Version 31.05.2021

Reading Instructions

Please carefully study the sections 5.1, 5.2 and 5.4 in the chapter on motion planning in the Springer Handbook of Robotics [1]. Carefully read the papers on probabilistic roadmap planner [2] and randomized kinodynamic planning (rapidly evolving random trees) [3].

This assignment utilizes two maps (simple and complex) provided in the assignment folder in Moodle in `maps.zip`. Download the map image file `turtlebot3world.pgm` contained in the `maps.zip` file from the Moodle workspace to your local folder in order to test your path planner within RViz and Stage in the Turtlebot3 World environment.

Path Planning

Path planning seeks a sequence of collision-free motions of a mobile robot from a start to a goal pose among a configuration of static obstacles.

Path planning assumes a complete geometric description of the robots footprint and the environment. In the case of mobile robots the workspace is a 2D environment populated with planar obstacles represented by a map. The objective is to find a collision-free path through the environment that connects the start and goal.

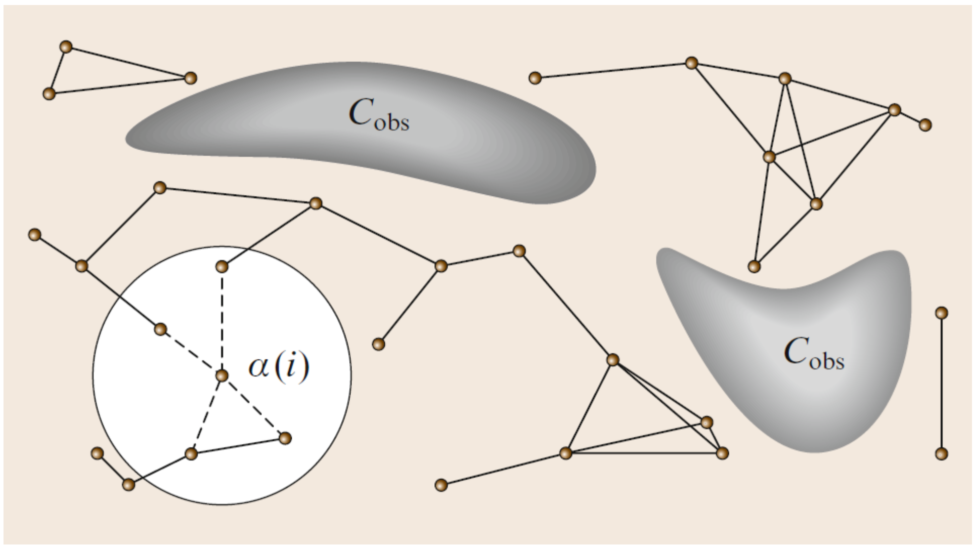

Figure 1: Sampling based road map planner [1]

Probabilistic Road Map Planner (PRM)

The probabilistic road map planner (PRM) belongs to the class of multiple query planners. In the preprocessing phase, the road map planner samples random poses in the free

[1]source: Springer Handbook of Robotics, Motion Planning [1]

space and establishes their connectivity. The roadmap is constituted by a graph G, with vertices corresponding to feasible poses and edges denote straight line paths connecting the vertices.

The roadmap G is constructed in the following manner as shown in figure 1:

- Initialization: $G(V, E)$ represents an undirected graph, which is initially empty. Vertices of G correspond to possible collision-free robot poses, and edges to collision-free paths that connect vertices.
- Pose sampling: A pose $\alpha(i)$ is sampled from the free space C_{free} and added to the vertex set V.
- Neighborhood: Based on the Euclidean distance $d(q, \alpha(i)$ or some other metric, those vertices q that are already part of V belong to $\alpha(i)$s neighborhood if they are located within a radius of $alpha(i)$ according to $d(q, \alpha(i)$.
- Edge selection: For those vertices q that do not belong in the same connected component of G with $\alpha(i)$ the algorithm attempts to connect them with an edge.
- Local path planning: Given $\alpha(i)$ and $q \in C_{free}$ attempt to construct a local path $\tau_s : [0, 1] \rightarrow C_{free}$ such that $\tau(0) = \alpha(i)$ and $\tau(1) = q$. Ensure that the path is collision free. A simple scheme merely considers a straight line path between $\alpha(i)$ and q.
- Edge insertion: Insert τ_s into E, as an edge from $\alpha(i)$ to q.
- Termination: The algorithm terminates once the roadmap is densely populated with a predefined number N of collision-free poses.

Algorithm 1 details the pseudocode for the sampling based roadmap. The original roadmap planner samples the poses $\alpha(i)$ randomly. The connection step checks whether the straight line path between q and $\alpha(i)$ belongs to the C-space. Modifications to the original approach sample poses at or near the boundary of the free space, move samples as far from the border as possible or employ deterministic grid-based sampling. The roadmap planner has difficulties in identifying narrow passages.

Given the roadmap G, a collision free path is planned from start q_I to goal pose q_G. For that purpose the planner adds q_I and q_G as vertices to the roadmap and performs a graph search for a sequence of edges that connect q_I and q_G.

1) Load map files from `exampleMaps.mat` contained in `maps.zip`. The file contains a simple and a more complex map represented by a binary occupancy grid (logical (binary) matrix) in which an entry of 1 indicates an occupied and 0 an unoccupied cell in the environment.

Algorithm 1 Sampling Based Roadmap

Input: N: number of vertices to include in the roadmap, C_{free} configuration space, $G(V, E)$ roadmap (graph with vertices V and edges E)

```
1: G.init(); $i \leftarrow 0$;
2: while $i < N$ do
3:     if $\alpha(i) \in C_{free}$ then
4:         G.add_vertex ($\alpha(i)$) ; $i \leftarrow i + 1$;
5:         for $q \in$ NEIGHBORHOOD($\alpha(i)$,G) do
6:             if CONNECT($\alpha(i)$,q) then
7:                 G.add_edge ($\alpha(i), q$)
```

2) The method `binaryOccupancyMap` generates a 2-D occupancy grid object, which partitions the environment into free space and obstacles. The occupancy grid is constructed either from map data or sensor data in conjunction with robot pose estimates.

   ```
   map = binaryOccupancyMap(p,res)
   ```

   creates a grid from the binary entries in matrix `p` at a resolution `res` specified in cells per meter. Notice, that this differs from the definition in YAML-files which define the resolution w.r.t. meter / pixel (cell). The resolution and the dimension of the binary matrix `p` determine the height and width of the occupancy grid. Generate the occupancy grids from the map data with a resolution of $2\frac{\text{cells}}{\text{m}}$. Visualize both occupancy grids with `show`.

3) Generate a probabilistic roadmap for the simple map with 50 nodes. `mobileRobotPRM(map,numnodes)` creates a roadmap from the binary occupancy grid `map`, with the maximum number of nodes `numnodes`. Visualize the road map with `show`.

4) Generate a denser probabilistic roadmap for the larger and more complex map with 500 nodes. Visualize the road map with `show`.

5) Plan a collision-free path from start location $\mathbf{x}_s = [2, 1]$ to goal location $\mathbf{x}_g = [12, 10]$ for the simple map using `findpath` and visualize the roadmap and path with `show`.

6) Plan a collision-free path from start location $\mathbf{x}_s = [2, 1]$ to goal location $\mathbf{x}_g = [22, 18]$ for the complex map. Visualize the roadmap and path.

7) Load the TurtleBot3 map from the image file `turtlebot3world.pgm` using

   ```
   turtlebot3world = ~logical(imread('maps/turtlebot3world.pgm'));
   ```

   Generate the corresponding binary occupancy grid. Notice that a resolution of 20 cells per meter corresponds to 0.05 meter per pixel in PGM file. Reduce the map to the relevant area. Also consider to adapt the origin of the coordinate system in the binary occupancy grid properly. Use the property `GridLocationInWorld` of the `binaryOccupancyMap` object.

8) Start the Turtlebot3 Gazebo simulation by running the command

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

in a Linux terminal.

9) Plan and visualize a path that connects the robot's current location with the goal location $\mathbf{x}_g = [0, 2]$ with the probabilistic road map planner in Matlab. For this, query from topic `odom` and use the helper function `OdometryMsg2Pose` to obtain the current robot pose.

10) Launch RViz by running the command

```
roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
```

in a Linux terminal.

11) Query the goal pose from the topic `/move_base_simple/goal` rather than using a fixed goal pose. Use the helper function `PoseStampedMsg2Pose` to convert the ROS message. Plan and visualize the path for the new custom goal location.

12) Notice that currently the planned path does not consider the dimensions of the circular robot and that some vertices of the roadmap are close to obstacles. To plan a safe and collision-free path inflate the occupancy grid by the robot radius. The function `inflate(map,radius)` enlarges occupied map cells by the specified radius. The status of free cells separated to an occupied cell by less than the radius changes to occupied (true). Inflate the world occupancy grid by the robot radius $r = 0.18\,\text{m}$ plus a safety margin $d = 0.12\,\text{m}$. Regenerate the roadmap for the world environment and replan the path.

Rapidly Exploring Random Trees (RRT)

Single-query planning methods focus on a single initial goal configuration pair. They explore the free space by expanding tree data structures which roots are initialized at known configurations and eventually connecting them. These methods proceed in the following manner:

- Initialization: Let $G(V, E)$ represent an undirected search graph, for which the vertex set V contains a vertex for the free goal location x_g, and the edge set E is empty. Vertices of G are collision-free configurations, and edges are collision-free paths that connect vertices. Notice that the expansion starts from the goal node and proceeds towards the start node. That way you obtain a global plan that can be utilized from other start poses.

- Vertex selection method: Choose a randomized (possibly biased) pose x_{rand}. Determine the vertex $x_{near} \in V$ that is closed to x_{rand} w.r.t. to some metric for expansion.
- Local planning method: For some $x_{new} \in C_{free}$ which may correspond to an existing vertex in V but on a different tree or a sampled configuration, attempt to construct a path $\tau_s : [0, 1] \to C_{free}$ such that $\tau(0) = x_{near}$ and $\tau(1) = x_{new}$. Collision detection ensures that τ_s is a collision free path. If this step fails to produce a collision-free path segment, then go to vertex selection.
- Insert an edge in the graph: Insert τ_s into E, as an edge from x_{near} to x_{new}. If x_{new} is not already in V, then it is inserted.
- Check for a solution: Determine whether G encodes a solution path.
- Return to vertex selection: Iterate unless a viable solution is found or the algorithm terminates reporting failure to identify a viable path.

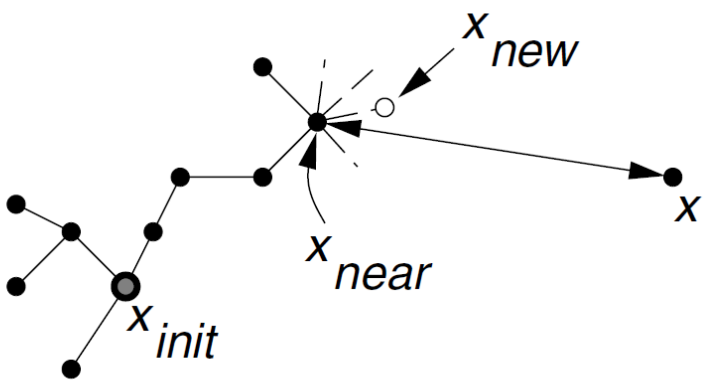

Figure 2: Extend operation of the RRT planner [2]

Rapidly exploring random tree (RRT) is an example of such an expanding single query planner. It induces a Voronoi bias in the exploration process by selecting for expansion the node in the tree that is closest to x_{rand} in each iteration. The tree construction is described in algorithms 2 and 3.

Figure 2 illustrates the extend operation of the RRT planner. x denotes the random state towards which the graph is supposed to expand, x_{near} the vertex closest to x and x_{new} a feasible pose resulting from local planning that is reachable from x_{near} with the admissible control u_{new}.

Matlab provides an implementation of the rapidly-exploring random tree (RRT) algorithm to plan a path for a vehicle through a known map. The state space defines the kinematic constraints of the vehicle for which RRT plans a feasible path. RRT requires an `occupancyMap` and a start and goal pose.

[2]source: Steven M. La Valle, James J. Kuffner, Randomized Kinodynamic Planning [3]

Algorithm 2 BUILD_RRT(x_g, x_s)

Input: K: exploration steps of the algorithm, C_{free} configuration space, $G(V, E)$ RRT graph with vertices V and edges E)

```
1: G.init;                                        ▷ empty graph
2: G.add_vertex ($x_g$)                           ▷ add goal pose
3: for $k = 1$ to $K$ do
4:     $x_{rand} \leftarrow$ RANDOM_STATE($C_{free}$,$G$)    ▷ sample random state from map
5:     EXTEND($G, x_{rand}$)                      ▷ extend graph
```

Algorithm 3 EXTEND(G,x)

Input: C_{free} configuration space, $G(V, E)$ RRT graph with vertices V and edges E, x random pose)

```
1: $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x$,$G$)    ▷ compute vertex $V \in G$ closest to $x$
2: if NEW_STATE($x, x_{near}, x_{new}, u_{new}$) then    ▷ compute feasible pose $x_{new}$ close to $x_{near}$
3:     G.add_vertex ($x_{new}$)                    ▷ add pose to graph
4:     G.add_edge ($x_{near}, x_{new}, u_{new}$)    ▷ connect $x_{near}$ and $x_{new}$
5:     if $x_{new} = x$ then
6:         Return Reached
7:     else
8:         Return Advanced
9: else
10:    Return Trapped
```

```
load('exampleMaps.mat');
omap = occupancyMap(complexMap,2);
startPose = [2 1 pi/2];
goalPose = [22 18 pi/2];
```

Notice, that in contrast to the PRM planner, RRT plans with the robots initial and final orientation θ_s, θ_f.

The state space object defines the kinematics of the vehicle from which feasible controls and connections are sampled. The possible state space models are:

- `stateSpaceSE2` : state x, y, θ no constraints on turning radius, e.g. unicyle kinematic model
- `stateSpaceSE3` : state $x, y, z, \eta, \epsilon_x, \epsilon_y, \epsilon_z$ 6 DOF motion in 3D-space, e.g. UAVs
- `stateSpaceDubins` : state x, y, θ with minimal turning radius (`MinTurningRadius`) (bicycle kinematic model) and only forward motion
- `stateSpaceReedsShepp` : state x, y, θ with minimal turning radius (`MinTurningRadius`) (bicycle kinematic model) and forward and backward motion

To show the general usage of the Matlab functions, we consider a `stateSpaceDubins` object with bicycle kinematics with Dubin curves as connections together with a `plannerRRT` object in the following. Later in the exercise, you should adapt / extend the following code to investigate different state spaces and planning algorithms. The minimal turning radius of 2.0m is rather challenging in our environment.

```
stateSpace=stateSpaceDubins;
stateSpace.StateBounds = [omap.XWorldLimits;omap.YWorldLimits; [-pi pi]];
stateSpace.MinTurningRadius=2.0;
```

RRT randomly samples states within the state space and attempts to connect the nodes. Sampled states and connections are checked for collisions and accepted or rejected on the map constraints.

The `validatorOccupancyMap` object performs the state validation. Its `Map` property is the `occupancyMap` object. The `ValdiationDistance` property defines the discretization of the path connections and determines the resolution at which states are checked for collision. States beyond the `StateBounds` are not considered either.

```
stateValidator=validatorOccupancyMap(stateSpace);
stateValidator.Map=omap;
stateValidator.ValidationDistance=0.01;
```

Instantiate the path planner and define the max connection distance and the maximum number of iterations for sampling states.

```
planner=plannerRRT(stateSpace,stateValidator);
planner.MaxIterations=10000;
planner.MaxConnectionDistance=0.8;
```

Invoke the planner using

```
[pathObj,solnInfo] = planner.plan(startPose,goalPose);
```

To display the information about the solution and plot the generated tree and path, the helper function

```
function displayAndPlotPlanningResult(pathObj, solnInfo, map)
  if ~solnInfo.IsPathFound
    disp('no path found');
  else
   disp(['number of iterations=' num2str(solnInfo.NumIterations)]);
   disp(['total pathlength=' num2str(pathObj.pathLength(),2)]);
   show(map);
   hold on;
   plot(solnInfo.TreeData(:,1),solnInfo.TreeData(:,2),'.-'); % tree expansion
   plot(pathObj.States(:,1),pathObj.States(:,2),'r-','LineWidth',2); % draw path
   hold off;
  end
end
```

is provided. Figure 3 show the tree and the path for the Dubins state space model.

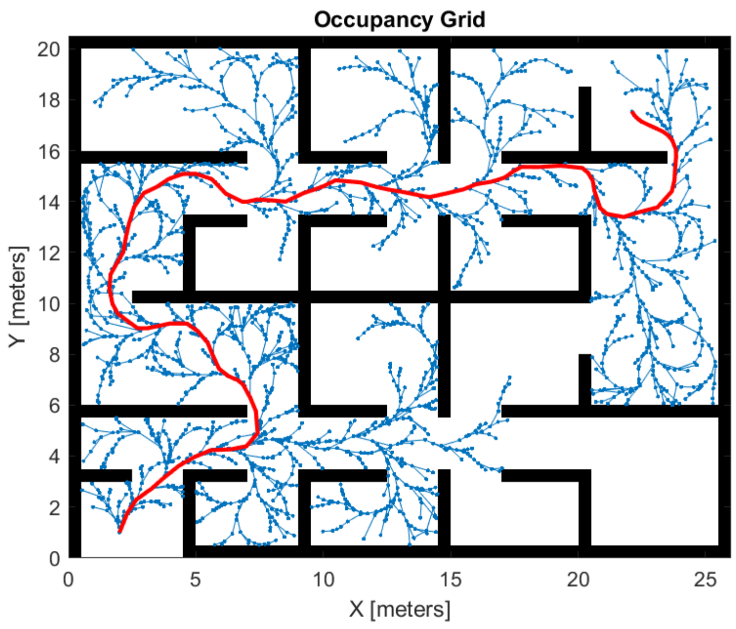

Figure 3: RRT graph and path for complex map for Dubins state space model [3]

[3]source: RST

13) Plan a path for a unicycle kinematics (`stateSpaceSE2`) with the RRT algorithm for the complex map with start and goal pose $x_i = [2\ 1\ \pi/2]$ and $x_f = [22\ 18\ \pi/2]$. Visualize the path and the tree using the helper function `displayAndPlotPlanningResult`.

14) Plan a path for a Dubins and Reeds-Shepp vehicle with minimum turning radius $r_{min} = 2.0$ with the RRT algorithm for the complex map with start and goal pose $x_i = [2\ 1\ \pi/2]$ and $x_f = [22\ 18\ \pi/2]$. It is advisable to use a Drop Down control field in the Live Editor code to select the state space model from a drop down list of options.

```
stateSpace = % Drop Down control field (stateSpaceSE2, stateSpaceDubins,
  stateSpaceReedsShepp)
stateSpace.StateBounds = [omap.XWorldLimits;omap.YWorldLimits; [-pi pi]];
if ~isa(stateSpace,'stateSpaceSE2')
stateSpace.MinTurningRadius=2.0;
end

```

Since the `stateSpaceSE2` model does not possess the property `MinTurningRadius` do the assignment only for the Dubins and ReedsShepp state space model.

15) Visualize the path and the tree.

16) Report the number of iterations and the path length for the three kinematic models according to the table below.

planner	kinematics	# iterations	path length
RRT	unicycle		
RRT	Dubins		
RRT	ReedsShepp		

17) Investigate the effect of the planner property `MaxConnectionDistance` on the number of iterations as well as the path length. It is advisable to use a Numeric Slider control field in the Live Editor to easily explore different values for `MaxConnectionDistance`.

Bidirectional Rapidly Exploring Random Trees (RRT)

The bidirectional rapidly exploring random tree partitions the graph expansion in two alternating phases, namely exploration of the state space and attempting to grow the trees into each other. It maintains two graphs Ga and Gb until they become connected. In each iteration either of both graphs expands with the attempt to connect the closest vertex of the opposite tree to the novel vertex. The next iteration reverses the roles of expansion and target graph. The extend operation switches between expansion to either random states or nearest vertex of the opposing graph.

The code for using the Bidirectional RRT planner is quite similar. Instantiate the path planner and define the max connection distance and the maximum number of iterations for sampling states.

Algorithm 4 RRT_BIDIRECTIONAL(x_g, x_s)

```
Input: $K$: exploration steps of the algorithm, $C_{free}$ configuration space, $G(V,E)$ RRT
    graph with vertices $V$ and edges $E$)
1: Ga.init; Gb.init                                        ▷ empty graphs
2: Ga.add_vertex ($x_i$) Gb.add_vertex ($x_g$)             ▷ add goal and start pose
3: for $k = 1$ to $K$ do
4:     $x_{rand} \leftarrow$ RANDOM_STATE($C_{free}$,$G$)  ▷ sample random state from map
5:     if not (EXTEND($Ga, x_{rand}$)=$Trapped$) then
6:         if EXTEND($Gb, x_{new}$=$Reached$) then
7:             Return PATH(Ga,Gb);
8:         SWAP(Ga,Gb)
9:     Return Failure
```

```
planner=plannerBiRRT(stateSpace,stateValidator);
planner.MaxIterations=10000;
planner.MaxConnectionDistance=0.8;
planner.EnableConnectHeuristic=true;
```

The option `EnableConnectHeuristic` attempts to directly join both trees during the connect phase of the planner.

Figure 4 show the two trees and the path for the Dubins state space model. The solution info object and the code for plotting the two trees are slightly different compared to the RRT planner.

```
plot(solnInfo.StartTreeData(:,1),solnInfo.StartTreeData(:,2),'.c-');
plot(solnInfo.GoalTreeData(:,1),solnInfo.GoalTreeData(:,2),'.m-');
```

18) Plan a path for a unicycle kinematics (`stateSpaceSE2`) with the BiRRT algorithm for the complex map with start and goal pose $x_i = [2\ 1\ \pi/2]$ and $x_f = [22\ 18\ \pi/2]$.

19) Visualize the path and the two trees. For this, extend the helper function `displayAndPlotPlanningResult` accordingly.

20) Plan a path for a Dubins and Reeds-Shepp vehicle ($r_{min} = 2.0$) with the BiRRT algorithm for the complex map with start and goal pose $x_i = [2\ 1\ \pi/2]$ and $x_f = [22\ 18\ \pi/2]$.

21) Report the number of iterations and the path length for the three kinematic models in the table below.

[5]source: RST

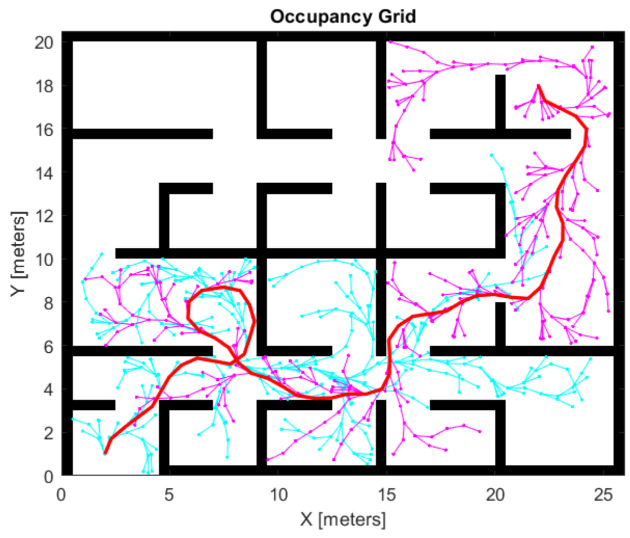

Figure 4: Bidirectional RRT graph and path for complex map for Dubins state space model [5]

planner	kinematics	# iterations	path length
RRT	unicycle		
RRT	Dubins		
RRT	ReedsShepp		
BiRRT	unicycle		
BiRRT	Dubins		
BiRRT	ReedsShepp		

22) Investigate the effect of the flag `EnableConnectHeuristic` on the number of iterations as well as the path length.

23) (Optional) Investigate the performance of ther RRT* planning algorithm (`plannerRRTStar`).

References

[1] Lydia .E. Kavreki, Steven M. La Valle, Motion Planning, *Springer Handbook of Robotics*

[2] Lydia .E. Kavreki et al, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, *IEEE Trans. Robotics and Automation*, vol. 12, no. 4, pp. 566-580, (1996)

[3] Steven M. La Valle, James J. Kuffner, Randomized Kinodynamic Planning, *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378-400, (2001)