PROJECT REPORT

ENPM808C

PLANNING FOR AUTONOMOUS ROBOTS

SAMPLING BASED KINO-DYNAMIC PLANNING FOR SELF-DRIVING CARS

BY

LAKSHMAN KUMAR KUTTUVA NANDHAKUMAR

SAMPLING BASED KINO-DYNAMIC PLANNING FOR SELF-DRIVING CARS

ABSTRACT

Robotics has enabled accelerated growth in various industries. One such is the transportation industry. The concept of a Self-driving car was a dream a few decades ago, this has now become reality as a result of technological advancements, particularly in the field of perception and planning. All the major automotive and software companies, like Google, Tesla, Toyota etc. , have now dedicated a significant portion of their work force towards research in Self-Driving Cars. The basic planning problem for a self-driving car would be to drive from an initial point to the desired location, as commanded by the user. A sampling based planner that takes into account the velocity constraints has been proposed here.

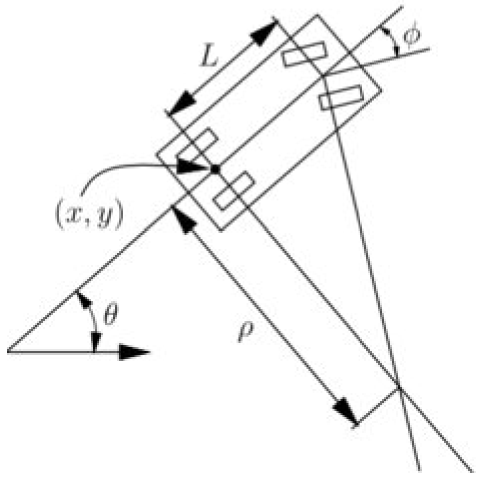
INTRODUCTION

The main motivation behind the development of self-driving cars is that they can significantly reduce the number of accidents happening on road. Also, the time spent commuting can now be spent for doing other useful things.

Cars can be considered as mobile robots that are non-holonomic in nature. In other words, the dimension of the admissible velocity space is smaller than the dimension of the configuration space. Sampling based planning algorithms are suitable for both holonomic and non-holonomic systems. PRM and RRT are two such algorithms that have been extensively used in solving these problems. These algorithms can be used to find a path to the goal location from an initial location taking into account the velocity constraints of the car. In other words, a trajectory from the source to the destination will be computed by these algorithms.

In order to use these algorithms, the forward kinematics of the car has to be derived. The forward kinematics of the car is computed with respect to its velocity and steering angle. The Ackermann Steering Model has been used to compute the forward kinematics of the car.

ACKERMANN STEERING MODEL



Ackermann steering geometry is a geometric arrangement of linkages in the steering of a car or other vehicle designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.

A simple approximation to perfect Ackermann steering geometry may be generated by moving the steering pivot points inward so as to lie on a line drawn between the steering kingpins and the centre of the rear axle. The steering pivot points are joined by a rigid bar called the tie rod which can also be part of the steering mechanism, in the form of a rack and pinions for instance. With perfect Ackermann, at any angle of steering, the centre point of all of the circles traced by all wheels will lie at a common point. Note that this may be difficult to arrange in practice with simple linkages, and designers are advised to draw or analyze their steering systems over the full range of steering angles.

Modern cars do not use *pure* Ackermann steering, partly because it ignores important dynamic and compliant effects, but the principle is sound for low-speed manoeuvres. Some race cars use reverse Ackermann geometry to compensate for the large difference in slip angle between the inner and outer front tyres while cornering at high speed. The use of such geometry helps reduce tyre temperatures during high-speed cornering but compromises performance in low-speed manoeuvres.

The equations for the simplified Ackermann Steering Model are given below.

$$\boldsymbol{x' = v.cos\theta}$$
$$\boldsymbol{y' = v.sin\theta}$$
$$\boldsymbol{\theta' = \frac{v}{L}.tan\phi}$$

Where $\boldsymbol{x'}$ is the horizontal velocity component of the car

$\boldsymbol{y'}$ is the vertical velocity component of the car

$\boldsymbol{\theta'}$ is the rotational velocity component of the car

$\boldsymbol{v}$ is the forward speed of the car with respect to ϕ**,**

ϕ is the steering angle of the car which is the angle between the line along the axis of the rear wheel joining the common point of rotation and the line connecting the common point of rotation & the centre of the axis along the front wheels.

$\boldsymbol{L}$ is the distance between the axis along the rear wheels and the axis along the front wheels

The position of the car can be obtained by integrating the equations above.

$$\boldsymbol{x = \int v.cos\theta\,.\,dt}$$
$$\boldsymbol{y = \int v.sin\theta\,.\,dt}$$
$$\boldsymbol{\theta = \int \frac{v}{L}.tan\phi\,.\,dt}$$

In this project ,

$\boldsymbol{v}$ has been restricted to two values. It can either be -1 or +1.

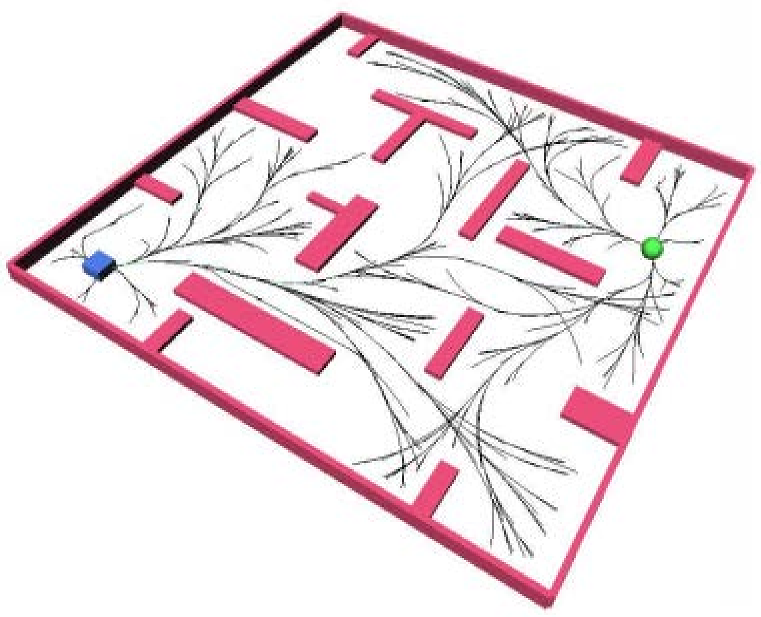
ϕ has been restricted between two limits. $\phi \in \boldsymbol{[-\frac{\pi}{4}, \frac{\pi}{4}]}$

The configuration space for the model remains 3 dimensional, C = $R^2 \times S^1$

The control action set which includes Velocity and Steering Angle will be 2 dimensional, U = R^2

RELATED WORKS

Google was among the first ones to start extensive research on Self Driving Cars. They have been using sophisticated perception systems to aid their planning algorithms. They have been using Rapidly exploring Random Trees (RRT) in conjunction with Reinforced Learning to solve the planning problem for Self-Driving Cars.



Steven.M.LaValle and James.J.Kuffner Jr, in their paper, "Randomized Kino-Dynamic Planning" , which was published in the International Journal of Robotics Research 2001, proposed a way to adapt RRT to perform trajectory planning and listed various suggestions on how to improve it. The algorithm proposed in this paper has been implemented and modified a little bit.

APPROACHES

Kino-Dynamic RRT

Pseudo Code

```
$KinoDynamicRRT(x_{init}, x_{goal})$

• $Initialize$ Tree $T$ With $x_{init}$

• For $n = 1\ to\ N$ Iterations do

        - Get $x_{rand}$ From $Goal\_Region\_Biased\_Random\_State(x_{goal})$

        - Get $x_{near}$ from $Nearest\_Neighbor(x_{rand}, T)$

        - Get [$x_{new}, u_{new}$] from $Best\_Collision\_Free\_Trajectory(x_{near}, x_{rand}, \Delta t)$

            o $T.add\_vertex(x_{new})$
            o $T.add\_edge(x_{near}, x_{new}, u_{new})$

        - $If\ x_{new} - x_{goal} < \delta$
            o $Return\ Path\_To\_Goal$ ()

• $Return\ Path\_To\_Point\_Nearest\_To\_Goal$ ( )
```

Explanation

The basic difference between the regular RRT and the Kino-Dynamic RRT is that when you select the new configuration to be added to the tree based on the random configuration, in addition to it being collision free , it should also satisfy the Ackermann steering model. Also instead of the Step Size parameter, Step Time parameter Δt is used.

The tree is first initialized with the starting configuration. Then for N number of iterations we will keep adding nodes to the tree unless and otherwise it reaches the goal configuration before that.

The process starts by first generating a random configuration. The random configuration here is generated such that it is biased towards the goal region, so that there is more chance that the tree rapidly expands towards the goal. Here it can be seen that instead of the exact goal point , the region surrounding the goal is biased since when the obstacle space around the goal is huge, exact path to the goal will be very difficult to find and so the tree will expand very slowly.

For this project, the algorithm has been designed such that , every once in 4 times the random node generated will be the goal node and every once in 7 times the random node generated will be in the region surrounding the goal. The radius of the goal region is 50 with the goal point as the center.

After getting the Goal Region Biased Random Configuration, we will find the nearest neighbour to it in the tree.

Now that we have the random and neighbouring configuration, we will have to generate a new configuration based on the Ackermann Steering Model. In order to do this, we will randomly assign the velocity to be either 1 or -1 and the steering angle to be somewhere between -45° and 45° . Based on the velocity and the steering angle, we will simulate a trajectory for a step time Δt. This process of simulating a trajectory is done for 'M' number of trials. For every trial the trajectory which has a point that is closest to the random configuration generated is tracked. After the 'M' Number of trials, the optimal trajectory with the closest point to the random configuration is selected and checked for collision. If its collision free, the point in the trajectory that is closest to the random configuration will be added as a new configuration to the tree and the corresponding velocity and steering angle to reach this new configuration from the neighbouring node is stored. If its not collision free, the entire new configuration generation process is repeated until a collision free new configuration is obtained.

Once the new configuration is added to the tree , it is checked to see if it approximately matches the goal point. If it does, then the algorithm returns the path to this new configuration. If it does not, the entire process is repeated for 'N' number of iterations.

After the algorithm reaches 'N' number of iterations, it returns the path nearest to the goal.

Here, on each iteration, only one configuration is added to the tree.

Kino-Dynamic PRM

Pseudo Code

```
$KinoDynamicPRM(x_{init}, x_{goal})$

• $Initialize$ Graph $G$ With $x_{init}$
• For $n = 1\ to\ N$ Iterations do
    - Get $x_{rand}$ From $Goal\_Region\_Biased\_Random\_State(x_{goal})$
    - Get $X_{near}$ from $K\_Nearest\_Neighbors(x_{rand}, G)$
    - For all $x_{near} \in X_{near}$
        o Get $[x_{new}, u_{new}]$ from $Best\_Collision\_Free\_Trajectory(x_{near}, x_{rand}, \Delta t)$
            ▪ $G.add\_vertex(x_{new})$
            ▪ $G.add\_edge(x_{near}, x_{new}, u_{new})$
• $Perform\ A\_Star\_Graph\_Search(x_{init}, x_{goal})$
• $Return\ Path\_To\_Point\_Nearest\_To\_Goal$
```

Explanation

The basic difference between the Kino-Dynamic RRT and the Kino-Dynamic PRM is that you select ‘K’ Nearest Neighbours to the generated random configuration, instead of just one. So for every iteration at most ‘K’ New Configurations will be added. Also instead of a tree structure, this will have a Graph structure.

The graph is first initialized with the starting configuration. Then for N number of iterations we will keep adding nodes to the graph.

The process starts by first generating a random configuration. The random configuration here is generated such that it is biased towards the goal region, so that there is more chance that the tree rapidly expands towards the goal. Here it can be seen that instead of the exact goal point , the region surrounding the goal is biased since when the obstacle space around the goal is huge, exact path to the goal will be very difficult to find and so the tree will expand very slowly.

For this project, the algorithm has been designed such that , every once in 4 times the random node generated will be the goal node and every once in 7 times the random node generated will be in the region surrounding the goal. The radius of the goal region is 50 with the goal point as the center.

After getting the Goal Region Biased Random Configuration, we will find 'K' nearest neighbour to it in the tree.

Now that we have the random and neighbouring configurations, we will have to generate a new configuration based on the Ackermann Steering Model. In order to do this, we will randomly assign the velocity to be either 1 or -1 and the steering angle to be somewhere between -45° and 45° . Based on the velocity and the steering angle, we will simulate a trajectory for a step time Δt. This process of simulating a trajectory is done for 'M' number of trials. For every trial the trajectory which has a point that is closest to the random configuration generated is tracked. After the 'M' Number of trials, the optimal trajectory with the closest point to the random configuration is selected and checked for collision. If its collision free, the point in the trajectory that is closest to the random configuration will be added as a new configuration to the tree and the corresponding velocity and steering angle to reach this new configuration from the neighbouring node is stored. If its not collision free, the entire new configuration generation process is repeated until a collision free new configuration is obtained. This is again repeated for all 'K' nearest neighbours.

The entire process goes on for 'N' number of iterations. After the algorithm reaches 'N' number of iterations, A Star graph search algorithm is implemented to find a path from the starting configuration to the configuration nearest to the goal configuration. The path returned by the A Star algorithm will be the shortest path to the goal configuration among the available paths.

Here, on each iteration, at most K configurations are added to the graph.

DESIGN OF EXPERIMENTS

Input

The input to the algorithm will be the Starting Configuration , the Goal Configuration and the Map containing the position, size and shape of various obstacles. In the algorithms, it has been assumed that perception of the environment by the car is perfect and the also the car knows where it is exactly located in the map. In other words, here the planning is done under certainty.

Output

The output of the algorithm will be the trajectory from the Starting Configuration to the Goal Configuration based on Ackermann Steering Model

Experiments

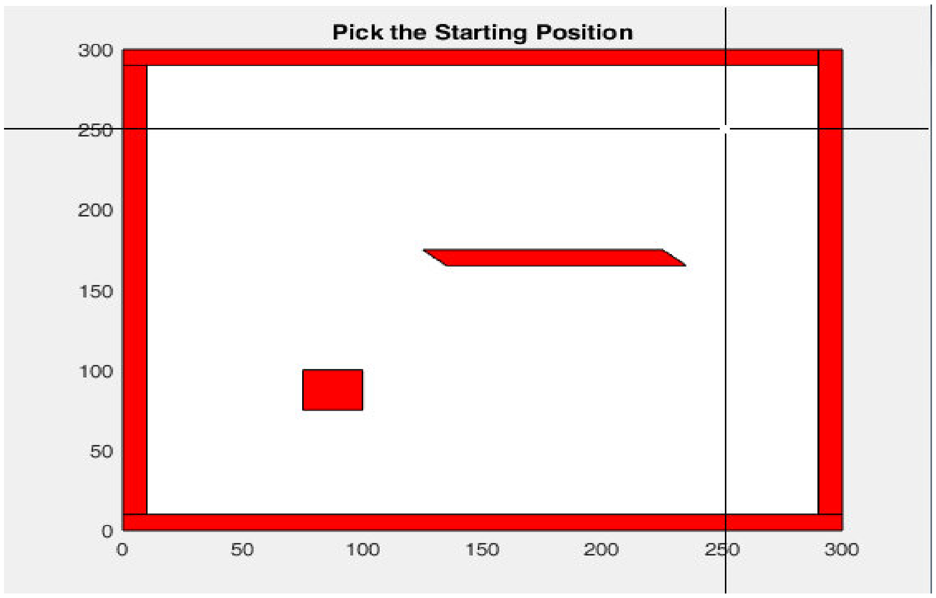
A lot of experiments can be done to check and improve the performance of the algorithms. Some of them are listed below.

- Variation of number of obstacles in the map
- Variation of the shape of obstacles
- Variation of the size of the obstacles
- Variation of the starting and goal configurations
- Variation of algorithm parameters like Step Time, Goal Region Radius, Number of Neighbors to be selected, Number of Iterations to be performed, and so on.

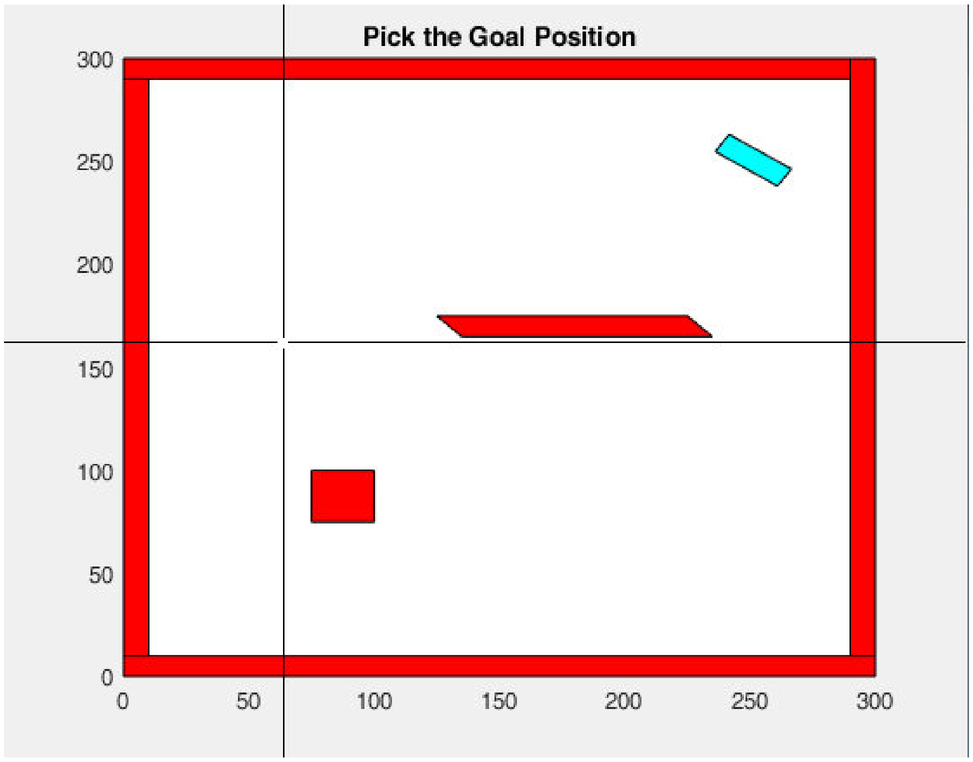
MATLAB SIMULATION RESULTS

Kino-Dynamic RRT

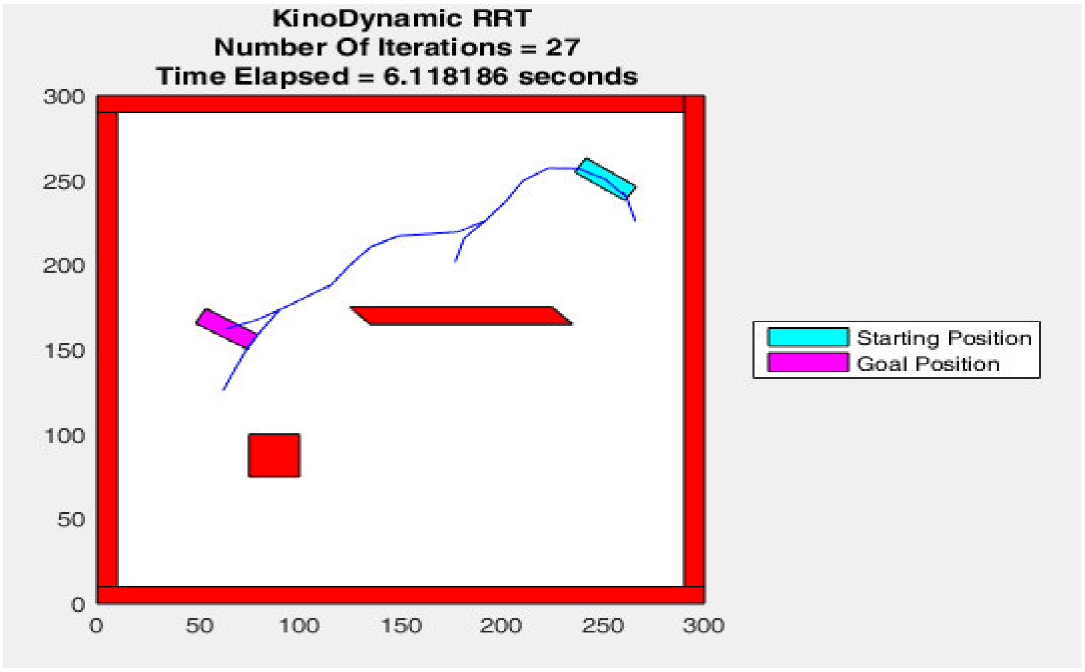
Picking the Starting Configuration



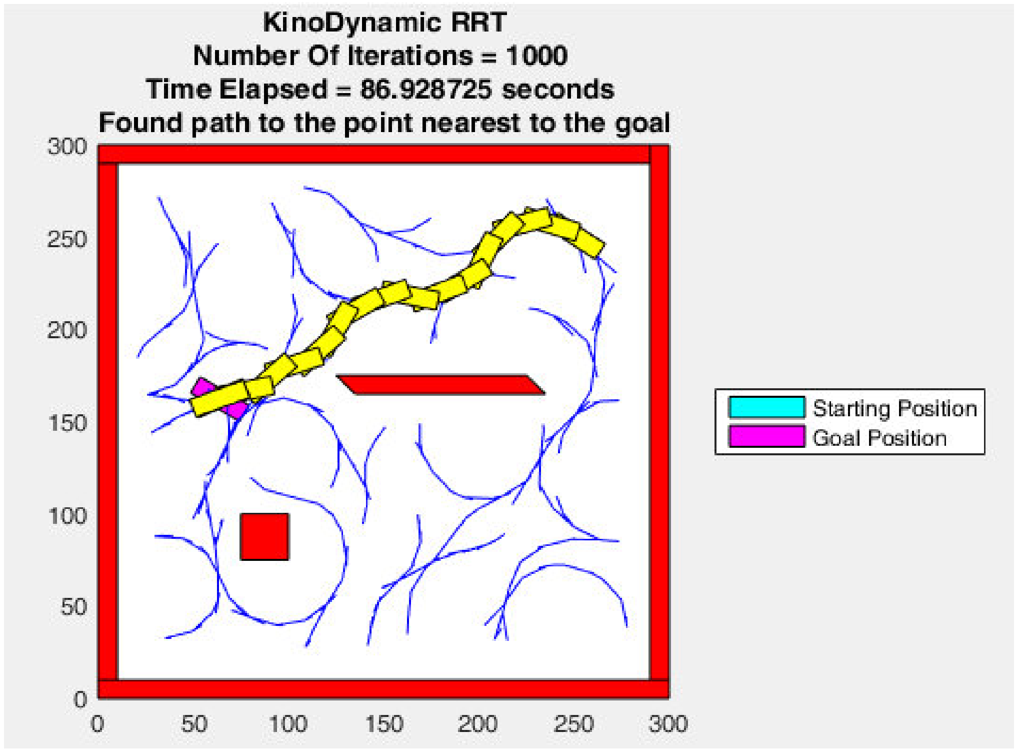
Picking the Goal Configuration



Building the Tree

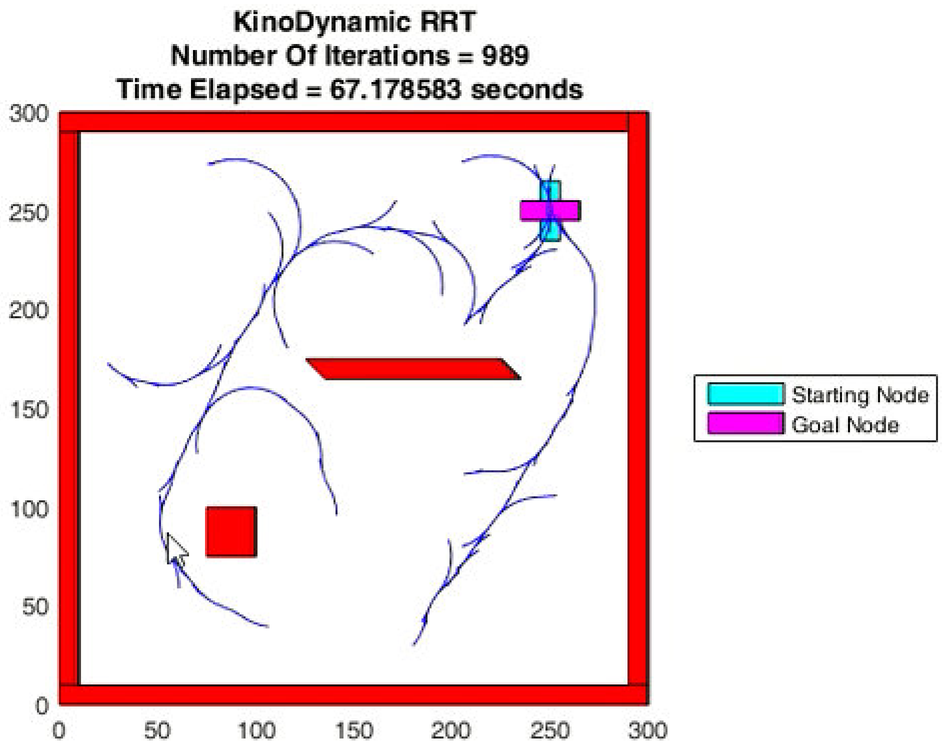


Trajectory to the Goal Configuration

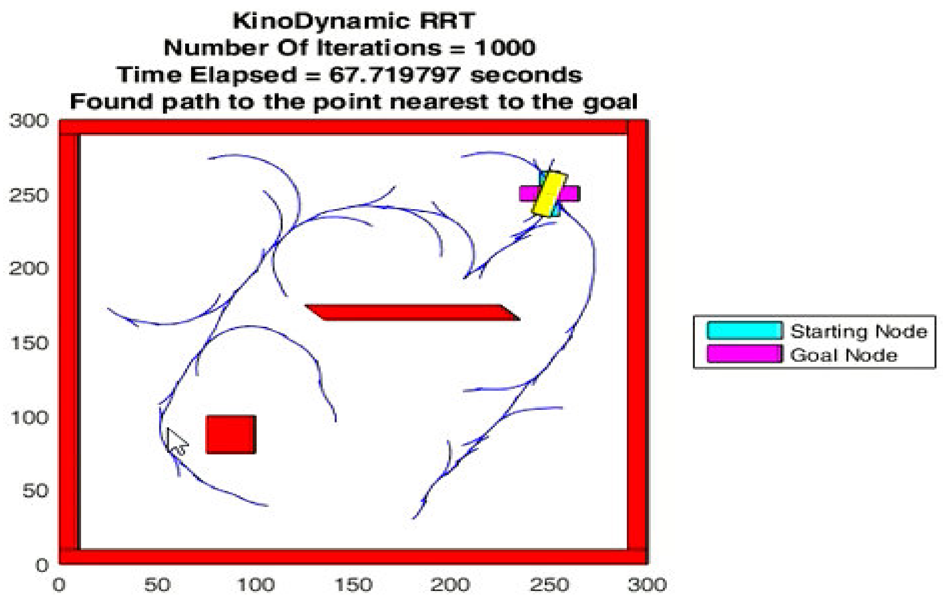


Special Case

Initial and goal configuration are at the same position, but different orientation.

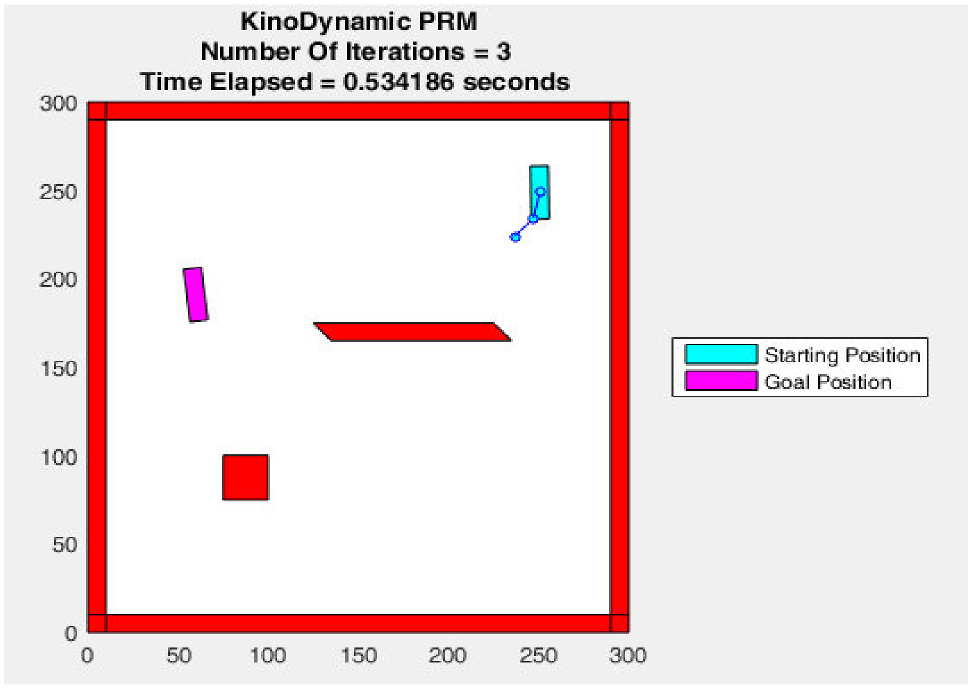


Final position of the car after trajectory is generated by RRT

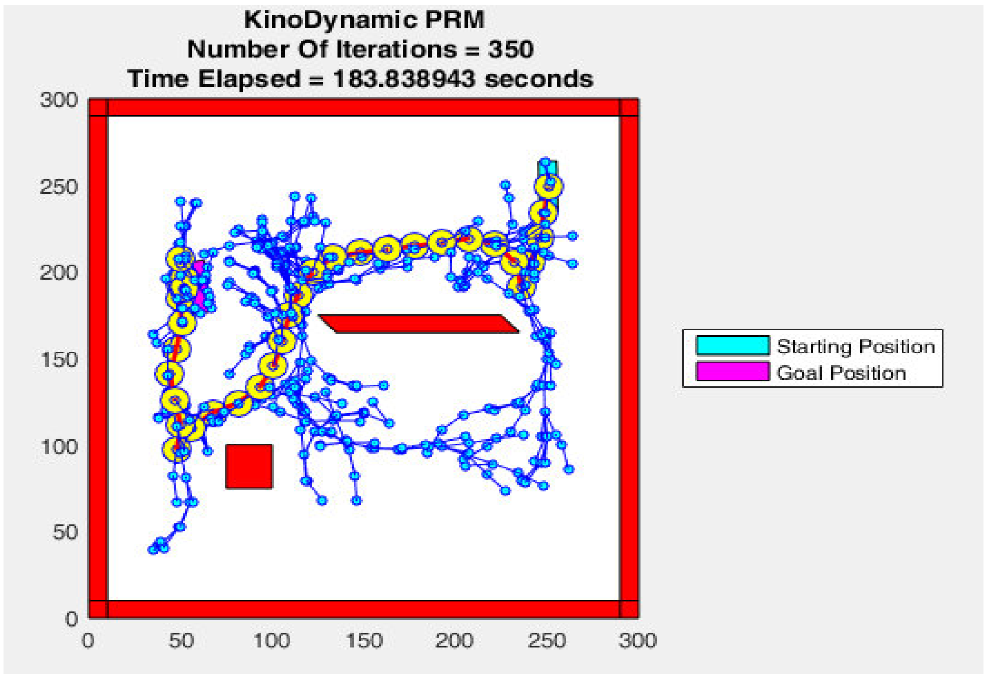


Kino-Dynamic PRM

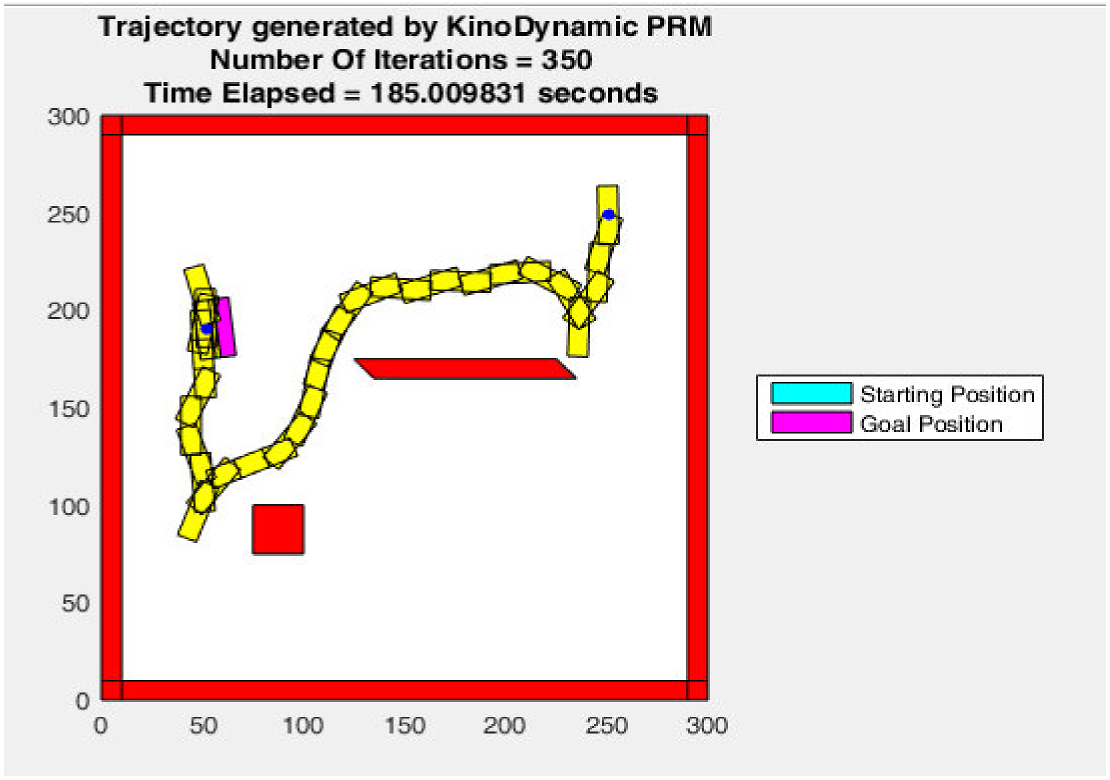
Initial and Goal Configurations



Graph generated by PRM

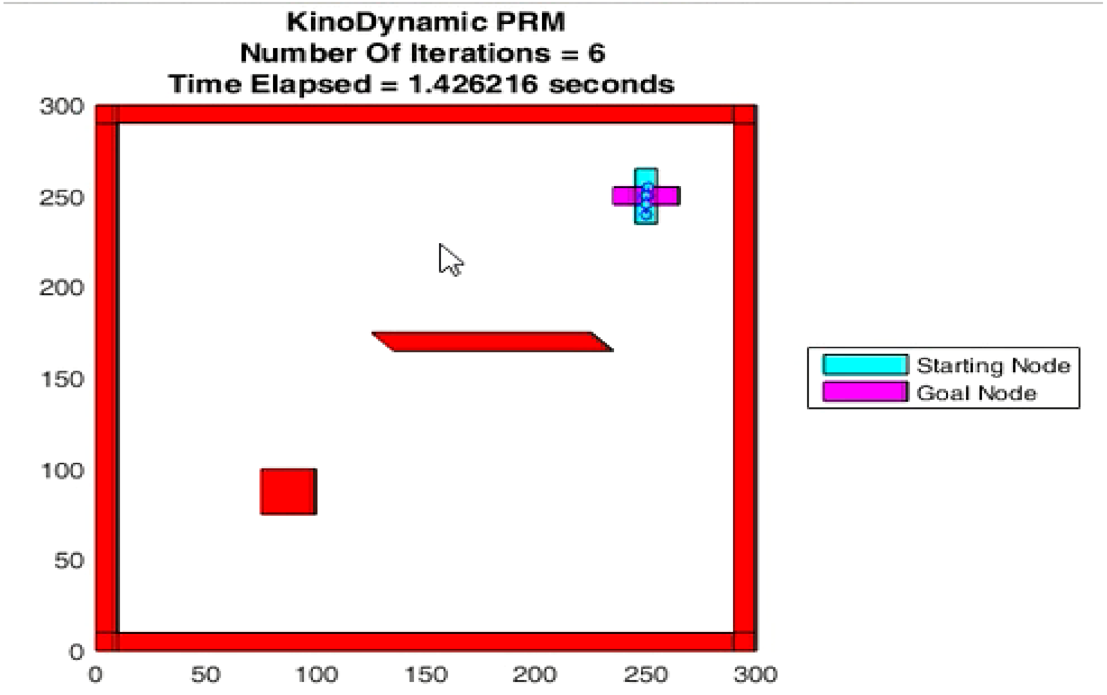


Trajectory from starting configuration to goal configuration generated by PRM

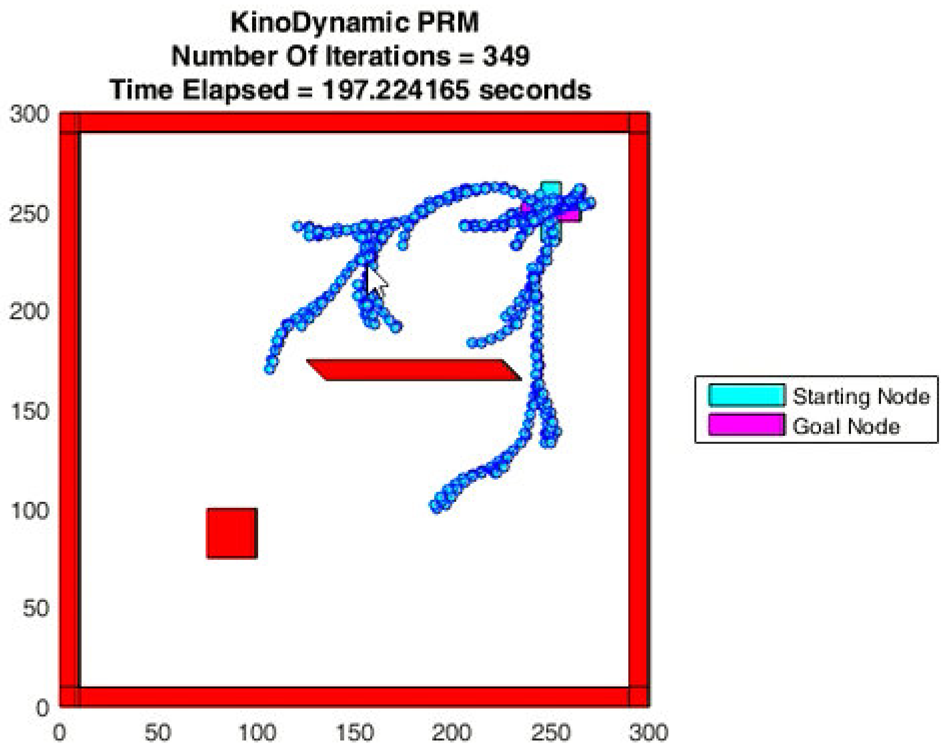


Special Case

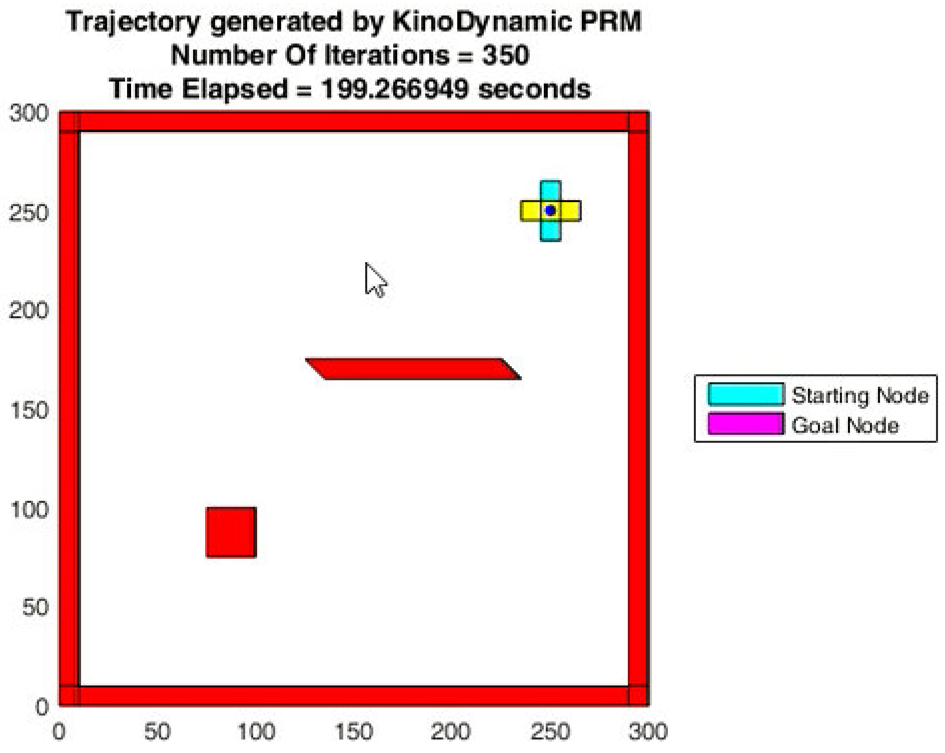
Initial and goal configuration are at the same position, but different orientation.



Graph built by the PRM



Final position of the car after trajectory is generated by PRM



Kino-Dynamic RRT vs Kino-Dynamic PRM

As seen from the results, Kino-Dynamic RRT and Kino-Dynamic PRM have their own advantages and disadvantages.

Kino-Dynamic RRT was faster in finding an approximate solution. However the solution is not better than what was generated by PRM. Also, when the initial configuration was very close to the goal configuration, the trajectory generated by RRT did not even reach the half way point to the goal.

On the other hand, Kino-Dynamic PRM was slower than RRT. However they provided way better solutions that RRT and also when the initial configuration is close to the goal configuration, the trajectory generated by PRM reached the exact goal configuration with ease. Also , Kino-Dynamic PRM provides alternate trajectories to the same point , which can be made use of when there are dynamic obstacles.

Hence it can be said that when the initial configuration and goal configuration are further apart , Kino-Dynamic RRT can be used. When the car gets very close to the goal configuration , Kino-Dynamic PRM can be used.

FUTURE WORKS

The algorithms proposed here do not provide an optimal solution. Hence the algorithms proposed here can be upgraded to Kino-Dynamic RRT* and Kino-Dynamic PRM*, which give optimal solutions. Also, the algorithms can be modified to accommodate Dynamic Obstacles and uncertainty of its location in the map.

REFERENCES

[1] Steven.M.LaValle and James.J.Kuffner Jr, “Randomized Kino-Dynamic Planning” , International Journal of Robotics Research 2001

[2] Steven M LaValle , “Planning Algorithms”, Cambridge University Press 2005

[3] Google, “Self-Driving Car” , https://www.google.com/selfdrivingcar/

[4] Jarrod.M.Snider,”Automatic Steering Methods for Autonomous Automobile Path Tracking”, Carnegie Mellon University

[5] Garrote et Al, “An RRT-based Navigation Approach for Mobile Robots and Automated Vehicles” , University of Coimbra, Portugal

[6] Jun Qu, “Nonholonomic Mobile Robot Motion Planning”, Motion Strategy Project, http://msl.cs.uiuc.edu/~lavalle/cs576_1999/projects/junqu/

[7] Peter Corke, “Robotics, Vision and Control”, Springer 2011

[8] Peter Corke, “MATLAB toolboxes: robotics and vision for students and teachers”, IEEE Robotics and Automation Magazine, Volume 14(4), December 2007

[9] Bruno Siciliano et al.” Robotics Modelling, Planning and Control”, Springer 2009.

[10] Peter Corke ,” A Robotics Toolbox for MATLAB”, IEEE Robotics and Automation Magazine, Volume 3(1), March 1996

APPENDIX

MATLAB CODE

Kino-Dynamic RRT

```
function KinoDynamicRRT

    clc;
    figure;
    pause(0);
    ShowMap;

    RobotLocationXInit = 250;
    RobotLocationYInit = 250;
    RobotOrientationInit = 0;
    RobotLocationXGoal = 50;
    RobotLocationYGoal = 50;
    RobotOrientationGoal = 90;

    MaxNumberOfIterations = 1000;
    MaxXYThreshold = 5;
    MaxOrientationThreshold = 15;
    MaxStepSize = 300;
    MinimumX = 0; MaximumX = 300;
    MinimumY = 0; MaximumY = 300;
    MinimumOrientation = 0 ; MaximumOrientation = 360;

    Velocity = 1;
    SteeringAngleMax = 45;
    NumberOfTrials = 20;
    SimulationTime = 15;
    SamplingInterval = 0.1;
    CarLength = 20;
    GoalBiasRegionStepSize = 50;
    Alternator = 0;

    title('Pick the Starting Position')

    while true;


     [RobotLocationXInit,RobotLocationYInit] =ginput(1);
     RobotOrientationInit = round((MaximumOrientation-
MinimumOrientation)*rand + MinimumOrientation );

       if ( IsThereAnObstacle(RobotLocationXInit,RobotLocationYInit) == 0 &&
CheckCollisionDetectionCorners(RobotLocationXInit,RobotLocationYInit,RobotOri
entationInit) == 0 &&
CheckCollisionDetection(RobotLocationXInit,RobotLocationYInit,RobotOrientatio
nInit) == 0 )
          break ;
       else
```

```
           title({'The selected starting position is not a valid
configuration.' ; ' Select the starting position again'})
       end

    end

    RobotLocationX = RobotLocationXInit;
    RobotLocationY = RobotLocationYInit;
    RobotOrientation = RobotOrientationInit;

    H1 = DrawRobot(RobotLocationX,RobotLocationY,RobotOrientation,'cyan');
    pause(0);
    hold on;
    title('Pick the Goal Position')

     while true;


     [RobotLocationXGoal,RobotLocationYGoal] =ginput(1);
     RobotOrientationGoal = round((MaximumOrientation-
MinimumOrientation)*rand + MinimumOrientation );

       if ( IsThereAnObstacle(RobotLocationXGoal,RobotLocationYGoal) == 0 &&
CheckCollisionDetectionCorners(RobotLocationXGoal,RobotLocationYGoal,RobotOri
entationGoal) == 0 &&
CheckCollisionDetection(RobotLocationXGoal,RobotLocationYGoal,RobotOrientatio
nGoal) == 0 )
          break ;
       else
           title({'The selected goal position is not a valid configuration';'
Select the goal position again'});
       end

     end

    title('KinoDynamic RRT')



    H2 =
DrawRobot(RobotLocationXGoal,RobotLocationYGoal,RobotOrientationGoal,'magenta
');

    pause(0);

    legend([H1 H2],{'Starting Position','Goal
Position'},'Location','eastoutside');

    tree.vertex(1).x = RobotLocationXInit;
    tree.vertex(1).y = RobotLocationYInit;
    tree.vertex(1).o = RobotOrientationInit;

    tree.vertex(1).PreviousX = RobotLocationXInit;
    tree.vertex(1).PreviousY = RobotLocationYInit;
    tree.vertex(1).PreviousOrientation = RobotOrientationInit;
    tree.vertex(1).Distance=0;
    tree.vertex(1).ind = 1; tree.vertex(1).indPrev = 0;

    tic;


    for iter = 2:MaxNumberOfIterations


        str = sprintf('Number Of Iterations = %d ',iter);
        TimeElapsed =  sprintf('Time Elapsed = %f seconds ',toc);

        title({'KinoDynamic RRT';str;TimeElapsed});


            while true
                while true

                Alternator = Alternator + 1;

                if mod(Alternator,7) == 0
                                RX = (RobotLocationXGoal -
(GoalBiasRegionStepSize/2));
                                RandX = (GoalBiasRegionStepSize)*rand;

                                if RX < MinimumX
                                    RX = MinimumX ;
                                else
                                    if RX > (MaximumX -
GoalBiasRegionStepSize)
                                        RX = MaximumX -
GoalBiasRegionStepSize;
                                    end
                                end


                                RandomVariableX = round( RandX + RX );

                                if RandomVariableX < MinimumX
                                    RandomVariableX = MinimumX;
                                else
                                    if RandomVariableX > MaximumX
                                        RandomVariableX = MaximumX;
                                    end
                                end

                                RY = (RobotLocationYGoal -
(GoalBiasRegionStepSize/2));
                                RandY = (GoalBiasRegionStepSize)*rand;

                                if RY < MinimumY
                                    RY = MinimumY ;
                                else
                                    if RY > (MaximumY -
GoalBiasRegionStepSize)
```

```
                                        RY = MaximumY -
GoalBiasRegionStepSize;
                                    end
                                end

                                RandomVariableY = round( RandY + RY );

                                if RandomVariableY < MinimumY
                                    RandomVariableY = MinimumY;
                                else
                                    if RandomVariableY > MaximumY
                                        RandomVariableY = MaximumY;
                                    end
                                end

                                RO = (RobotOrientationGoal -
(GoalBiasRegionStepSize/2));
                                RandO = (GoalBiasRegionStepSize)*rand;

                                if RO < MinimumOrientation
                                    RO = MinimumOrientation ;
                                else
                                    if RO > (MaximumOrientation -
GoalBiasRegionStepSize)

                                        RO = MaximumOrientation -

GoalBiasRegionStepSize;
                                    end
                                end

                                RandomVariableO = round( RandO + RO );

                                if RandomVariableO < MinimumOrientation
                                    RandomVariableO = MinimumOrientation;
                                else
                                    if RandomVariableO > MaximumOrientation
                                        RandomVariableO = MaximumOrientation;
                                    end
                                end


                else

                    if mod(Alternator,4) == 0
                          RandomVariableX = RobotLocationXGoal ;
                          RandomVariableY = RobotLocationYGoal ;
                          RandomVariableO = RobotOrientationGoal;
                    else
                        RandomVariableX = round(MaxStepSize*rand);
                        RandomVariableY = round(MaxStepSize*rand);
                        RandomVariableO = round((MaximumOrientation-
MinimumOrientation)*rand + MinimumOrientation );
                    end
                end

                if ( IsThereAnObstacle(RandomVariableX,RandomVariableY) == 0
&&
CheckCollisionDetectionCorners(RandomVariableX,RandomVariableY,RandomVariable
O) == 0 &&
CheckCollisionDetection(RandomVariableX,RandomVariableY,RandomVariableO) == 0
)
                    break ;
                end

                end



                Distance = Inf*ones(1,length(tree.vertex));

                for j = 1:length(tree.vertex)
                  Distance(j) = sqrt( (RandomVariableX - tree.vertex(j).x)^2
+ (RandomVariableY-tree.vertex(j).y)^2 + (deg2rad(RandomVariableO)-
deg2rad(tree.vertex(j).o))^2 );
                end

                [val, ind] = min(Distance);

                NearestNodeX = tree.vertex(ind).x;
                NearestNodeY = tree.vertex(ind).y;
                NearestNodeOrientation = tree.vertex(ind).o;

                FoundAFeasiblePath = 1;
                NumberOfTries =0;
                            while true


[PathPoints,Index,OptimalVelocity,OptimalSteeringAngle] =
BestPath(NearestNodeX,NearestNodeY,NearestNodeOrientation,RandomVariableX,Ran
domVariableY,RandomVariableO);

                            ObstacleFlag = 0;

                            for m = 1 : Index
                                  if (
IsThereAnObstacle(PathPoints(m,1),PathPoints(m,2)) == 1 ||
CheckCollisionDetectionCorners(PathPoints(m,1),PathPoints(m,2),PathPoints(m,3
)) == 1 ||
CheckCollisionDetection(PathPoints(m,1),PathPoints(m,2),PathPoints(m,3)) == 1
)

                                     ObstacleFlag = 1;
                                     break ;
                                  end
                            end

                            if ObstacleFlag == 0
                                break;
                            end

                            NumberOfTries = NumberOfTries + 1 ;

                            if NumberOfTries > 3
```

```
                                FoundAFeasiblePath = 0;
                                break;
                            end

                            end

                if FoundAFeasiblePath == 1
                    break;
                end

                end



            NewX = PathPoints(Index,1);
            NewY = PathPoints(Index,2);
            NewO = PathPoints(Index,3);

            tree.vertex(iter).x = NewX; tree.vertex(iter).y = NewY;
    tree.vertex(iter).o = NewO;
            tree.vertex(iter).v = OptimalVelocity; tree.vertex(iter).s =
    OptimalSteeringAngle;

            tree.vertex(iter).PreviousX = tree.vertex(ind).x;
            tree.vertex(iter).PreviousY = tree.vertex(ind).y;
            tree.vertex(iter).PreviousOrientation = tree.vertex(ind).o;
            tree.vertex(iter).ind = iter; tree.vertex(iter).indPrev = ind;

            if sqrt( (NewX-RobotLocationXGoal)^2 + (NewY-RobotLocationYGoal)^2 )
    <= MaxXYThreshold
                if abs(NewO - RobotOrientationGoal) < MaxOrientationThreshold
                plot([tree.vertex(iter).x;
    tree.vertex(ind).x],[tree.vertex(iter).y; tree.vertex(ind).y], 'r');
                break;
                end
            end

            plot([tree.vertex(iter).x; tree.vertex(ind).x],[tree.vertex(iter).y;
    tree.vertex(ind).y], 'b');
            pause(0);

        end


        if iter >= MaxNumberOfIterations


            Statement = 'Found path to the point nearest to the goal';

            title({'KinoDynamic RRT';str;TimeElapsed;Statement});

        else
            Statement = 'Found path to the goal';

            title({'KinoDynamic RRT';str;TimeElapsed;Statement});
        end




            Distance = Inf*ones(1,length(tree.vertex));
            for j = 1:length(tree.vertex)
              Distance(j) = sqrt( (RobotLocationXGoal - tree.vertex(j).x)^2 +
    (RobotLocationYGoal-tree.vertex(j).y)^2 + (deg2rad(RobotOrientationGoal)-
    deg2rad(tree.vertex(j).o))^2 );
            end

              [val, ind] = min(Distance);

            NearestNodeX = tree.vertex(ind).x;
            NearestNodeY = tree.vertex(ind).y;
            NearestNodeOrientation = tree.vertex(ind).o;

            path.pos(1).x = NearestNodeX; path.pos(1).y = NearestNodeY;
    path.pos(1).o = NearestNodeOrientation;
            pathIndex = tree.vertex(ind).ind;
            j= -1;
            PathIndices = [pathIndex];

            while true
                pathIndex = tree.vertex(pathIndex).indPrev;
                PathIndices = union(PathIndices,pathIndex);

                if pathIndex == 1
                    break;
                end

            end



            PI = 1;

            while 1
                path.pos(j+2).x = tree.vertex(PathIndices(PI)).x;
                path.pos(j+2).y = tree.vertex(PathIndices(PI)).y;
                path.pos(j+2).o = tree.vertex(PathIndices(PI)).o;
                  PI = PI + 1;

                H3 =
    DrawRobot(path.pos(j+2).x,path.pos(j+2).y,path.pos(j+2).o,'yellow');

                pause(0.2);

                if PI == length(PathIndices)+1
                    break;

                else
                   % delete(H3);
                end

                j=j+1;
            end



            for j = 2:length(path.pos)

                plot([path.pos(j).x; path.pos(j-1).x;], [path.pos(j).y;
    path.pos(j-1).y], 'r', 'Linewidth', 3);

            end




        function H =
    DrawRobot(RobotLocation_X,RobotLocation_Y,Robot_Orientation,Color)

        RobotLocationXCorner1 = RobotLocation_X + (-15*cosd(Robot_Orientation)) -
    ( 5*sind(Robot_Orientation)) ;
        RobotLocationYCorner1 = RobotLocation_Y + (-15*sind(Robot_Orientation)) +
    ( 5*cosd(Robot_Orientation)) ;

        RobotLocationXCorner2 = RobotLocation_X + (15*cosd(Robot_Orientation)) -
    ( 5*sind(Robot_Orientation)) ;
        RobotLocationYCorner2 = RobotLocation_Y + (15*sind(Robot_Orientation)) +
    ( 5*cosd(Robot_Orientation)) ;

        RobotLocationXCorner3 = RobotLocation_X + (15*cosd(Robot_Orientation)) -
    ( -5*sind(Robot_Orientation)) ;
        RobotLocationYCorner3 = RobotLocation_Y + (15*sind(Robot_Orientation)) +
    ( -5*cosd(Robot_Orientation)) ;

        RobotLocationXCorner4 = RobotLocation_X + (-15*cosd(Robot_Orientation)) -
    ( -5*sind(Robot_Orientation)) ;
        RobotLocationYCorner4 = RobotLocation_Y + (-15*sind(Robot_Orientation)) +
    ( -5*cosd(Robot_Orientation)) ;

        RobotXVertices = [RobotLocationXCorner1 RobotLocationXCorner2
    RobotLocationXCorner3 RobotLocationXCorner4 ];
        RobotYVertices = [RobotLocationYCorner1 RobotLocationYCorner2
    RobotLocationYCorner3 RobotLocationYCorner4 ];

        H = patch(RobotXVertices,RobotYVertices,Color);

        end


        function [PathPoints,Index,OptimalVelocity,OptimalSteeringAngle] =
    BestPath(NearestNodeX,NearestNodeY,NearestNodeOrientation,RandomVariableX,Ran
    domVariableY,RandomVariableO)

        DistanceThreshold = Inf;
        for i = 1 : NumberOfTrials

                if rand >= 0.5
                    Speed = Velocity;
                else
                    Speed = -Velocity;
                end

                SteeringAngle = (2*rand - 1)* SteeringAngleMax;

                TempX = NearestNodeX;
                TempY = NearestNodeY;
                TempO = NearestNodeOrientation;
                PathHistory = [];

                for r = 1 : (SimulationTime/SamplingInterval)

                    TempX = TempX + (Speed*SamplingInterval*cosd(TempO));
                    TempY = TempY + (Speed*SamplingInterval*sind(TempO));
                    TempO = rad2deg(deg2rad(TempO) +
    (Speed*SamplingInterval/CarLength*deg2rad(SteeringAngle)));
                    PathHistory = [PathHistory;[TempX TempY TempO]];

                end

                DistanceBetweenPathPointsFromRandomNumber =
    Inf*ones(1,length(PathHistory));
                for k = 1:length(PathHistory)
                  DistanceBetweenPathPointsFromRandomNumber(k) = sqrt(
    (RandomVariableX - PathHistory(k,1))^2 + (RandomVariableY-PathHistory(k,2))^2
    + ((deg2rad(RandomVariableO) - deg2rad(PathHistory(k,3))))^2 );
                end

                [MinimumDistanceBetweenPathPointsFromRandomNumber, IndexNumber] =
    min(DistanceBetweenPathPointsFromRandomNumber);

                if MinimumDistanceBetweenPathPointsFromRandomNumber <
    DistanceThreshold

                    DistanceThreshold =
    MinimumDistanceBetweenPathPointsFromRandomNumber;
                    PathPoints = PathHistory;
                    Index = IndexNumber;
                    OptimalVelocity = Speed;
                    OptimalSteeringAngle = SteeringAngle;


                end

        end

        end

    end
```

Kino-Dynamic PRM

```
function KinoDynamicPRM

    clc;

    pause(0);

    figure;

    ShowMap;

    RobotLocationXInit = 250;
    RobotLocationYInit = 250;
    RobotOrientationInit = 0;

    RobotLocationXGoal = 50;
    RobotLocationYGoal = 50;
    RobotOrientationGoal = 90;

    MaxNumberOfPointsToBePlaced = 350;

    MaxStepSize = 300;

    MinimumX = 0; MaximumX = 300;
    MinimumY = 0; MaximumY = 300;

    MinimumOrientation = 0 ; MaximumOrientation = 360;

    Velocity = 1;
    SteeringAngleMax = 45;

    NumberOfTrials = 20;
    SimulationTime = 15;
    SamplingInterval = 0.1;
    CarLength = 20;

    MaximumNumberOfNodes = 2;

    StepSize = 50;

    title('Select the Starting Position')

    while true;


     [RobotLocationXInit,RobotLocationYInit] =ginput(1);
     RobotOrientationInit = round((MaximumOrientation-
MinimumOrientation)*rand + MinimumOrientation );

       if ( IsThereAnObstacle(RobotLocationXInit,RobotLocationYInit) == 0 &&
CheckCollisionDetectionCorners(RobotLocationXInit,RobotLocationYInit,RobotOri
entationInit) == 0 &&
CheckCollisionDetection(RobotLocationXInit,RobotLocationYInit,RobotOrientatio
nInit) == 0 )
```

```
          break ;
       else
           title({'The selected starting position is not a valid
configuration.';' Select the starting position again'})
       end

    end



    RobotLocationX = RobotLocationXInit;
    RobotLocationY = RobotLocationYInit;
    RobotOrientation = RobotOrientationInit;

    H1 = DrawRobot(RobotLocationX,RobotLocationY,RobotOrientation,'cyan');

    pause(0);

    hold on;

     title('Select the Goal Position')

     while true;



     [RobotLocationXGoal,RobotLocationYGoal] =ginput(1);
     RobotOrientationGoal = round((MaximumOrientation-
MinimumOrientation)*rand + MinimumOrientation );
       if ( IsThereAnObstacle(RobotLocationXGoal,RobotLocationYGoal) == 0 &&
CheckCollisionDetectionCorners(RobotLocationXGoal,RobotLocationYGoal,RobotOri
entationGoal) == 0 &&
CheckCollisionDetection(RobotLocationXGoal,RobotLocationYGoal,RobotOrientatio
nGoal) == 0 )
          break ;
       else
           title({'The selected goal position is not a valid configuration';'
Select the goal position again'})
       end

     end

    title('KinoDynamic PRM')

    H2 =
DrawRobot(RobotLocationXGoal,RobotLocationYGoal,RobotOrientationGoal,'magenta
');

    pause(0);

    legend([H1 H2],{'Starting Position','Goal
Position'},'Location','eastoutside');

    PRMGraph = PGraph(3);

    StartNode = [RobotLocationXInit;RobotLocationYInit;RobotOrientationInit];
    StartNodeID = PRMGraph.add_node(StartNode);
    PRMGraph.highlight_node(StartNodeID,'NodeSize',4,'NodeFaceColor','cyan');
    pause(0);
    Alternator = 0;



    tic
    for NumberOfSamplePoints = 2 : MaxNumberOfPointsToBePlaced

        str = sprintf('Number Of Iterations = %d ',NumberOfSamplePoints);
        TimeElapsed =  sprintf('Time Elapsed = %f seconds ',toc);

        title({'KinoDynamic PRM';str;TimeElapsed});
           while true
                while true

                    Alternator = Alternator + 1;

                     %Generate Random Number in Goal Biased Region once in 15
times

                            if mod(Alternator,7) == 0
                                            RX = (RobotLocationXGoal -
(StepSize/2));
                                            RandX = (StepSize)*rand;

                                            if RX < MinimumX
                                                RX = MinimumX ;
                                            else
                                                if RX > (MaximumX - StepSize)
                                                    RX = MaximumX - StepSize;
                                                end
                                            end


                                            RandomVariableX = round( RandX +
RX );

                                            if RandomVariableX < MinimumX
                                                RandomVariableX = MinimumX;
                                            else
                                                if RandomVariableX > MaximumX
                                                    RandomVariableX =
MaximumX;
                                                end
                                            end

                                            RY = (RobotLocationYGoal -
(StepSize/2));
                                            RandY = (StepSize)*rand;

                                            if RY < MinimumY
                                                RY = MinimumY ;
                                            else
                                                if RY > (MaximumY - StepSize)
```

```
                                                    RY = MaximumY - StepSize;
                                                end
                                            end

                                            RandomVariableY = round( RandY +
RY );

                                            if RandomVariableY < MinimumY
                                                RandomVariableY = MinimumY;
                                            else
                                                if RandomVariableY > MaximumY
                                                    RandomVariableY =
MaximumY;
                                                end
                                            end

                                            RO = (RobotOrientationGoal -
(StepSize/2));
                                            RandO = (StepSize)*rand;

                                            if RO < MinimumOrientation
                                                RO = MinimumOrientation ;
                                            else
                                                if RO > (MaximumOrientation -
StepSize)
                                                    RO = MaximumOrientation -
StepSize;
                                                end
                                            end

                                            RandomVariableO = round( RandO +
RO );

                                            if RandomVariableO <
MinimumOrientation
                                                RandomVariableO =
MinimumOrientation;
                                            else
                                                if RandomVariableO >
MaximumOrientation
                                                    RandomVariableO =
MaximumOrientation;
                                                end
                                            end


                            else

                                 %Generate the Goal Node once in 4 times

                               if mod(Alternator,4) == 0
                              RandomVariableX = RobotLocationXGoal ;
                              RandomVariableY = RobotLocationYGoal ;
                              RandomVariableO = RobotOrientationGoal;
                               else
```

```
                                    RandomVariableX =
round(MaxStepSize*rand);
                                    RandomVariableY =
round(MaxStepSize*rand);
                                    RandomVariableO =
round((MaximumOrientation-MinimumOrientation)*rand + MinimumOrientation );
                               end

                            end

                    if ( IsThereAnObstacle(RandomVariableX,RandomVariableY)
== 0 &&
CheckCollisionDetectionCorners(RandomVariableX,RandomVariableY,RandomVariable
O) == 0 &&
CheckCollisionDetection(RandomVariableX,RandomVariableY,RandomVariableO) == 0
)
                        break ;
                    end


                end



                    RandomNode =
[RandomVariableX;RandomVariableY;RandomVariableO];

                    [Distances,NeighborNodeID] =
PRMGraph.distances(RandomNode);

                    if length(Distances) < MaximumNumberOfNodes
                        MaxNodes = length(Distances);
                    else
                        MaxNodes = MaximumNumberOfNodes ;
                    end

                    NumberOfNodesForWhichPathIsNotFound = 0;

                    for node = 1 : MaxNodes
                        if PRMGraph.coord(NeighborNodeID(node)) == RandomNode
                            continue;
                        end


                        NeighborNode = PRMGraph.coord(NeighborNodeID(node)) ;
                        NearestNodeX = NeighborNode(1);
                        NearestNodeY = NeighborNode(2);
                        NearestNodeOrientation = NeighborNode(3);
                        FoundAFeasiblePath = 1;
                        NumberOfTries = 0;

                                while true


[PathPoints,Index,OptimalVelocity,OptimalSteeringAngle] =
BestPath(NearestNodeX,NearestNodeY,NearestNodeOrientation,RandomVariableX,Ran
domVariableY,RandomVariableO);

                                ObstacleFlag = 0;


                                    for m = 1 : Index
                                          if (
IsThereAnObstacle(PathPoints(m,1),PathPoints(m,2)) == 1 ||
CheckCollisionDetectionCorners(PathPoints(m,1),PathPoints(m,2),PathPoints(m,3
)) == 1 ||
CheckCollisionDetection(PathPoints(m,1),PathPoints(m,2),PathPoints(m,3)) == 1
)
                                             ObstacleFlag = 1;
                                             break ;
                                           end
                                    end

                                    if ObstacleFlag == 0
                                        break;
                                    end

                                NumberOfTries = NumberOfTries + 1 ;

                                    if NumberOfTries > 3
                                        FoundAFeasiblePath = 0;
                                        break;
                                    end


                                end

                        NewX = PathPoints(Index,1);
                        NewY = PathPoints(Index,2);
                        NewO = PathPoints(Index,3);

                        if FoundAFeasiblePath == 0
                            NumberOfNodesForWhichPathIsNotFound =
NumberOfNodesForWhichPathIsNotFound + 1;
                            continue;
                        end



                            NewNode = [NewX;NewY;NewO];
                            NewNodeID = PRMGraph.add_node(NewNode);

PRMGraph.highlight_node(NewNodeID,'NodeSize',4,'NodeFaceColor','cyan');
                            pause(0);


PRMGraph.add_edge(NewNodeID,NeighborNodeID(node)); %
,[OptimalVelocity;OptimalSteeringAngle]

PRMGraph.highlight_edge(NewNodeID,NeighborNodeID(node),'EdgeColor','blue','Ed
geThickness',0.2);
                            pause(0);
```

```
                end

                if(NumberOfNodesForWhichPathIsNotFound < MaxNodes)
                    break;
                end

         end

    end


    VertexClosestToGoal =
PRMGraph.closest([RobotLocationXGoal,RobotLocationYGoal,RobotOrientationGoal]
);
    VertexClosestToStartingPoint =
PRMGraph.closest([RobotLocationXInit,RobotLocationYInit,RobotOrientationInit]
);

    if PRMGraph.component(VertexClosestToStartingPoint) ~=
PRMGraph.component(VertexClosestToGoal)
        error(' Starting Node and Goal Node are not connected. Rerun the
planner');
    end


    PRMGraph.goal(VertexClosestToGoal);
    AStarPath = AStar(PRMGraph,VertexClosestToStartingPoint,
VertexClosestToGoal);
    PRMGraph.highlight_path(AStarPath, 'red', 2);

    hold off;

    figure;

    ShowMap;

    str = sprintf('Number Of Iterations = %d ',NumberOfSamplePoints);
    TimeElapsed =  sprintf('Time Elapsed = %f seconds ',toc);

    title({'Trajectory generated by KinoDynamic PRM'; str ;TimeElapsed});

    H3 =
DrawRobot(RobotLocationXInit,RobotLocationYInit,RobotOrientationInit,'cyan');

    pause(0);


    H4 =
DrawRobot(RobotLocationXGoal,RobotLocationYGoal,RobotOrientationGoal,'magenta
');

    pause(0);
    legend([H3 H4],{'Starting Position','Goal
Position'},'Location','eastoutside');


PRMGraph.highlight_node(VertexClosestToStartingPoint,'NodeSize',4,'NodeFaceCo
lor','blue');

PRMGraph.highlight_node(VertexClosestToGoal,'NodeSize',4,'NodeFaceColor','blu
e');

    AStarPathLength = length(AStarPath);

    for PathPointIndex = 1 :  AStarPathLength

        RobotLocation = PRMGraph.coord(AStarPath(PathPointIndex));

        RobotLocationX = RobotLocation(1);
        RobotLocationY = RobotLocation(2);
        RobotOrientation = RobotLocation(3);

        H5 =
DrawRobot(RobotLocationX,RobotLocationY,RobotOrientation,'yellow');

        pause(0.2);

        if PathPointIndex <  AStarPathLength
           % delete(H5);
        end




     end

    function H =
DrawRobot(RobotLocation_X,RobotLocation_Y,Robot_Orientation,Color)




        RobotLocationXCorner1 = RobotLocation_X + (-
15*cosd(Robot_Orientation)) - ( 5*sind(Robot_Orientation)) ;
        RobotLocationYCorner1 = RobotLocation_Y + (-
15*sind(Robot_Orientation)) + ( 5*cosd(Robot_Orientation)) ;

        RobotLocationXCorner2 = RobotLocation_X +
(15*cosd(Robot_Orientation)) - ( 5*sind(Robot_Orientation)) ;
        RobotLocationYCorner2 = RobotLocation_Y +
(15*sind(Robot_Orientation)) + ( 5*cosd(Robot_Orientation)) ;

        RobotLocationXCorner3 = RobotLocation_X +
(15*cosd(Robot_Orientation)) - ( -5*sind(Robot_Orientation)) ;
        RobotLocationYCorner3 = RobotLocation_Y +
(15*sind(Robot_Orientation)) + ( -5*cosd(Robot_Orientation)) ;
```

```
        RobotLocationXCorner4 = RobotLocation_X + (-
15*cosd(Robot_Orientation)) - ( -5*sind(Robot_Orientation)) ;
        RobotLocationYCorner4 = RobotLocation_Y + (-
15*sind(Robot_Orientation)) + ( -5*cosd(Robot_Orientation)) ;

        RobotXVertices = [RobotLocationXCorner1 RobotLocationXCorner2
RobotLocationXCorner3 RobotLocationXCorner4 ];
        RobotYVertices = [RobotLocationYCorner1 RobotLocationYCorner2
RobotLocationYCorner3 RobotLocationYCorner4 ];


        H = patch(RobotXVertices,RobotYVertices,Color);


    end

    function AStarPath = AStar(PRMGraph,VertexClosestToStartingPoint,
VertexClosestToGoal)


        ClosedSet = [];

        OpenSet = [VertexClosestToStartingPoint] ;

        CameFrom = [];

        GoalID = VertexClosestToGoal;

        G_Cost(VertexClosestToStartingPoint) = 0;
        DistanceBetweenStartAndGoal = sqrt( (RobotLocationXInit -
RobotLocationXGoal)^2 + (RobotLocationYInit-RobotLocationYGoal)^2 +
(deg2rad(RobotOrientationInit)-deg2rad(RobotOrientationGoal))^2 );
        H_Cost(VertexClosestToStartingPoint) = DistanceBetweenStartAndGoal;

        F_Cost(VertexClosestToStartingPoint) =
G_Cost(VertexClosestToStartingPoint) + H_Cost(VertexClosestToStartingPoint);

        while ~isempty(OpenSet)

            [DistanceValue,IndexOfTheNodeWithLowestFCost] =
min(F_Cost(OpenSet));
            CurrentNodeID = OpenSet(IndexOfTheNodeWithLowestFCost);

            if CurrentNodeID == GoalID
                AStarPath = [];
                TempID = GoalID;

                while true
                    AStarPath = [TempID AStarPath];
                    TempID = CameFrom(TempID);
                    if TempID == 0
                        break;
                    end
                end
                return;

            end

           %Remove Current Node From The Open Set

           OpenSet = setdiff(OpenSet,CurrentNodeID);

           %Add Current Node To The Closed Set

           ClosedSet = union(ClosedSet,CurrentNodeID);

                 for Neighbour = PRMGraph.neighbours(CurrentNodeID)
                         if ismember(Neighbour, ClosedSet)
                             continue;
                         end
                         Tentative_G_Cost = G_Cost(CurrentNodeID) + ...
                            PRMGraph.distance(CurrentNodeID,Neighbour);

                         if ~ismember(Neighbour, OpenSet)
                             %add neighbor to openset
                             OpenSet = union(OpenSet, Neighbour);
                             H_Cost(Neighbour) = PRMGraph.distance(Neighbour,
GoalID);
                             Tentative_Is_Better = true;
                         elseif Tentative_G_Cost < G_Cost(Neighbour)
                             Tentative_Is_Better = true;
                         else
                             Tentative_Is_Better = false;
                         end
                         if Tentative_Is_Better
                             CameFrom(Neighbour) = CurrentNodeID;
                             G_Cost(Neighbour) = Tentative_G_Cost;
                             F_Cost(Neighbour) = G_Cost(Neighbour) +
H_Cost(Neighbour);
                         end
                 end

        end

        AStarPath = [];

     end


    function [PathPoints,Index,OptimalVelocity,OptimalSteeringAngle] =
BestPath(NearestNodeX,NearestNodeY,NearestNodeOrientation,RandomVariableX,Ran
domVariableY,RandomVariableO)

        DistanceThreshold = Inf;

        for i = 1 : NumberOfTrials

            if rand >= 0.5
                Speed = Velocity;
            else
                Speed = -Velocity;
            end

            SteeringAngle = (2*rand - 1)* SteeringAngleMax;

            TempX = NearestNodeX;
            TempY = NearestNodeY;
            TempO = NearestNodeOrientation;
            PathHistory = [];

            for r = 1 : (SimulationTime/SamplingInterval)

                TempX = TempX + (Speed*SamplingInterval*cosd(TempO));
                TempY = TempY + (Speed*SamplingInterval*sind(TempO));
                TempO = rad2deg(deg2rad(TempO) +
(Speed*SamplingInterval/CarLength*deg2rad(SteeringAngle)));
                PathHistory = [PathHistory;[TempX TempY TempO]];

            end

            DistanceBetweenPathPointsFromRandomNumber =
Inf*ones(1,length(PathHistory));
            for k = 1:length(PathHistory)
              DistanceBetweenPathPointsFromRandomNumber(k) = sqrt(
(RandomVariableX - PathHistory(k,1))^2 + (RandomVariableY-PathHistory(k,2))^2
+ ((deg2rad(RandomVariableO) - deg2rad(PathHistory(k,3))))^2 );
            end

            [MinimumDistanceBetweenPathPointsFromRandomNumber, IndexNumber] =
min(DistanceBetweenPathPointsFromRandomNumber);

            if MinimumDistanceBetweenPathPointsFromRandomNumber <
DistanceThreshold

                DistanceThreshold =
MinimumDistanceBetweenPathPointsFromRandomNumber;
                PathPoints = PathHistory;
                Index = IndexNumber;
                OptimalVelocity = Speed;
                OptimalSteeringAngle = SteeringAngle;

            end

        end

    end

end
```

Display Map

```
function ShowMap

x = [0 0 10 10];
y = [0 300 300 0];
patch(x,y,'red')

x = [ 0 300 300 0];
y = [ 300 300 290 290];
patch(x,y,'red')

x = [ 290 300 300 290];
y = [ 300 300 0 0];
patch(x,y,'red')

x = [ 300 0 0 300];
y = [  0 0 10 10];
patch(x,y,'red')

x = [ 100 100 75 75];
y = [  100 75 75 100];
patch(x,y,'red')

x = [ 125 225 235 135];
y = [  175 175 165 165];
patch(x,y,'red')

end
```

Check Obstacles at a 2d Point

```
function ObstacleDetection = IsThereAnObstacle(xq,yq)


x1 = [0 0 10 10];
y1 = [0 300 300 0];

x2 = [ 0 300 300 0];
y2= [ 300 300 290 290];


x3 = [ 290 300 300 290];
y3 = [ 300 300 0 0];

x4 = [ 300 0 0 300];
y4 = [  0 0 10 10];


x5 = [ 100 100 75 75];
y5 = [  100 75 75 100];
```

```
x6 = [ 125 225 235 135];
y6 = [  175 175 165 165];

in1 = inpolygon(xq,yq,x1,y1);
in2 = inpolygon(xq,yq,x2,y2);
in3 = inpolygon(xq,yq,x3,y3);
in4 = inpolygon(xq,yq,x4,y4);
in5 = inpolygon(xq,yq,x5,y5);
in6 = inpolygon(xq,yq,x6,y6);


ObstacleDetection = 0;

if in1 == 1 || in2 == 1 ||in3 == 1 || in4 == 1 || in5 == 1 || in6 == 1
    ObstacleDetection = 1;
end

end
```

Check Collision Detection For All The Points Along The Boundaries Of The Car

```
function Collision = CheckCollisionDetection(xRand,yRand,oRand)

    XCorner1 = round(xRand + (-15*cosd(oRand)) - ( 5*sind(oRand))) ;
    YCorner1 = round(yRand + (-15*sind(oRand)) + ( 5*cosd(oRand))) ;

    XCorner2 = round(xRand + (15*cosd(oRand)) - ( 5*sind(oRand))) ;
    YCorner2 = round(yRand + (15*sind(oRand)) + ( 5*cosd(oRand))) ;

    XCorner3 = round(xRand + (15*cosd(oRand)) - ( -5*sind(oRand))) ;
    YCorner3 = round(yRand + (15*sind(oRand)) + ( -5*cosd(oRand))) ;

    XCorner4 = round(xRand + (-15*cosd(oRand)) - ( -5*sind(oRand))) ;
    YCorner4 = round(yRand + (-15*sind(oRand)) + ( -5*cosd(oRand))) ;


    Collision = 0;

     if( IsThereAnObstacle(XCorner1,YCorner1) == 1 ||
IsThereAnObstacle(XCorner2,YCorner2) == 1 ||
IsThereAnObstacle(XCorner3,YCorner3) == 1 ||
IsThereAnObstacle(XCorner4,YCorner4) == 1 )
        Collision = 1;
        return
     end

     if(LineObstacleChecker(XCorner1,YCorner1,XCorner2,YCorner2,0,0) == 1)
         Collision = 1;
        return
     end
     if(LineObstacleChecker(XCorner2,YCorner2,XCorner3,YCorner3,0,0) == 1)
         Collision = 1;
        return
     end

     if(LineObstacleChecker(XCorner3,YCorner3,XCorner4,YCorner4,0,0) == 1)
         Collision = 1;
        return
     end

     if(LineObstacleChecker(XCorner4,YCorner4,XCorner1,YCorner1,0,0) == 1)
         Collision = 1;
        return
     end

 end
```

Check Collision Detection At Corners Of The Car

```
function Collision = CheckCollisionDetectionCorners(xRand,yRand,oRand)

    XCorner1 = (xRand + (-15*cosd(oRand)) - ( 5*sind(oRand))) ;
    YCorner1 = (yRand + (-15*sind(oRand)) + ( 5*cosd(oRand))) ;

    XCorner2 = (xRand+ (15*cosd(oRand)) - ( 5*sind(oRand))) ;
    YCorner2 = (yRand + (15*sind(oRand)) + ( 5*cosd(oRand))) ;

    XCorner3 = (xRand + (15*cosd(oRand)) - ( -5*sind(oRand))) ;
    YCorner3 = (yRand + (15*sind(oRand)) + ( -5*cosd(oRand))) ;

    XCorner4 = (xRand + (-15*cosd(oRand)) - ( -5*sind(oRand))) ;
    YCorner4 = (yRand + (-15*sind(oRand)) + ( -5*cosd(oRand))) ;

    Collision = 0;

     if( IsThereAnObstacle(XCorner1,YCorner1) == 1 ||
IsThereAnObstacle(XCorner2,YCorner2) == 1 ||
IsThereAnObstacle(XCorner3,YCorner3) == 1 ||
IsThereAnObstacle(XCorner4,YCorner4) == 1 )
        Collision = 1;

     end

end
```

Check For Obstacles Along A Line

```
function Collision =
LineObstacleChecker(XCorner1,YCorner1,XCorner2,YCorner2,PathFlag,Orientation)

Collision = 0;
N = 5;
X = linspace(XCorner1,XCorner2,N);
Y = linspace(YCorner1,YCorner2,N);



for i = 1 : N
    if( IsThereAnObstacle(X(i),Y(i)) == 1 )
        Collision = 1;
        return;
    end
    if (PathFlag == 1)
        if(CheckCollisionDetection(X(i),Y(i),Orientation) == 1)
            Collision = 1;
            return;
        end
    end


end

end
```

PETER CORKE'S CODE USED IN THIS PROJECT

Graph Data Structure For PRM Planning

```
classdef PGraph < handle

    properties (SetAccess=private, GetAccess=private)
        vertexlist      % vertex coordinates, columnwise, vertex number is
the column number
        edgelist        % 2xNe matrix, each column is vertex index of edge
start and end
        edgelen         % length (cost) of this edge

        curLabel        % current label
        ncomponents     % number of components
        labels          % label of each vertex (1xN)
        labelset        % set of all labels (1xNc)

        goaldist        % distance from goal, after planning

        userdata        % per vertex data, cell array
        ndims           % number of coordinate dimensions, height of vertices
matrix
```

```
        verbose
        measure         % distance measure: 'Euclidean', 'SE2'
    end

    properties (Dependent)
        n               % number of nodes/vertices
        ne              % number of edges
        nc              % number of components
    end

    methods

        function g = PGraph(ndims, varargin)
        %PGraph.PGraph Graph class constructor
        %
        % G=PGraph(D, OPTIONS) is a graph object embedded in D dimensions.
        %
        % Options::
        %  'distance',M   Use the distance metric M for path planning which
is either
        %                 'Euclidean' (default) or 'SE2'.
        %  'verbose'      Specify verbose operation
        %
        % Notes::
        % - Number of dimensions is not limited to 2 or 3.
        % - The distance metric 'SE2' is the sum of the squares of the
difference
        %   in position and angle modulo 2pi.
        % - To use a different distance metric create a subclass of PGraph
and
        %   override the method distance_metric().

            if nargin < 1
                ndims = 2;  % planar by default
            end
            g.ndims = ndims;
            opt.distance = 'Euclidean';
            opt = tb_optparse(opt, varargin);

            g.clear();
            g.verbose = opt.verbose;
            g.measure = opt.distance;
            g.userdata = {};
        end

        function n = get.n(g)
        %Pgraph.n Number of vertices
        %
        % G.n is the number of vertices in the graph.
        %
        % See also PGraph.ne.
            n = numcols(g.vertexlist);
        end

        function ne = get.ne(g)
        %Pgraph.ne Number of edges
        %
        % G.ne is the number of edges in the graph.
        %
        % See also PGraph.n.
            ne = numcols(g.edgelist);
        end

        function ne = get.nc(g)
        %Pgraph.nc Number of components
        %
        % G.nc is the number of components in the graph.
        %
        % See also PGraph.component.

            ne = g.ncomponents;
        end

        function clear(g)
        %PGraph.clear Clear the graph
        %
        % G.clear() removes all vertices, edges and components.

            g.labelset = zeros(1, 0);
            g.labels = zeros(1, 0);
            g.edgelist = zeros(2, 0);
            g.edgelen = zeros(1, 0);
            g.vertexlist = zeros(g.ndims, 0);

            g.ncomponents = 0;
            g.curLabel = 0;
        end

        function v = add_node(g, coord, varargin)
        %PGraph.add_node Add a node
        %
        % V = G.add_node(X) adds a node/vertex with coordinate X (Dx1) and
        % returns the integer node id V.
        %
        % V = G.add_node(X, V2) as above but connected by a directed edge
from vertex V
        % to vertex V2 with cost equal to the distance between the vertices.
        %
        % V = G.add_node(X, V2, C) as above but the added edge has cost C.
        %
        % Notes::
        % - Distance is computed according to the metric specified in the
        %   constructor.
        %
        % See also PGraph.add_edge, PGraph.data, PGraph.getdata.

            if length(coord) ~= g.ndims
                error('coordinate length different to graph coordinate
dimensions');
            end

            % append the coordinate as a column in the vertex matrix
            g.vertexlist = [g.vertexlist coord(:)];
            v = numcols(g.vertexlist);
            g.labels(v) = g.newlabel();

            if g.verbose
                fprintf('add node (%d) = ', v);
                fprintf('%f ', coord);
                fprintf('\n');
            end

            % optionally add an edge
            if nargin > 2
                g.add_edge(v, varargin{:});
            end
        end

        function u = setdata(g, v, u)
        %PGraph.setdata Set user data for node
        %
        % G.setdata(V, U) sets the user data of vertex V to U which can be of 
any
        % type such as a number, struct, object or cell array.
        %
        % See also PGraph.data.

            g.userdata{v} = u;
        end

        function u = data(g, v)
        %PGraph.data Get user data for node
        %
        % U = G.data(V) gets the user data of vertex V which can be of any
        % type such as a number, struct, object or cell array.
        %
        % See also PGraph.setdata.
            u = g.userdata{v};
        end



        function add_edge(g, v1, v2, d)
        %PGraph.add_edge Add an edge
        %
        % E = G.add_edge(V1, V2) adds a directed edge from vertex id V1 to 
vertex id V2, and
        % returns the edge id E.  The edge cost is the distance between the 
vertices.
        %
        % E = G.add_edge(V1, V2, C) as above but the edge cost is C.
        %
        % Notes::
        % - Distance is computed according to the metric specified in the
        %   constructor.
        % - Graph connectivity is maintained by a labeling algorithm and this
        %   is updated every time an edge is added.
        %
        % See also PGraph.add_node, PGraph.edgedir.
            if g.verbose
                fprintf('add edge %d -> %d\n', v1, v2);
            end
            for vv=v2(:)'
                g.edgelist = [g.edgelist [v1; vv]];
                if (nargin < 4) || isempty(d)
                    d = g.distance(v1, vv);
                end
                g.edgelen = [g.edgelen d];
                if g.labels(vv) ~= g.labels(v1)
                    g.merge(g.labels(vv), g.labels(v1));
                end
            end
        end

        function c = component(g, v)
        %PGraph.component Graph component
        %
        % C = G.component(V) is the id of the graph component that contains
vertex
        % V.
            c = [];
            for vv=v
                tf = ismember(g.labelset, g.labels(vv));
                c = [c find(tf)];
            end
        end

        % which edges contain v
        %  elist = g.edges(v)
        function e = edges(g, v)
        %PGraph.edges Find edges given vertex
        %
        % E = G.edges(V) is a vector containing the id of all edges connected
to vertex id V.
        %
        % See also PGraph.edgedir.
            e = [find(g.edgelist(1,:) == v) find(g.edgelist(2,:) == v)];
        end



        function dir = edgedir(g, v1, v2)
        %PGraph.edgedir Find edge direction
        %
        % D = G.edgedir(V1, V2) is the direction of the edge from vertex id
V1
        % to vertex id V2.
        %
        % If we add an edge from vertex 3 to vertex 4
        %         g.add_edge(3, 4)
        % then
        %         g.edgedir(3, 4)
        % is positive, and
        %         g.edgedir(4, 3)
        % is negative.
        %
        % See also PGraph.add_node, PGraph.add_edge.
            n = g.edges(v1);
            if any(ismember( g.edgelist(2, n), v2))
                dir = 1;
            elseif any(ismember( g.edgelist(1, n), v2))
                dir = -1;
            else
                dir = 0;
            end
        end

        function v = vertices(g, e)
        %PGraph.vertices Find vertices given edge
        %
        % V = G.vertices(E) return the id of the vertices that define edge E.
            v = g.edgelist(:,e);
        end


        function [n,c] = neighbours(g, v)
        %PGraph.neighbours Neighbours of a vertex
        %
        % N = G.neighbours(V) is a vector of ids for all vertices which are
        % directly connected neighbours of vertex V.
        %
        % [N,C] = G.neighbours(V) as above but also returns a vector C whose
elements
        % are the edge costs of the paths corresponding to the vertex ids in
N.
            e = g.edges(v);
            n = g.edgelist(:,e);
            n = n(:)';
            n(n==v) = [];   % remove references to self
            if nargout > 1
                c = g.cost(e);
            end
        end




        function [n,c] = neighbours_d(g, v)
        %PGraph.neighbours_d Directed neighbours of a vertex
        %
        % N = G.neighbours_d(V) is a vector of ids for all vertices which are
        % directly connected neighbours of vertex V.  Elements are positive
        % if there is a link from V to the node, and negative if the link
        % is from the node to V.
        %
        % [N,C] = G.neighbours_d(V) as above but also returns a vector C
whose elements
        % are the edge costs of the paths corresponding to the vertex ids in
N.
            e = g.edges(v);
            n = [-g.edgelist(1,e) g.edgelist(2,e)];
            n(abs(n)==v) = [];   % remove references to self
            if nargout > 1
                c = g.cost(e);
            end
        end

        function d = cost(g, e)
```

```
%PGraph.cost Cost of edge
%
% C = G.cost(E) is the cost of edge id E.
    d = g.edgelen(e);
end

function d = setcost(g, e, c)
%PGraph.cost Set cost of edge
%
% G.setcost(E, C) set cost of edge id E to C.
    g.edgelen(e) = c;
end

function p = coord(g, v)
%PGraph.coord Coordinate of node
%
% X = G.coord(V) is the coordinate vector (Dx1) of vertex id V.

    p = g.vertexlist(:,v);
end



function c = connectivity(g)
%PGraph.connectivity Graph connectivity
%
% C = G.connectivity() is a vector (Nx1) with the number of edges per
% vertex.
%
% The average vertex connectivity is
%         mean(g.connectivity())
%
% and the minimum vertex connectivity is
%         min(g.connectivity())

    for k=1:g.n
        c(k) = length(g.edges(k));
    end
end

function plot(g, varargin)
%PGraph.plot Plot the graph
%
% G.plot(OPT) plots the graph in the current figure.  Nodes
% are shown as colored circles.
%
% Options::
%  'labels'               Display vertex id (default false)
%  'edges'                Display edges (default true)
%  'edgelabels'           Display edge id (default false)
%  'NodeSize',S           Size of vertex circle (default 8)
%  'NodeFaceColor',C      Node circle color (default blue)
%  'NodeEdgeColor',C      Node circle edge color (default blue)
%  'NodeLabelSize',S      Node label text sizer (default 16)
%  'NodeLabelColor',C     Node label text color (default blue)
%  'EdgeColor',C          Edge color (default black)
%  'EdgeLabelSize',S      Edge label text size (default black)
%  'EdgeLabelColor',C     Edge label text color (default black)
```

```
        %  'componentcolor'      Node color is a function of graph component

            colorlist = 'bgmyc';

            % show vertices
            holdon = ishold;
            hold on

            % parse options
            opt.componentcolor = false;
            opt.labels = false;
            opt.edges = true;
            opt.edgelabels = false;
            opt.NodeSize = 8;
            opt.NodeFaceColor = 'b';
            opt.NodeEdgeColor = 'b';
            opt.NodeLabelSize = 16;
            opt.NodeLabelColor = 'b';
            opt.EdgeColor = 'k';
            opt.EdgeLabelSize = 8;
            opt.EdgeLabelColor = 'k';

            [opt,args] = tb_optparse(opt, varargin);

            % set default color if none specified
            if ~isempty(args)
                mcolor = args{1};
            else
                mcolor = 'b';
            end

            % show the vertices as filled circles
            for i=1:g.n
                % for each node
                if opt.componentcolor
                    j = mod( g.component(i)-1, length(colorlist) ) + 1;
                    c = colorlist(j);
                else
                    c = mcolor;
                end
                args = {'LineStyle', 'None', ...
                    'Marker', 'o', ...
                    'MarkerFaceColor', opt.NodeFaceColor, ...
                    'MarkerSize', opt.NodeSize, ...
                    'MarkerEdgeColor', opt.NodeEdgeColor };
                if g.ndims == 3
                    plot3(g.vertexlist(1,i), g.vertexlist(2,i), 
g.vertexlist(3,i), args{:});
                else
                    plot(g.vertexlist(1,i), g.vertexlist(2,i), args{:});
                end
            end
            % show edges
            if opt.edges
                for e=g.edgelist
                    v1 = g.vertexlist(:,e(1));
                    v2 = g.vertexlist(:,e(2));
                    if g.ndims == 3
                        plot3([v1(1) v2(1)], [v1(2) v2(2)], [v1(3) v2(3)],
...
                            'Color', opt.EdgeColor);
                    else
                        plot([v1(1) v2(1)], [v1(2) v2(2)], ...
                            'Color', opt.EdgeColor);
                    end
                end
            end
            % show the edge labels
            if opt.edgelabels
                for i=1:numcols(g.edgelist)
                    e = g.edgelist(:,i);
                    v1 = g.vertexlist(:,e(1));
                    v2 = g.vertexlist(:,e(2));

                    text('String', sprintf('  %g', g.cost(i)), ...
                        'Position', (v1 + v2)/2, ...
                        'HorizontalAlignment', 'left', ...
                        'VerticalAlignment', 'middle', ...
                        'FontUnits', 'pixels', ...
                        'FontSize', opt.EdgeLabelSize, ...
                        'Color', opt.EdgeLabelColor);
                end
            end
            % show the labels
            if opt.labels
                for i=1:numcols(g.vertexlist)
                    text('String', sprintf('  %d', i), ...
                        'Position', g.vertexlist(:,i), ...
                        'HorizontalAlignment', 'left', ...
                        'VerticalAlignment', 'middle', ...
                        'FontUnits', 'pixels', ...
                        'FontSize', opt.NodeLabelSize, ...
                        'Color', opt.NodeLabelColor);
                end
            end
            if ~holdon
                hold off
            end
        end

        function v = pick(g)
        %PGraph.pick Graphically select a vertex
        %
        % V = G.pick() is the id of the vertex closest to the point clicked
        % by the user on a plot of the graph.
        %
        % See also PGraph.plot.
            [x,y] = ginput(1);
            v = g.closest([x; y]);
        end

        function goal(g, vg)
        %PGraph.goal Set goal node
        %
```

```
        % G.goal(VG) computes the cost of reaching every vertex in the graph
connected
        % to the goal vertex VG.
        %
        % Notes::
        % - Combined with G.path performs a breadth-first search for paths to
the goal.
        %
        % See also PGraph.path, PGraph.Astar, Astar.

            % cost is total distance from goal
            g.goaldist = Inf*ones(1, numcols(g.vertexlist));

            g.goaldist(vg) = 0;
            g.descend(vg);
        end



        function p = path(g, v)
        %PGraph.path Find path to goal node
        %
        % P = G.path(VS) is a vector of vertex ids that form a path from
        % the starting vertex VS to the previously specified goal.  The path
        % includes the start and goal vertex id.
        %
        % To compute path to goal vertex 5
        %        g.goal(5);
        % then the path, starting from vertex 1 is
        %        p1 = g.path(1);
        % and the path starting from vertex 2 is
        %        p2 = g.path(2);
        %
        % Notes::
        % - Pgraph.goal must have been invoked first.
        % - Can be used repeatedly to find paths from different starting
points
        %   to the goal specified to Pgraph.goal().
        %
        % See also PGraph.goal, PGraph.Astar.
            p = [v];

            while g.goaldist(v) ~= 0
                v = g.next(v);
                p = [p v];
            end
        end



        function d = distance(g, v1, v2)
        %PGraph.distance Distance between vertices
        %
        % D = G.distance(V1, V2) is the geometric distance between
        % the vertices V1 and V2.
        %
        % See also PGraph.distances.

            d = g.distance_metric( g.vertexlist(:,v1), g.vertexlist(:,v2));
        end

        function [d,k] = distances(g, p)
        %PGraph.distances Distances from point to vertices
        %
        % D = G.distances(X) is a vector (1xN) of geometric distance from the
point
        % X (Dx1) to every other vertex sorted into increasing order.
        %
        % [D,W] = G.distances(P) as above but also returns W (1xN) with the
        % corresponding vertex id.
        %
        % Notes::
        % - Distance is computed according to the metric specified in the
        %   constructor.
        %
        % See also PGraph.closest.

            d = g.distance_metric(p(:), g.vertexlist);
            [d,k] = sort(d, 'ascend');
        end

        function [c,dn] = closest(g, p)
        %PGraph.closest Find closest vertex
        %
        % V = G.closest(X) is the vertex geometrically closest to coordinate
X.
        %
        % [V,D] = G.closest(X) as above but also returns the distance D.
        %
        % See also PGraph.distances.
            d = g.distance_metric(p(:), g.vertexlist);
            [mn,c] = min(d);

            if nargin > 1
                dn = mn;
            end
        end

        function display(g)
        %PGraph.display Display graph
        %
        % G.display() displays a compact human readable representation of the
        % state of the graph including the number of vertices, edges and
components.
        %
        % See also PGraph.char.
            loose = strcmp( get(0, 'FormatSpacing'), 'loose');
            if loose
                disp(' ');
            end
            disp([inputname(1), ' = '])
            disp( char(g) );
        end % display()

        function s = char(g)
        %PGraph.char Convert graph to string
        %
        % S = G.char() is a compact human readable representation of the
        % state of the graph including the number of vertices, edges and
components.

            s = '';
            s = strvcat(s, sprintf('  %d dimensions', g.ndims));
            s = strvcat(s, sprintf('  %d vertices', g.n));
            s = strvcat(s, sprintf('  %d edges', numcols(g.edgelist)));
            s = strvcat(s, sprintf('  %d components', g.ncomponents));
        end

        %% convert graphs to matrix representations

        function L = laplacian(g)
        %Pgraph.laplacian Laplacian matrix of graph
        %
        % L = G.laplacian() is the Laplacian matrix (NxN) of the graph.
        %
        % Notes::
        % - L is always positive-semidefinite.
        % - L has at least one zero eigenvalue.
        % - The number of zero eigenvalues is the number of connected
components
        %   in the graph.
        %
        % See also PGraph.adjacency, PGraph.incidence, PGraph.degree.

            L = g.degree() - (g.adjacency() > 0);
        end

        function D = degree(g)
        %Pgraph.degree Degree matrix of graph
        %
        % D = G.degree() is a diagonal matrix (NxN) where element D(i,i) is
the number
        % of edges connected to vertex id i.
        %
        % See also PGraph.adjacency, PGraph.incidence, PGraph.laplacian.

            D = diag( g.connectivity() );
        end

        function A = adjacency(g)
        %Pgraph.adjacency Adjacency matrix of graph
        %
        % A = G.adjacency() is a matrix (NxN) where element A(i,j) is the
cost
        % of moving from vertex i to vertex j.
        %
        % Notes::
        % - Matrix is symmetric.
        % - Eigenvalues of A are real and are known as the spectrum of the
graph.
        % - The element A(I,J) can be considered the number of walks of one
        %   edge from vertex I to vertex J (either zero or one).  The element
(I,J)
        %   of A^N are the number of walks of length N from vertex I to
vertex J.
        %
        % See also PGraph.degree, PGraph.incidence, PGraph.laplacian.

            A = zeros(g.n, g.n);
            for i=1:g.n
                [n,c] = g.neighbours(i);
                for j=1:numel(n)
                    A(i,n(j)) = c(j);
                    A(n(j),i) = c(j);
                end
            end
        end

        function I = incidence(g)
        %Pgraph.degree Incidence matrix of graph
        %
        % IN = G.incidence() is a matrix (NxNE) where element IN(i,j) is
        % non-zero if vertex id i is connected to edge id j.
        %
        % See also PGraph.adjacency, PGraph.degree, PGraph.laplacian.
            I = zeros(g.n, numcols(g.edgelist));
            for i=1:g.n
                for n=g.edges(i)
                    I(i,n) = 1;
                end
            end
        end

        %% these are problematic, dont advertise them
        %
        % removing an edge may divide the graph into 2 components, this is
expensive
        % to check and currently not implemented
        function delete_edge(g, e)
            g.edgelist(:,e) = [];
            % really need to check if the two halves are connected, is
expensive
            % could use path planner
        end

        function delete_node(g, v)
            el = g.edges(v);
            el
            % make the column invalid, really should remove it but this
            % requires changing all the edgelist entries, and the vertex
            % numbers will change...
            g.vertexlist(:,v) = [NaN; NaN];
            g.delete_edge(el);
            g.n = g.n - 1;
        end



        function highlight_node(g, verts, varargin)
        %PGraph.highlight_node Highlight a node
        %
        % G.highlight_node(V, OPTIONS) highlights the vertex V with a yellow
marker.
        % If V is a list of vertices then all are highlighted.
        %
        % Options::
        %  'NodeSize',S          Size of vertex circle (default 12)
        %  'NodeFaceColor',C     Node circle color (default yellow)
        %  'NodeEdgeColor',C     Node circle edge color (default blue)
        %
        % See also PGraph.highlight_edge, PGraph.highlight_path,
PGraph.highlight_component.

            hold on

            % parse options
            opt.NodeSize = 12;
            opt.NodeFaceColor = 'y';
            opt.NodeEdgeColor = 'b';

            [opt,args] = tb_optparse(opt, varargin);
            markerprops = {'LineStyle', 'None', ...
                'Marker', 'o', ...
                'MarkerFaceColor', opt.NodeFaceColor, ...
                'MarkerSize', opt.NodeSize, ...
                'MarkerEdgeColor', opt.NodeEdgeColor };

            for v=verts
                if g.ndims == 3
                    plot3(g.vertexlist(1,v), g.vertexlist(2,v),
g.vertexlist(3,v), ...
                        markerprops{:});
                else
                    plot(g.vertexlist(1,v), g.vertexlist(2,v),
markerprops{:});
                end
            end
        end

        function highlight_component(g, c, varargin)
        %PGraph.highlight_component Highlight a graph component
        %
        % G.highlight_component(C, OPTIONS) highlights the vertices that
belong to
        % graph component C.
        %
        % Options::
        %  'NodeSize',S          Size of vertex circle (default 12)
        %  'NodeFaceColor',C     Node circle color (default yellow)
        %  'NodeEdgeColor',C     Node circle edge color (default blue)
        %
        % See also PGraph.highlight_node, PGraph.highlight_edge,
PGraph.highlight_component.
            nodes = find(g.labels == g.labelset(c));
            for v=nodes
                g.highlight_node(v, varargin{:});
            end
        end

        function highlight_edge(g, e, varargin)
        %PGraph.highlight_node Highlight a node
        %
        % G.highlight_edge(V1, V2) highlights the edge between vertices V1
and V2.
        %
        % G.highlight_edge(E) highlights the edge with id E.
        %
        % Options::
        % 'EdgeColor',C          Edge edge color (default black)
        % 'EdgeThickness',T      Edge thickness (default 1.5)
        %
        % See also PGraph.highlight_node, PGraph.highlight_path,
PGraph.highlight_component.

            % parse options
            opt.EdgeColor = 'k';
            opt.EdgeThickness = 1.5;

            [opt,args] = tb_optparse(opt, varargin);

            hold on
            if (length(args) > 0) && isnumeric(args{1})
                % highlight_edge(V1, V2)
                v1 = e;
                v2 = args{1};

                v1 = g.vertexlist(:,v1);
                v2 = g.vertexlist(:,v2);
             else
                % highlight_edge(E)
                e = g.edgelist(:,e);
                v1 = g.vertexlist(:,e(1));
                v2 = g.vertexlist(:,e(2));
            end

            % create the line properties for the edges
            lineprops = {
                'Color', opt.EdgeColor, ...
                'LineWidth', opt.EdgeThickness };

            if g.ndims == 3
                plot3([v1(1) v2(1)], [v1(2) v2(2)], [v1(3) v2(3)],
lineprops{:});
            else
                plot([v1(1) v2(1)], [v1(2) v2(2)], lineprops{:});
            end
        end

        function highlight_path(g, path, edgecolor, edgethickness)
        %PGraph.highlight_path Highlight path
        %
        % G.highlight_path(P, OPTIONS) highlights the path defined by vector
P
```

```
        % which is a list of vertex ids comprising the path.
        %
        % Options::
        %  'NodeSize',S          Size of vertex circle (default 12)
        %  'NodeFaceColor',C     Node circle color (default yellow)
        %  'NodeEdgeColor',C     Node circle edge color (default blue)
        %  'EdgeColor',C         Node circle edge color (default black)
        %
        % See also PGraph.highlight_node, PGraph.highlight_edge,
PGraph.highlight_component.
            g.highlight_node(path);

            % highlight the edges
            for i=1:numel(path)-1
                v1 = path(i);
                v2 = path(i+1);
                g.highlight_edge(v1, v2, 'EdgeColor', edgecolor,
'EdgeThickness', edgethickness );
            end
        end



    end % method

    methods (Access='protected')
    % private methods

        % depth first
        function descend(g, vg)

            % get neighbours and their distance
            for nc = g.neighbours2(vg);
                vn = nc(1);
                d = nc(2);
                newcost = g.goaldist(vg) + d;
                if isinf(g.goaldist(vn))
                    % no cost yet assigned, give it this one
                    g.goaldist(vn) = newcost;
                    %fprintf('1: cost %d <- %f\n', vn, newcost);
                    descend(g, vn);
                else
                    % it already has a cost
                    if g.goaldist(vn) <= newcost
                        continue;
                    else
                        g.goaldist(vn) = newcost;
                        %fprintf('2: cost %d <- %f\n', vn, newcost);
                        descend(g, vn);
                    end
                end
            end
        end

        % breadth first
        function descend2(g, vg)

            % get neighbours and their distance
            for vn = g.neighbours2(vg);
                vn = nc(1);
                d = nc(2);
                newcost = g.goaldist(vg) + d;
                if isinf(g.goaldist(vn))
                    % no cost yet assigned, give it this one
                    g.goaldist(vn) = newcost;
                    fprintf('1: cost %d <- %f\n', vn, newcost);
                    descend(g, vn);
                elseif g.goaldist(vn) > newcost
                    % it already has a cost
                        g.goaldist(vn) = newcost;
                end
            end
            for vn = g.neighbours(vg);
                descend(g, vn);
            end
        end

        function l = newlabel(g)
            g.curLabel = g.curLabel + 1;
            l = g.curLabel;
            g.ncomponents = g.ncomponents + 1;
            g.labelset = union(g.labelset, l);
        end

        function merge(g, l1, l2)

            % merge label1 and label2, lowest label dominates

            % get the dominant and submissive labels
            ldom = min(l1, l2);
            lsub = max(l1, l2);

            % change all instances of submissive label to dominant one
            g.labels(g.labels==lsub) = ldom;

            % reduce the number of components
            g.ncomponents = g.ncomponents - 1;
            % and remove the submissive label from the set of all labels
            g.labelset = setdiff(g.labelset, lsub);
        end

        function nc = neighbours2(g, v)
            e = g.edges(v);
            n = g.edgelist(:,e);
            n = n(:)';
            n(n==v) = [];   % remove references to self
            c = g.cost(e);
            nc = [n; c];
        end

        function d = distance_metric(g, x1, x2)

            % distance between coordinates x1 and x2 using the relevant
metric
```

```
            % x2 can be multiple points represented by multiple columns
            switch g.measure
                case 'Euclidean'
                   d = colnorm( bsxfun(@minus, x1, x2) );

                case 'SE2'
                    d = bsxfun(@minus, x1, x2);
                    d(3,:) = angdiff(d(3,:));
                    d = colnorm( d );
                otherwise
                    error('unknown distance measure', g.measure);
            end
        end

        function vn = next(g, v)

            % V = G.next(VS) return the id of a node connected to node id VS
            % that is closer to the goal.
            n = g.neighbours(v);
            [mn,k] = min( g.goaldist(n) );
            vn = n(k);
        end

    end % private methods
end % classdef
```

Number of Columns in a Matrix

```
function c = numcols(m)
    c = size(m,2);
```

Number of Rows in a Matrix

```
function r = numrows(m)

    r = size(m, 1);
```

Column Wise Norm of a Matrix

```
function n = colnorm(a)

    n = sqrt(sum(a.^2));
```

Line Generator

```
function p = bresenham(x1, y1, x2, y2)

    if nargin == 2
    p1 = x1; p2 = y1;

    x1 = p1(1); y1 = p1(2);
    x2 = p2(1); y2 = p2(2);
elseif nargin ~= 4
    error('expecting 2 or 4 arguments');
end

x = x1;
if x2 > x1
    xd = x2-x1;
    dx = 1;
else
    xd = x1-x2;
    dx = -1;
end

y = y1;
if y2 > y1
    yd = y2-y1;
    dy = 1;
else
    yd = y1-y2;
    dy = -1;
end

p = [];

if xd > yd
  a = 2*yd;
  b = a - xd;
  c = b - xd;

  while 1
    p = [p; x y];
    if all([x-x2 y-y2] == 0)
        break
    end
    if  b < 0
        b = b+a;
        x = x+dx;
    else
        b = b+c;
        x = x+dx; y = y+dy;
    end
  end
else
  a = 2*xd;
  b = a - yd;
  c = b - yd;

  while 1
    p = [p; x y];
    if all([x-x2 y-y2] == 0)
        break
    end
    if  b < 0
```

```
            b = b+a;
            y = y+dy;
        else
            b = b+c;
            x = x+dx; y = y+dy;
        end
      end
    end
end
```

Option Parser

```
function [opt,others,ls] = tb_optparse(in, argv)

    if nargin == 1
        argv = {};
    end

    if ~iscell(argv)
        error('RTB:tboptparse:badargs', 'input must be a cell array');
    end

    arglist = {};

    argc = 1;
    opt = in;

    if ~isfield(opt, 'verbose')
        opt.verbose = false;
    end
    if ~isfield(opt, 'debug')
        opt.debug = 0;
    end

    showopt = false;
    choices = [];

    while argc <= length(argv)
        % index over every passed option
        option = argv{argc};
        assigned = false;

        if isstr(option)

            switch option
            % look for hardwired options
            case 'verbose'
                opt.verbose = true;
                assigned = true;
            case 'verbose=2'
                opt.verbose = 2;
                assigned = true;
            case 'verbose=3'
```

```
                opt.verbose = 3;
                assigned = true;
            case 'verbose=4'
                opt.verbose = 4;
                assigned = true;
            case 'debug'
                opt.debug = argv{argc+1};
                argc = argc+1;
                assigned = true;
            case 'setopt'
                new = argv{argc+1};
                argc = argc+1;
                assigned = true;

                % copy matching field names from new opt struct to current
one
                for f=fieldnames(new)'
                    if isfield(opt, f{1})
                        opt.(f{1}) = new.(f{1});
                    end
                end
            case 'showopt'
                showopt = true;
                assigned = true;

            otherwise
                % does the option match a field in the opt structure?
%                 if isfield(opt, option) || isfield(opt, ['d_' option])
%                if any(strcmp(fieldnames(opt),option)) ||
any(strcmp(fieldnames(opt),))
                 if isfield(opt, option) || isfield(opt, ['d_' option]) ||
isprop(opt, option)

                    % handle special case if we we have opt.d_3d, this
                    % means we are looking for an option '3d'
                    if isfield(opt, ['d_' option]) || isprop(opt, ['d_'
option])
                        option = ['d_' option];
                    end

                    %** BOOLEAN OPTION
                    val = opt.(option);
                    if islogical(val)
                        % a logical variable can only be set by an option
                        opt.(option) = true;
                    else
                        %** OPTION IS ASSIGNED VALUE FROM NEXT ARG
                        % otherwise grab its value from the next arg
                        try
                            opt.(option) = argv{argc+1};
                        catch me
                            if strcmp(me.identifier, 'MATLAB:badsubscript')
                                error('RTB:tboptparse:badargs', 'too few
arguments provided for option: [%s]', option);
                            else
                                rethrow(me);
                            end
                        end
                        argc = argc+1;
                    end
                    assigned = true;
                elseif length(option)>2 && strcmp(option(1:2), 'no') &&
isfield(opt, option(3:end))
                    %* BOOLEAN OPTION PREFIXED BY 'no'
                    val = opt.(option(3:end));
                    if islogical(val)
                        % a logical variable can only be set by an option
                        opt.(option(3:end)) = false;
                        assigned = true;
                    end
                else
                    % the option doesn't match a field name
                    % let's assume it's a choice type
                    %     opt.choose = {'this', 'that', 'other'};
                    %
                    % we need to loop over all the passed options and look
                    % for those with a cell array value
                    for field=fieldnames(opt)'
                        val = opt.(field{1});
                        if iscell(val)
                            for i=1:length(val)
                                if isempty(val{i})
                                    continue;
                                end
                                % if we find a match, put the final value
                                % in the temporary structure choices
                                %
                                % eg. choices.choose = 'that'
                                %
                                % so that we can process input of the form
                                %
                                %  'this', 'that', 'other'
                                %
                                % which should result in the value 'other'
                                if strcmp(option, val{i})
                                    choices.(field{1}) = option;
                                    assigned = true;
                                    break;
                                elseif val{i}(1) == '#' && strcmp(option,
val{i}(2:end))

                                    choices.(field{1}) = i;
                                    assigned = true;
                                    break;
                                end
                            end
                            if assigned
                                break;
                            end
                        end
                    end
                end
            end % switch
        end
        if ~assigned
            % non matching options are collected
            if nargout >= 2
                arglist = [arglist argv(argc)];
            else
                if isstr(argv{argc})
                    error(['unknown options: ' argv{argc}]);
                end
            end
        end

        argc = argc + 1;
    end % while

    % copy choices into the opt structure
    if ~isempty(choices)
        for field=fieldnames(choices)'
           opt.(field{1}) = choices.(field{1});
        end
    end

    % if enumerator value not assigned, set the default value
    if ~isempty(in)
        for field=fieldnames(in)'
            if iscell(in.(field{1})) && iscell(opt.(field{1}))
                val = opt.(field{1});
                if isempty(val{1})
                    opt.(field{1}) = val{1};
                elseif val{1}(1) == '#'
                    opt.(field{1}) = 1;
                else
                    opt.(field{1}) = val{1};
                end
            end
        end
    end

    if showopt
        fprintf('Options:\n');
        opt
        arglist
    end

    if nargout == 3
        % check to see if there is a valid linespec floating about in the
        % unused arguments
        for i=1:length(arglist)
            s = arglist{i};
            % get color
            [b,e] = regexp(s, '[rgbcmywk]');
            s2 = s(b:e);
            s(b:e) = [];

            % get line style
            [b,e] = regexp(s, '(--)|(-.)|-|:');
            s2 = [s2 s(b:e)];
            s(b:e) = [];

            % get marker style
            [b,e] = regexp(s, '[o\+\*\.xsd\^v><ph]');
            s2 = [s2 s(b:e)];
            s(b:e) = [];

            % found one
            if isempty(s)
                ls = arglist{i};
                arglist(i) = [];
                others = arglist;
                break;
            end
        end
        ls = [];
        others = arglist;
    elseif nargout == 2
        others = arglist;
    end
```