trajectory at the end points.

6) Calculating Initial x values:

To guess the initial values of x, a simple 3rd trajectory f(x) = y, passing through 2 points at given direction is calculated as follows.

$$A = \begin{bmatrix} 1 & p_{x0} & p_{x0}^2 & p_{x0}^3 \\ 1 & p_{x1} & p_{x1}^2 & p_{x2}^3 \\ 0 & 1 & 2p_{x0} & 3p_{x0}^2 \\ 0 & 1 & 2p_{x1} & 3p_{x1}^2 \end{bmatrix} \qquad b = \begin{bmatrix} p_{y0} \\ p_{y1} \\ \tan\chi_0 \\ \tan\chi_1 \end{bmatrix}$$

$$a = A^{-1}b$$

The above calculation of coefficients will give us following equation.

$$p_y(p_x) = a_0 + a_1 p_x + a_2 p_x^2 + a_3 p_x^3$$

Now let us consider p_x changes constantly with t and $p_x(0) = p_{x0}$ and $p_x(1) = p_{x1}$. Therefore,

$$p_x(t) = p_{xo} + (p_{x1} - p_{x0})t$$

Substituting $p_x(t)$ in the calculated 2D curve equation, we get

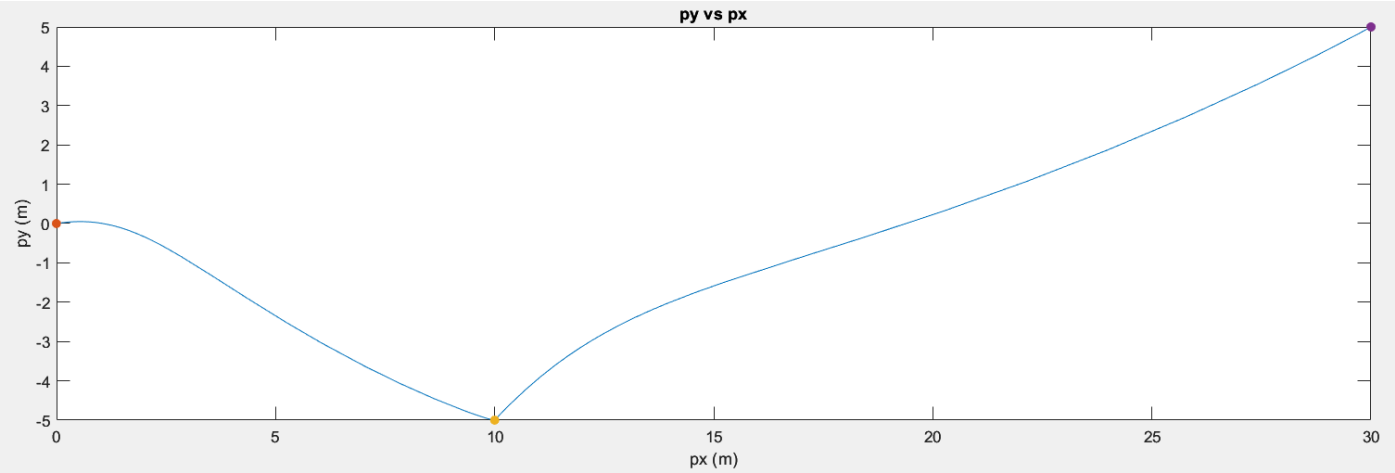
$$p_y(t) = a_0 + a_1(p_{xo} + (p_{x1} - p_{x0})t) + a_2(p_{xo} + (p_{x1} - p_{x0})t)^2 + a_3(p_{xo} + (p_{x1} - p_{x0})t)^3$$

Rearranging the above equation, we get initial coefficients for trajectory $p_y(t)$. These calculated coefficients are used as initial guesses for the optimization problem.

Results:

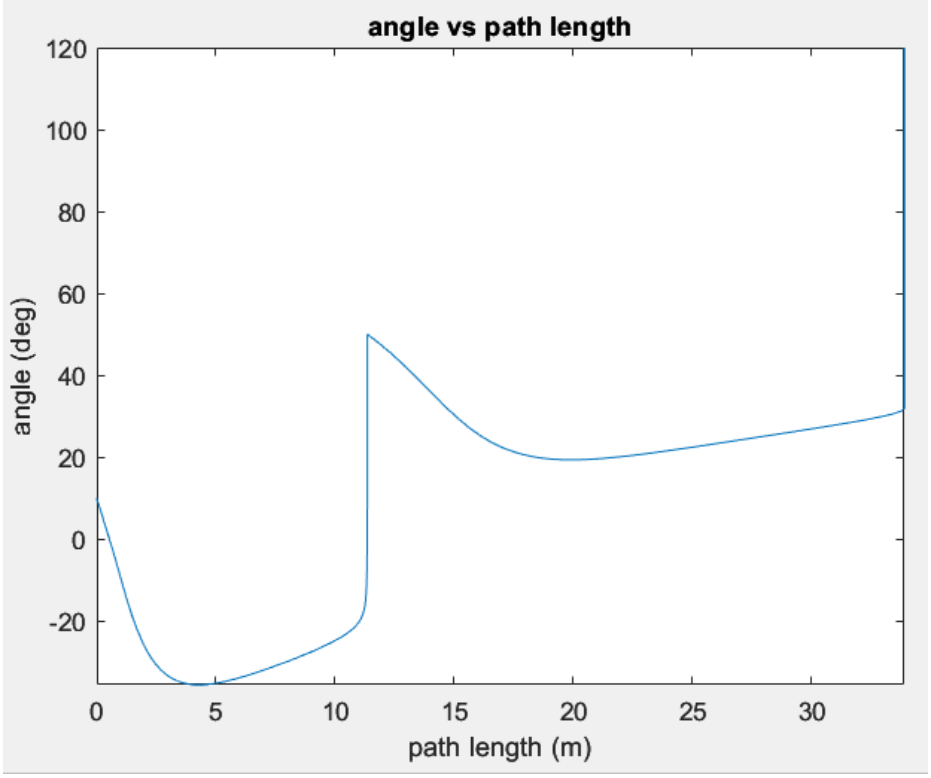
After optimizing the trajectory based on the above stated formulation, following results were obtained.

Total Path length: 33.9077



The above plot shows the curve in in x-y plane. From the first look, it doesn't seem to follow the directional constrain.

But when we plot path angle vs path-length graph we can see an interesting outcome.

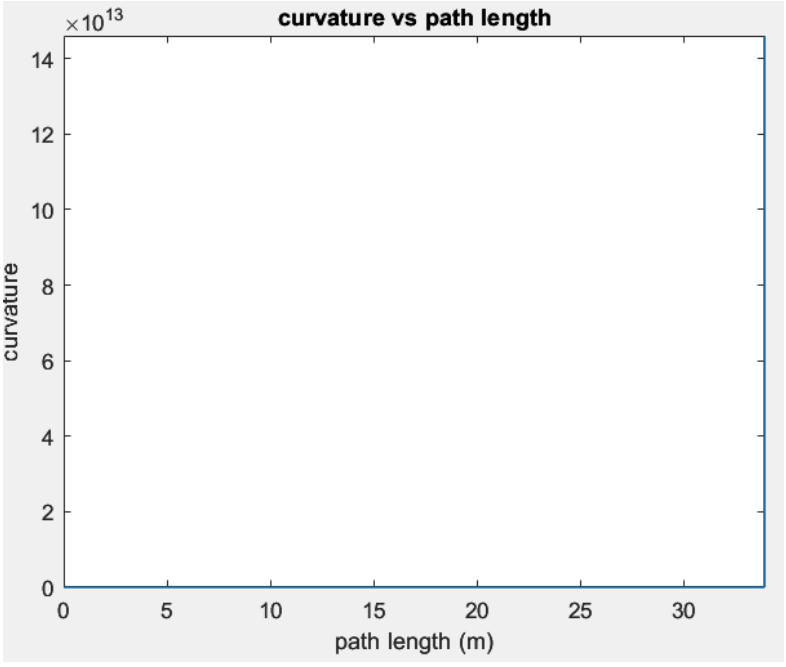


The algorithm seems to satisfy the constrain while minimizing length and velocity.

The algorithm minimizes the velocity to zero at the end points.

The curve obtained achieves the desired angle but the change is almost instantaneous.

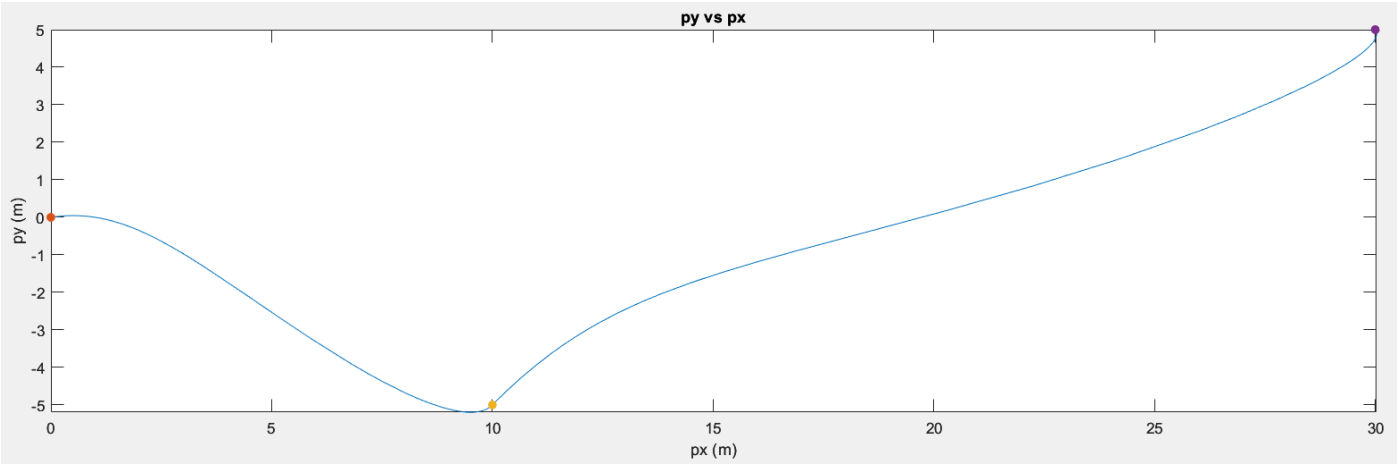
Due to this the path curvature at those points shoots to infinity as shown in the plot below.



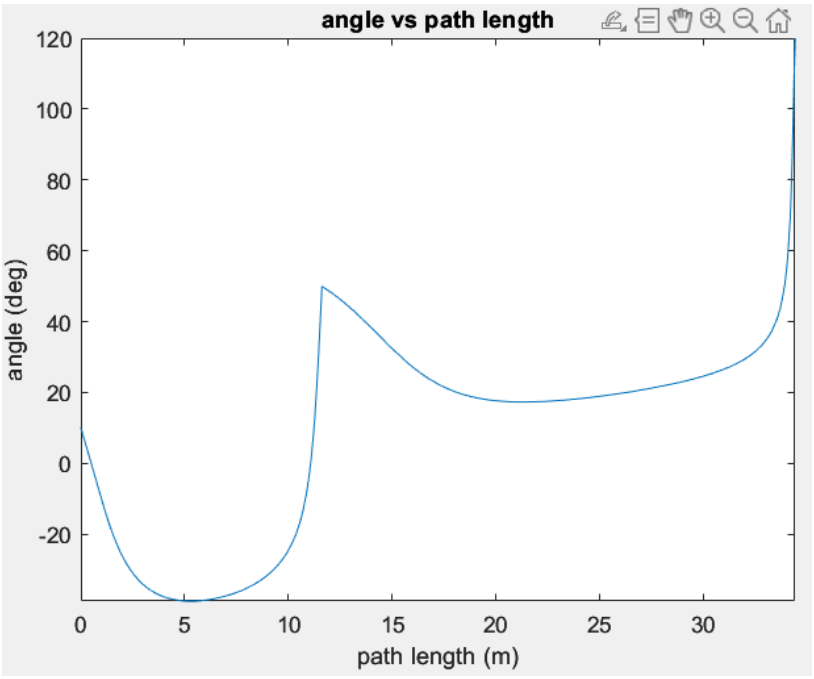
To avoid this, minimum velocity constrains are added.

Following were the results obtained after adding minimum velocity constrains.

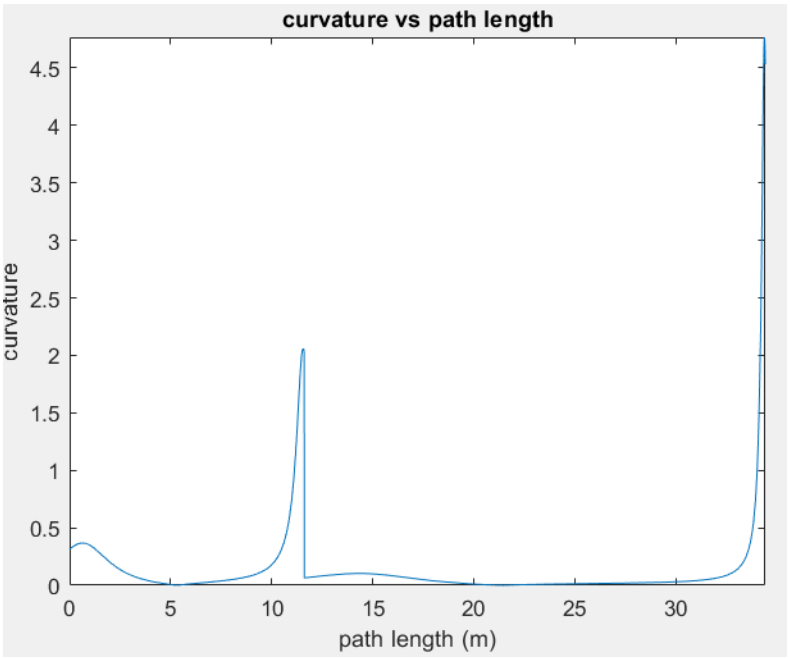
Path length: 34.4366 m



It can be observed from the plot that now the change in direction is not instantaneous.

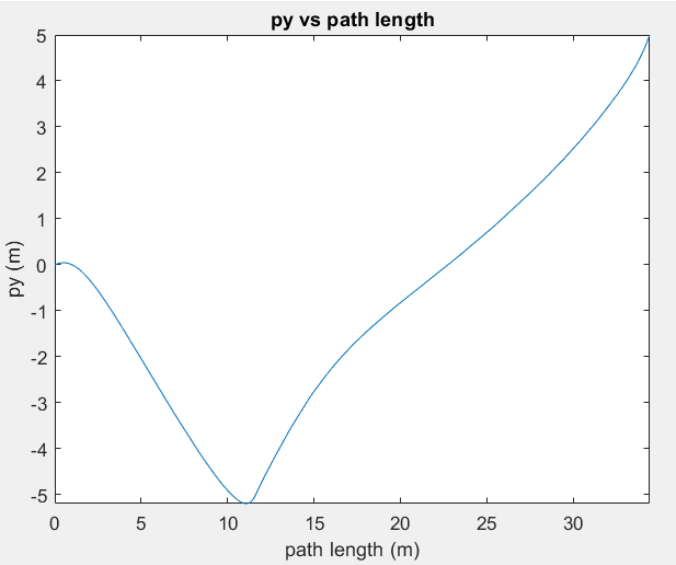


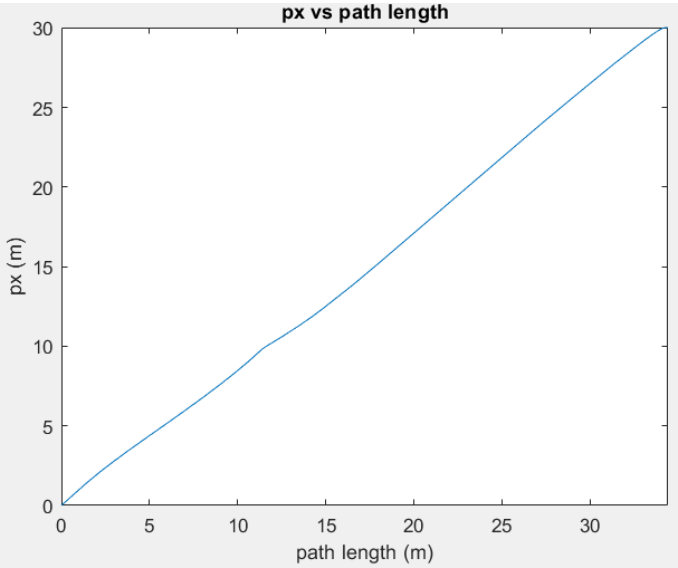
The above curve shows angle vs path length plot. Still there is sharp change in the angles at the end of part 1 and part 2 of the trajectory. But it is not instantaneous.



The above plot shows the curvature vs path length plot. Here as well, there is still a sharp rise in the curvature at the end points of two trajectories but it is better than the trajectory without velocity constrain.

The following curves shows the py vs path length and px vs path length plots.





Matlab Script for problem 3 and problem 4 is attached in the appendix at the end of the report.

Problem 4:

The formulation of the problem is same as problem 3.

The only difference is the addition of acceleration in the minimization function to minimize the curvature throughout the curve.

Updating the cost for minimizing acceleration:

Adding acceleration term in the cost function.

$$f(x) = \int_0^1 \begin{pmatrix} (a_{x0} + a_{x1}t + a_{x2}t^2 + a_{x3}t^3 + a_{x4}t^4)^2 + (a_{y0} + a_{y1}t + a_{y2}t^2 + a_{y3}t^3 + a_{y4}t^4)^2 \\ +\mu\left((a_{x1} + 2a_{x2}t + 3a_{x3}t^2 + 4a_{x4}t^3)^2 + \left(a_{y1} + 2a_{y2}t + 3a_{y3}t^2 + 4a_{y4}t^3\right)^2\right) \end{pmatrix} dt$$

Converting the equation in matrix form.

$$f(x) = \int_0^1 x^T H(t) x \, dt$$

$$H(t) = H_1(t) + \mu H_2(t)$$

Where,

$$H_1(x) = \begin{bmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \\ 4t^3 \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} 0 & 1 & 2t & 3t^2 & 4t^3 & \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ 0 \\ 1 \\ 2t \\ 3t^2 \\ 4t^3 \end{bmatrix} \begin{bmatrix} \mathbf{0} & 0 & 1 & 2t & 3t^2 & 4t^3 \end{bmatrix}$$

$$H_2(x) = \begin{bmatrix} 0 \\ 0 \\ 2 \\ 6t \\ 12t^2 \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} 0 & 0 & 2 & 6t & 12t^2 & \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ 0 \\ 0 \\ 2 \\ 6t \\ 12t^2 \end{bmatrix} \begin{bmatrix} \mathbf{0} & 0 & 0 & 2 & 6t & 12t^2 \end{bmatrix}$$

Applying trapezoidal rule to the integral.

$$f(x) = x^T \tilde{H} x$$

Where,

$$\tilde{H} = \delta t \left(\tfrac{1}{2}H(0) + \tfrac{1}{2}H(1) + H(\delta t) + H(2\delta t) + \cdots + H\big((M-1)\delta t\big)\right)$$

All the other formulations are same as the previous problem.

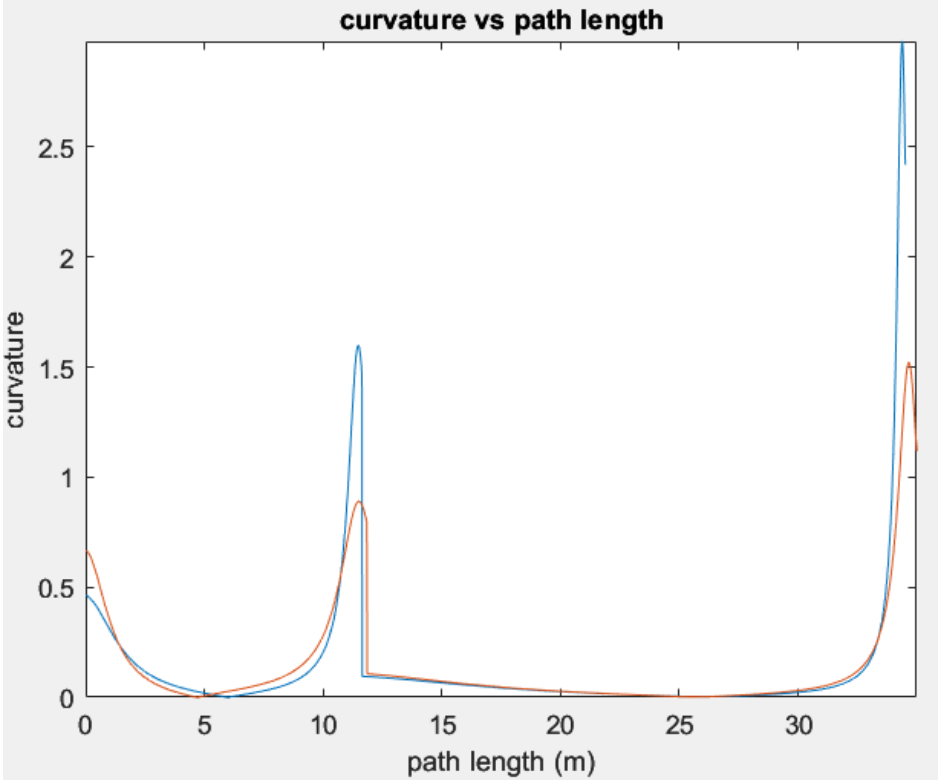
Results:

After optimizing the trajectory based on the above stated formulation, following results were obtained.

Path length ($\mu = 0.1$) = 34.5118 m

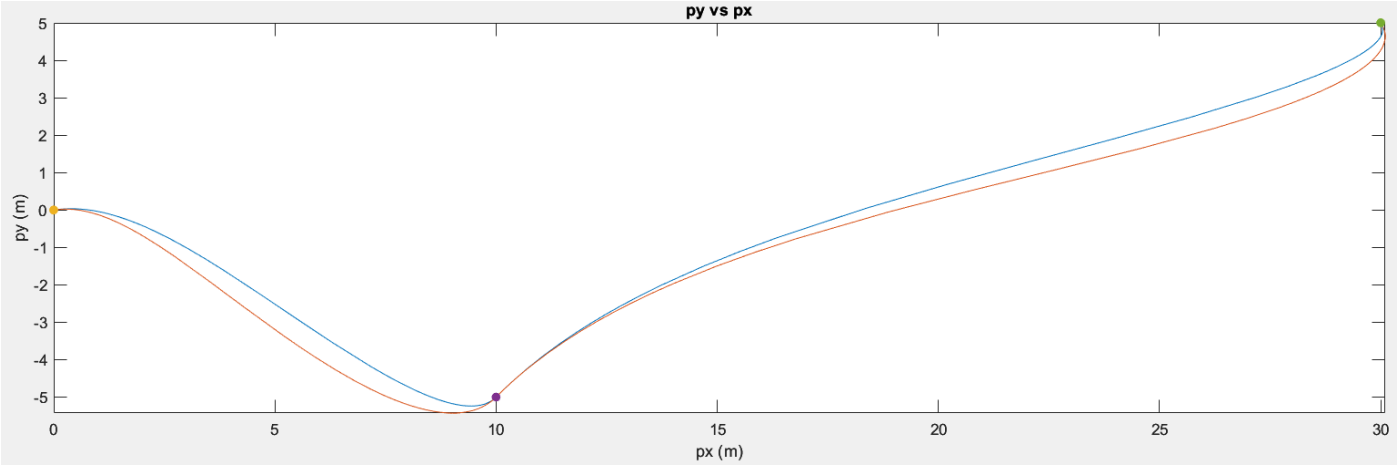
Path length ($\mu = 1$) = 34.9877 m

Following plot shows the difference in curvature between $\mu = 0.1$ and $\mu = 1$.



The blue curve shows the curvature of trajectory with $\mu = 0.1$ and the red curve shows the curvature of trajectory with $\mu = 1$.

Following curve shows the difference between the trajectories of $\mu = 0.1$ and $\mu = 1$.



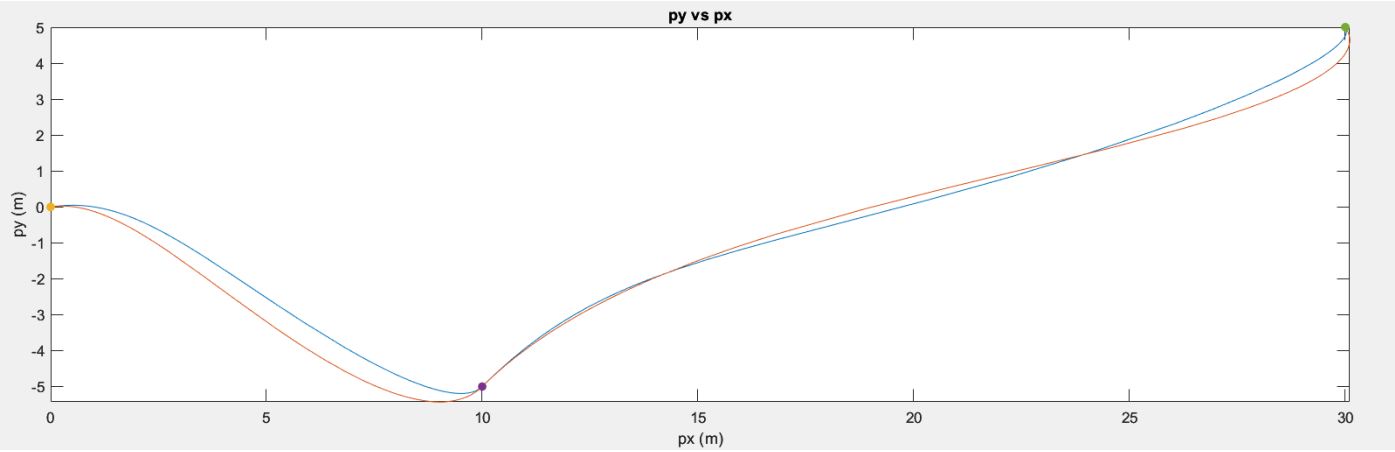
In the above plot, blue curve shows the trajectory with $\mu = 0.1$ and the red curve shows the trajectory with $\mu = 1$.

It can observe in the above comparison that as μ is increased, the curvature throughout the trajectory decreases and the path length increases.

Higher μ guarantees smoother curve with higher path length cost.

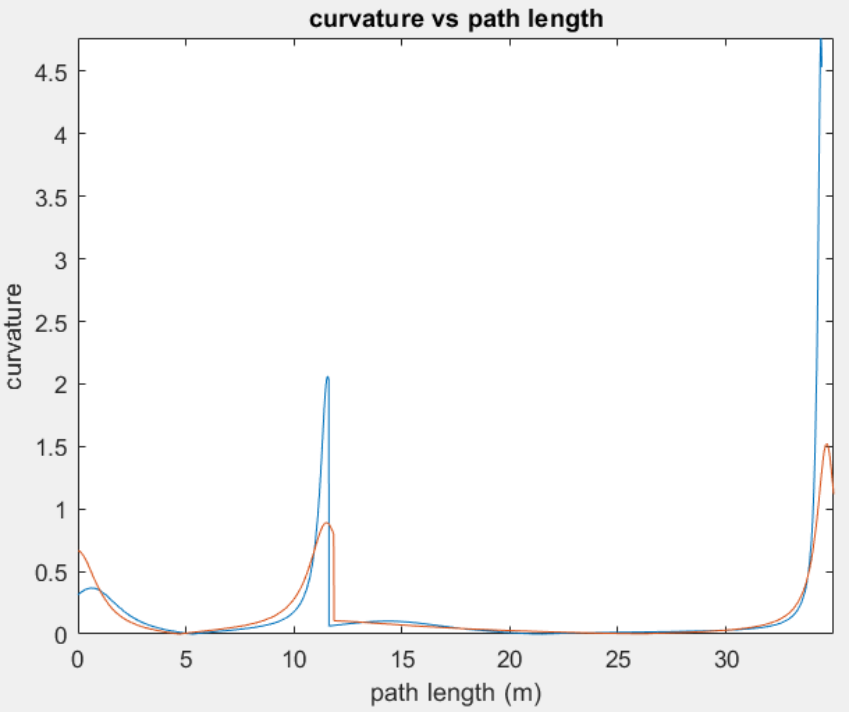
But the smoothness increases to a threshold, after that increasing μ make very negligible difference in the trajectory.

Following plots compares results of trajectory from problem 3 and the trajectory with $\mu = 1$.



In the above plot the blue curve represents the trajectory from problem 3 and the red curve represents the trajectory with $\mu = 1$.

It can be observed that the red curve is smoother than the blue curve.



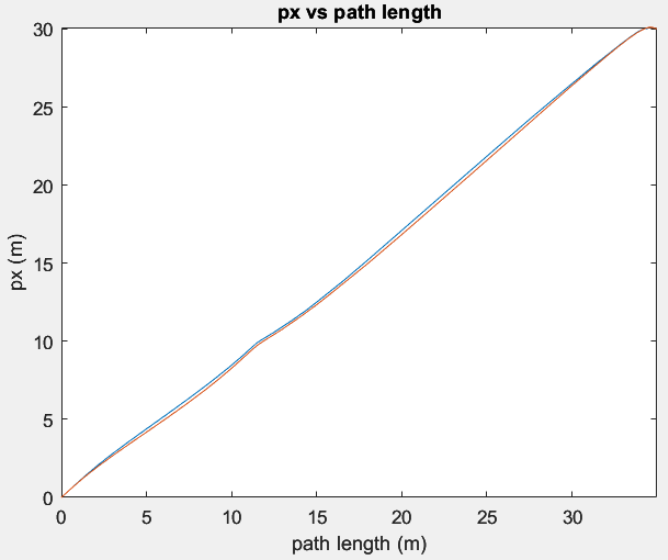
The above plot compares the curvature vs path length plots of both the trajectories.

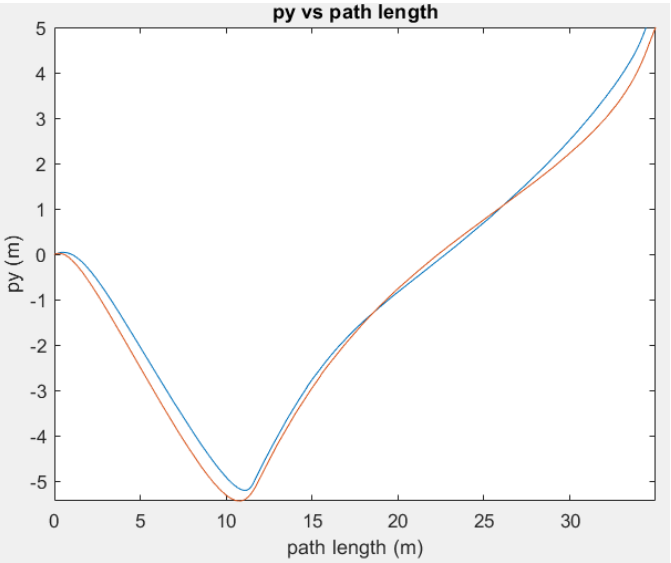
Blue curve represents the curvature of trajectory from problem 3 and the red curve represents the curvature of trajectory with $\mu = 1$.

It can be observed there is significant decrease in the value of curvatures at the end points of the trajectory.

The following plots compares the px vs path length and py vs path length plots.

The Blue curve represents the trajectory from problem 3 and the red represents trajectory with $\mu = 1$.





Matlab Script for problem 3 and problem 4 is attached in the appendix at the end of the report.

Problem 5:

This paper divides the problem of finding the shortest path in two hierarchical level, where the high level satisfies the logical task specification, and the low level satisfies the vehicles control law. The hierarchical approach simplifies a much complex problem which involves solving two different subproblem together. The higher level also called the geometric path planning level is mainly concerned with finding an obstacle free path. The geometric planner is unaware of the vehicle's kinematic and dynamic constraints. The lower level also called the trajectory planning level is concerned with smoothens the geometric path and imposes a suitable time parameterization and a tracking controller which generates the control inputs to the follow the trajectory. The paper also proposes a novel mode of interaction between the two level of planners.

The idea of keeping the geometric planner independent form the vehicular dynamics was driven from other methods of motion planning involving sampling-based algorithms which requires very efficient low level for collision detection and trajectory planning algorithm, planners that use cell decomposition are coupled with feedback control law method to ensure feasible transition between cells. For the Geometric path planner, the paper proposes a history-based transition cost as algorithms based on cell decomposition provides no guarantee that the resultant path can be traversed feasibly. In a normal cell decomposition, there is no way to penalize cell-dependent properties associated between two adjacent cells. The problem with maintaining a history is that the search time increases exponentially, since the vertices and edges increases grows exponentially with history. The paper also proposes a way to reduce the search time significantly by introducing a parameter that allows a trade-off between the path optimality and search time.

To test the geometric algorithm a graph with uniform cell decomposition of 4-connectivity was used, it was tried with 30 times where the initial and final position was randomly selected. The time complexity was three order magnitude faster in average case and four magnitudes faster in best case scenario compared to approaches that first construct the lifted graph and then execute the search. The drawback was it used K times the memory required where K is the valency of the graph. To improve this a fixed number of histories was introduced that deletes set that are not in the first L histories.

For generating the trajectories, it assumes planar position and orientation and the method to generate these trajectories is called tile planner. The tile planner generates the control input that follows two constraints, first the position should always stick on the path and second the position leaves the tile for a finite time. The yaw is calculated using the velocity which acts as a constraint and has an upper and lower bound. If the tile plan is used with dubins car an upper bound for the local curvature may then be computed which is based on the specific vehicle model. The motion planning framework consists of a geometric path planner that repeatedly invokes tile plan determine H-costs of histories.

The results were compared with RRT algorithms, and the tile plan was used based on model predictive control. Two different environments were used with 30 trials for three fixed time interval that integrate the vehicle model with the RRT based planner. The proposed planner performed significantly better that the RRT based planners in the basis of path cost. Also, the

number of states explored in RRT based planner were more, but the time required to explore each state in RRT was lesser than the proposed planner. Thus, a direct comparison between the planner was inconclusive. The paper proposes the benefits of tile planner is that it can easily incorporate complex vehicle dynamics, but in the paper all the tests are done assuming the point-mass vehicle model.

Problem 6:

The article “Interactive sensing and planning for a quadrotor vehicle in partially known environments” proposes a viable solution to the path planning problem of quadcopters to be operated in partially known environments and has limited onboard environment sensing capabilities. There are many research that focuses on path planning of quadcopter but they all either assume that the accurate 3D map of the environment is already available or have onboard sensing capability to accurately sense the obstacles and plan accordingly. This research article represents an interesting blend of the problems where the 2D counter of the environment is pre known and the onboard sensing capability is limited to one depth sensor that is fixed to the frame of the drone. This problem mimics a situation where a floor plan of the building is pre known but its 3D map is not.

To approach this problem, the article suggests an algorithm which is a blend of path planning and simultaneous sensing of the environment. The algorithm pre plans the initial trajectory based on the available 2D map information and scans the area while on the trajectory and if a 3D short cut is found, it updates the trajectory. The paper also discusses an optimal way to plan the initial 2D path such that the probability of finding a 3D short cut increases.

In the article, it is assumed that the onboard depth sensor has limited field of view and range. Thus, when on trajectory, the quadcopter has limited capability of scanning the environment (i.e., at a point in trajectory, the quadcopter can only scan the environment in the range and field of view of the sensor). So, for the better result, it becomes important that the sensor is pointing in the right direction where there is maximum possibility of finding a 3D shortcut. This is accomplished by running a cost minimization algorithm on each vertex on the map to find the best pointing angle for the sensor. Thus, while following the trajectory, on any vertex, the quad copter will have an optimal pointing direction where it has maximum possibility of finding a 3D short cut. As this optimization is based on initial 2D map, it can be precomputed before starting the trajectory.

After pre-computing the optimal 2D path (this path refers to the path which has maximum possibility of finding a 3D shortcut) and optimal heading in the environment, the drone generates a minimum snap trajectory and starts following it. On the trajectory the drone constantly searches for a 3D shortcut. If found, an optimal path repair policy checks if the possible 3D-shortcut is optimal than the available path. This policy along with the path cost,

also takes ‘possible future 3D shortcuts on the path’ into account before it decides to follow the path. If the shortcut is optimal, it generates the trajectory and takes the shortcut.

In the conclusion the article discusses the simulation results where it shows the capability of the algorithm for finding an optimal path. It is also mentioned that the capability of the algorithm to find an optimal path is limited by the range and field of view of the sensor onboard.

Observed limitation of the algorithm:

In the formulation of the problem, it is assumed that the quadcopter always has accurate location data which is mostly not the case while navigating in indoor GPS denied environments. To navigate in such environments, the quadcopter needs the same 3D depth sensor for localization purpose, which will largely limit the scope of pointing the sensor at specific direction to find a 3D shortcut to repair the current path.

Appendix

Problem 1:

```
import numpy as np #using numpy arrays
import matplotlib.pyplot as plt #representing the grid in graph
import random
import math




def is_valid(i,j,grid,visited):  #to check if the neighbour is not out of bound, a
obstacle or a visited node
    if i<0 or j<0 or i>=rows or j>=cols:
        return False
    elif grid[i][j]==1:
        return False
    elif visited[i][j]==1:
        return False
    return True

def priority(queue): #find the path with shortest distance that will be selected from the
priority queue
    min = 99999999
```

```
    index = 0
    for check in range(len(queue)):
        _,value,_ = queue[check]
        if value<min:
            min = value
            index = check #index of the shortest path
    return index

def dist(point1,point2):
    dist = math.sqrt(math.pow((point1[0]-point2[0]),2)+math.pow((point1[1]-point2[1]),2))
# calculate distance for cost function
    return dist

def Dijkstra(grid):
    queue = []
    path =[start] #keep check for the final path
    value = 0
    visited = np.zeros((rows,cols)) # to check visited nodes
    queue.append((start,value,path)) #this creates a tree structure to trace the final
path
    neighbours = [[1,0],[-1,0],[0,1],[0,-1],[1,1],[-1,1],[1,-1],[-1,-1]] #considering 8
connected neighbour
    iteration =0
    while not(len(queue)==0):
        iteration+=1
        index = priority(queue)
        (shortest,value,path) = queue[index] #select the node with lowest distance
        queue.pop(index)
        if visited[shortest[0]][shortest[1]]==0:
            visited[shortest[0]][shortest[1]]=1
            #print(shortest)
            if shortest == end:
                print("Reached")
                return path,iteration,True # to indicate a valid path exists
            for m,n in neighbours:
                if (is_valid(shortest[0]+m,shortest[1]+n,grid,visited)):  #to check if the
neighbour is not out of bound, a obstacle or a visited node
                    queue.append(([shortest[0]+m,shortest[1]+n],value+dist([shortest[0],sh
ortest[1]],[shortest[0]+m,shortest[1]+n]),path + [[shortest[0]+m,shortest[1]+n]]))  #add a
valid neighbour to queue
    return path,iteration,False




rows = 31
cols = 31
grid_size = rows*cols
start = [23,24] #strating position of planner
end = [3,3] #goal position of planner
grid = np.zeros((rows,cols)) #creating an obstacle free grid
for i in range (6,10):
    for j in range (1,12):
```

```
        grid[i][j] = 1
for i in range (13,23):
    for j in range (2,8):
        grid[i][j] = 1
for i in range (15,17):
    for j in range (10,20):
        grid[i][j] = 1
for i in range (1,15):
    for j in range (15,17):
        grid[i][j] = 1
for i in range (22,27):
    for j in range (12,23):
        grid[i][j] = 1
for i in range (13,19):
    for j in range (22,28):
        grid[i][j] = 1
for i in range (5,10):
    for j in range (20,27):
        grid[i][j] = 1



path,itr,check = Dijkstra(grid)
plt.figure(1) #display windows of plot
fig = plt.gcf() # represnt the final grid
fig.canvas.manager.set_window_title('Dijkstra')
ax = fig.gca()
ax.set_xticks(np.arange(0, rows+1, 1))
ax.set_yticks(np.arange(0, cols+1, 1))
plt.scatter(start[1]+0.5,start[0]+0.5,c='green',s=15)
plt.scatter(end[1]+0.5,end[0]+0.5,c='red',s=15)
for x in range(rows):
    for y in range(cols):
        if grid[x][y] == 1:
            plt.scatter(y+0.5,x+0.5,c='black',s=10) #plot obstacles

plt.xlim([0,rows-1])#set graph from 0 to 30 for both x and y
plt.ylim([0,cols-1])
x = np.array([])
y = np.array([])
for i in range(len(path)):
    x = np.append(x,path[i][0]+0.5)
    y = np.append(y,path[i][1]+0.5)

plt.plot(y,x) #plot the final path
plt.grid()
plt.xticks(np.arange(0,31),np.arange(-15,16))
plt.yticks(np.arange(0,31),np.arange(-15,16))
plt.show()
```

Problem 2:

```
clear variables; close all; clc;
```

```
dubConnObj = dubinsConnection("MinTurningRadius",0.15);
startPose = [0,0,0];
goalPose = [1,1,0];
% startPose = [1,1,0];
% goalPose = [1.25,1,pi/2];
[pathSegObj, pathCosts] = connect(dubConnObj,startPose,goalPose);
show(pathSegObj{1})
exportgraphics(gcf, 'dubinsPath.png')
pathSegObj{1}.MotionTypes
disp(pathCosts);
```

Problem 3 and 4:


```
clear variables; close all; clc
```

Problem Data

```
p1 = [0 0]';
p2 = [10 -5]';
p3 = [30, 5]';

X1 = 10*pi/180;
X2 = 50*pi/180;
X3 = 120*pi/180;


p = [p1, p2, p3];
X = [X1, X2, X3];

size_p = size(p);
```

Initial Guess

Fitting a 3rd degree polynomil curve between the 2 points and stiching it. Varying x constantly with t and calculating the initial guesses for the coefficitnes.

```
x_init = [];
for i = 1:(size_p(2)-1)

    A_init = [1 p(1,i) p(1,i)^2 p(1,i)^3;
              1 p(1,i+1) p(1,i+1)^2 p(1,i+1)^3;
              0 1 2*p(1,i) 3*p(1,i)^2;
              0 1 2*p(1,i+1) 3*p(1,i+1)^2];
    b_init = [p(2,i) p(2,i+1) tan(X(i)) tan(X(i+1))]';

    a = pinv(A_init)*b_init;
    ax0 = p(1,i);
    ax1 = p(1,i+1) - p(1,i);
    ay0 = a(1) + a(2)*ax0 + a(3)*ax0^2 + a(4)*ax0^3;
    ay1 = a(2)*ax1 + 2*a(3)*ax0*ax1 + 3*a(4)*ax0^2*ax1;
    ay2 = a(3)*ax1^2 + 3*a(4)*ax0*ax1^2;
    ay3 = a(4)*ax1^3;
    x_init(:,i) = [ax0; ax1; 0; 0; 0; ay0; ay1; ay2; ay3; 0];
end
```

Optimization

```
solver_options = optimoptions('fmincon', 'MaxIterations', 1E4, ...
        'OptimalityTolerance', 1E-6, 'ConstraintTolerance', 1E-6, ...
        'MaxFunctionEvaluations', 1E6, 'Display', 'iter');

% Optimizing the trajectory using fmincon
x_opt = [];
for i = 1:(size_p(2)-1)

    [x_opt(:,i), fval, exit_flag, solver_output] = ....
            fmincon(@(x) my_cost(x, 100, 1), ...
            x_init(:,1), [], [], [], [], [], [], ...
        % <--- The two zeros here are lower bounds for the decision variables
            @(x) my_constraints(x, p(:,i), p(:,i+1), X(i), X(i+1), 8), ...
            solver_options);

end
```

Generating data and plotting the results

```
path_lenght1 = arc_length(x_opt, 1000, 2);
l = @(t) arc_length(x_opt, 1000, t);
kappa = @(t) curvature(x_opt, t);
xt = @(t) traj_x(x_opt,t);
yt = @(t) traj_y(x_opt,t);
angle = @(t) traj_angle(x_opt, t);
velocity = @(t) traj_vel(x_opt, t);


figure
fplot(xt, yt, [0,2])
title('py vs px')
xlabel 'px (m)';
ylabel 'py (m)';
hold on
plot(p1(1), p1(2), '.', 'markersize', 20);
plot(p2(1), p2(2), '.', 'markersize', 20);
plot(p3(1), p3(2), '.', 'markersize', 20);

figure
fplot(l, xt, [0,2])
```

```
title('px vs path length')
xlabel 'path length (m)';
ylabel 'px (m)';

figure
fplot(l, yt, [0,2])
title('py vs path length')
xlabel 'path length (m)';
ylabel 'py (m)';

figure
fplot(l, kappa, [0,2])
title('curvature vs path length')
xlabel 'path length (m)';
ylabel 'curvature';

figure
fplot(l, angle, [0,2])
title('angle vs path length')
xlabel 'path length (m)';
ylabel 'angle (deg)';
```

Function to calculate total arc length of the trajectory

```
function total_length = arc_length(x, M, t_end)
    dt = 1/M;
    length = 0;

    size_x = size(x);
    for i = 1 : size_x(2)
        for j = 0 : M
            t = j*dt;
            if t+i-1 >= t_end
                break;
            end
            dpx = x(2,i) + 2*x(3,i)*t + 3*x(4,i)*t^2 + 4*x(5,i)*t^3;
            dpy = x(7,i) + 2*x(8,i)*t + 3*x(9,i)*t^2 + 4*x(10,i)*t^3;
            d_length = sqrt(dpx^2 + dpy^2);

            if j == 0 || j == M
                length = length + 0.5*dt*d_length;
            else
                length = length + dt*d_length;
            end

        end
        if t+i-1 >= t_end
            break;
        end
    end
    total_length = length;
end
```

These functions returns px and py of the trajectories of given coefficients

```
function px = traj_x(x, t)
    size_x = size(x);
    t=min(max(t,0),size_x(2));
    i = t - mod(t,1);
    i = min(i, size_x(2) - 1);
    t = t-i;
    px = x(1,i+1) + x(2,i+1)*t + x(3,i+1)*t^2 + x(4,i+1)*t^3 + x(5,i+1)*t^4;
end
```

```
function py = traj_y(x, t)
    size_x = size(x);
    t=min(max(t,0),size_x(2));
    i = t - mod(t,1);
    i = min(i, size_x(2) - 1);
    t = t-i;
    py = x(6,i+1) + x(7,i+1)*t + x(8,i+1)*t^2 + x(9,i+1)*t^3 + x(10,i+1)*t^4;
end

function X = traj_angle(x, t)
    size_x = size(x);
    t=min(max(t,0),size_x(2));
    i = t - mod(t,1);
    i = min(i, size_x(2) - 1);
    t = t-i;
    dpx = x(2,i+1) + 2*x(3,i+1)*t + 3*x(4,i+1)*t^2 + 4*x(5,i+1)*t^3;
    dpy = x(7,i+1) + 2*x(8,i+1)*t + 3*x(9,i+1)*t^2 + 4*x(10,i+1)*t^3;
    X = atan2(dpy, dpx)*180/pi;
end

function X = traj_vel(x, t)
    size_x = size(x);
    t=min(max(t,0),size_x(2));
    i = t - mod(t,1);
    i = min(i, size_x(2) - 1);
    t = t-i;
    dpx = x(2,i+1) + 2*x(3,i+1)*t + 3*x(4,i+1)*t^2 + 4*x(5,i+1)*t^3;
    dpy = x(7,i+1) + 2*x(8,i+1)*t + 3*x(9,i+1)*t^2 + 4*x(10,i+1)*t^3;
    X = sqrt(dpy^2 + dpx^2);
end
```

Function to calculate curvature at given t

```
function kappa = curvature(x, t)
    size_x = size(x);
    t=min(max(t,0),size_x(2));
    i = t - mod(t,1);
    i = min(i, size_x(2) - 1);
    t = t-i;
    dpx = x(2,i+1) + 2*x(3,i+1)*t + 3*x(4,i+1)*t^2 + 4*x(5,i+1)*t^3;
    dpy = x(7,i+1) + 2*x(8,i+1)*t + 3*x(9,i+1)*t^2 + 4*x(10,i+1)*t^3;
    ddpx = 2*x(3,i+1) + 6*x(4,i+1)*t + 12*x(5,i+1)*t^2;
    ddpy = 2*x(8,i+1) + 6*x(9,i+1)*t + 12*x(10,i+1)*t^2;
    dp = [dpx; dpy; 0];
    ddp = [ddpx; ddpy; 0];
    temp1 = norm(cross(dp, ddp));
    temp2 = norm(dp);
    kappa = temp1/temp2^3;
end
```

Cost function for optimization

```
function f_ = my_cost(x, M, u)

       dt = 1 / M;
    H_tilda = zeros(10,10);
    for i = 0 : M
        t = i*dt;
        H_temp_1 = [0 1 2*t 3*t^2 4*t^3 0 0 0 0 0];
        H_temp_2 = [0 0 0 0 0 0 1 2*t 3*t^2 4*t^3];
        H_temp_3 = [0 0 2 6*t 12*t^2 0 0 0 0 0];
        H_temp_4 = [0 0 0 0 0 0 0 2 6*t 12*t^2];

        H_i = (H_temp_1'*H_temp_1 + H_temp_2'*H_temp_2) ...
```

```
            + u*(H_temp_3'*H_temp_3 + H_temp_4'*H_temp_4);
        if i == 0 || i == M
            H_tilda = H_tilda + 0.5*dt*H_i;
        else
            H_tilda = H_tilda + dt*H_i;
        end
    end

        f_ = x'*H_tilda*x;
end
```

Defining constraint for optimization

```
function [g_, h_] = my_constraints(x, p0, p1, X0, X1, v)
    M = 1000;

    % Heading constrains at the end points of the trajectory
    A_g = [0 0 0 0 0 0 -sign(sin(X0)) 0 0 0;
           0 -sign(cos(X0)) 0 0 0 0 0 0 0 0;
           0 0 0 0 0 0 -sign(sin(X1)) -2*sign(sin(X1)) -3*sign(sin(X1)) -4*sign(sin(X1));
           0 -sign(cos(X1)) -2*sign(cos(X1)) -3*sign(cos(X1)) -4*sign(cos(X1)) 0 0 0 0 0;
           zeros(M-1,10)];

    % Veloity constrain for the trajectory
    dt = 1/M;
    vel = zeros(M+3,1);
    for i = 1 : M-1
        t = i*dt;
        dpx = x(2) + 2*x(3)*t + 3*x(4)*t^2 + 4*x(5)*t^3;
        dpy = x(7) + 2*x(8)*t + 3*x(9)*t^2 + 4*x(10)*t^3;
        vel(i+4,1) = -sqrt(dpx^2 + dpy^2);
    end


        g_ = A_g*x + vel + [zeros(4,1);v*ones(M-1,1)];

    % Equality constrains
    A = [1 0 0 0 0 0 0 0 0 0;
         0 0 0 0 0 1 0 0 0 0;
         1 1 1 1 1 0 0 0 0 0;
         0 0 0 0 0 1 1 1 1 1;
         0 tan(X0) 0 0 0 0 -1 0 0 0;
         0 tan(X1) 2*tan(X1) 3*tan(X1) 4*tan(X1) 0 -1 -2 -3 -4];
    b = [p0(1) p0(2) p1(1) p1(2) 0 0]';
        h_ = A*x - b;

end
```

Published with MATLAB® R2022b