

Homework 5c

ME 570 — Prof. Tron

2020-11-20

In this homework, you will implement the simplest variation of EST, one of the sampling-based methods seen in class. The main effort for its implementation is in the sampling functions (since it builds a tree, there is no need for a separate graph search). If the assignment mentions a file not specifically provided, it refers to a file from a previous assignment (in fact, you will need functions from Homework 2, 3 and 4).

General instructions

Programming For your convenience, together with this document, you will find a **zip** archive containing Matlab files with stubs for each of the questions in this assignment; each stub contains an automatically generated description of the function and the function header. You will have to complete these files with the body of each function requested. The goal of this files is to save you a little bit of time, and to avoid misspellings in the function or argument names. The functions from the parts marked as **provided** (see also the *Grading* paragraph below) contain already the body of the function.

Please refer to the guidelines on Blackboard under Information/Programming Tips & Tricks/Coding Practices.

Homework report Along the programming of the requested functions, prepare a small PDF report containing one or two sentences of comments for each question marked as **report**, and including:

- Embedded figures and outputs that are representative of those generated by your code.
- All analytical derivations if required.

You can include comments also on the questions marked as **code**, if you feel it necessary (e.g., to explain any difficulty you might have encountered in the programming), but these will not be graded. In general, *do not* insert the listing of the functions in the report (I will have access to the source files); however, you can insert *short* snippets of code if you want to specifically discuss them.

A small amount of *beauty points* are dedicated to reward reports that present their content in a professional way (see the *Grading criteria* section in the syllabus).

Analytical derivations To include the analytical derivations in your report you can type them in L^AT_EX(preferred method), any equation editor or clearly write them on paper and use a scanner (least preferred method).

Submission

The submission will be done on Blackboard. Please upload a single ZIP file with your name, and containing both the PDF of the report, and all the code necessary to reproduce the figures in the paper.

Optional and provided questions. Questions marked as **optional** are provided just to further your understanding of the subject, and not for credit. Questions marked as **provided** have already a solution provided in the file accompanying this assignment, which you can directly use unless you want to answer it by yourself. If you submit an answer for optional or provided questions I will correct it, but it will not count toward your grade.

Hints

Some hints are available for some questions, and can be found at the end of the assignment (you are encouraged to try to solve the questions without looking at the hints first). If you use these hints, please state so in your report (your grading will not change based on this information, but it is a useful feedback for me).

Use of external libraries and toolboxes All the problems can be solved using Matlab's standard features. You are **not allowed** to use functions or scripts from external libraries or toolboxes (e.g., mapping toolbox), unless specifically instructed to do so.

Problem 1: Sampling tree

This problem considers a random *tree* for solving a single path query. One advantage of using a tree instead of a general graph is that you will not need to call `graph_search`(`_`), as the backpointers in `graphVector` can be set while constructing the tree.

We will use the sphere world from Homework 3 for testing. Please refer to that homework for a detailed description of the `world` data structure.

We will use the sphere world from Homework 3 for testing. Please refer to that homework for a detailed description of the `world` data structure.

Question provided 1.1. This function implements a collision check against all spheres in the world.

```
[flag] = sphereworld_isCollision(world,x)
```

Description: Checks if the points with coordinates `x` is inside any of the spheres in `world`. This is done by iterating over all spheres, using the function `sphere_distance`(`_`) from Homework 3 on each one of them, and then combining the results.

Input arguments

- `world` (dim. $[NSpheres \times 1]$, type `struct array`): see definition in Homework 3.
- `x` (dim. $[2 \times NPoints]$): array with the coordinates of the sampled locations (one for each column) that need to be tested for collisions.

Output arguments

- `flag` (dim. $[1 \times NPoints]$, type `logical`): `flag(iPoint)` is `true` if `x(:,iPoint)`

is inside one of the spheres (i.e., it is a collision), `false` otherwise.

Question provided 1.2. Create a function that samples from the free configuration space using rejection sampling.

```
[xSample]=sphereworld_sample(world,...)
```

Input arguments

- `world` (dim. $[NSpheres \times 1]$, type `struct array`): see definition in Homework 3.

Optional arguments

- `'distribution', distribution` : Selects the type of distribution to be used. Possible values for `distribution` are 'uniform' and 'Gaussian'.
- `'size', sz` : Size parameter for the distribution (default value: `sz=10`). For the 'uniform' distribution, the resulting domain of the random variable is the square $[-maxSize, maxSize]^2$. For the 'Gaussian' distribution, `sz` represents the variance.
- `'mean', mu` (dim. $[2 \times 1]$): Mean of the distribution (default value: `mu=[0;0]`).

Output arguments

- `xSample` (dim. $[2 \times 1]$): a sample in the free space. Can be empty if a sample cannot be generated in `nbTrials=1000` trials.

Description: Generate samples using the specified distribution, until `sphereworld_isCollision(.)` returns `false` for a sample, or a maximum number of trials `nbTrials` is reached.

Question code 1.1. This function samples locations along a line between two samples, and checks if this line can be used to create an edge in the roadmap.

```
[flag]=prm_localPlannerIsCollision(world,x1,x2,maxDistEdgeCheck)
```

Description: Generates `NPoints` equispaced points on the line between `x1` and `x2`, such that the maximum distance between two consecutive samples is less than `maxDistEdgeCheck`. Note that the value of `NPoints` is determined by the distance between the two points and `maxDistEdgeCheck`.

Input arguments

- `world` (dim. $[NSpheres \times 1]$, type `struct array`): see definition in Homework 3.
- `x1` (dim. $[2 \times 1]$), `x2` (dim. $[2 \times 1]$): coordinates of the endpoints of the line on which to sample the locations.
- `maxDistEdgeCheck` (dim. $[1 \times 1]$): maximum distance between sampled locations.

Output arguments

- `flag` (dim. $[1 \times 1]$, type `logical`): `true` if *any* of the sampled locations is in collision, `false` otherwise.

Requirements: This function returns an “all-or-nothing” result, meaning that the edge between `x1` and `x2` even if only one of the samples is in collision.

Question provided 1.3. Implement a function that can sample from any arbitrary discrete probability distribution.

```
[x]=rand_discrete(xDistr,fDistr)
```

Description: Returns a sample `x` of a random variable X that can assume values in `xDistr` with probability given by `fDistr` (that is, $p(\mathbf{x} = \mathbf{xDistr}(i)) = \mathbf{fDistr}(i)$).

Input arguments

- `xDistr` (dim. $[N\text{Values} \times 1]$): set of values that X can assume.
- `fDistr` (dim. $[N\text{Values} \times 1]$): array describing the probability of each value in `xDistr`. Note: the function normalizes `fDistr` so that it sums to one.

Output arguments

- `x` (dim. $[1 \times 1]$): a sample from the distribution. Internally, the function uses `rand(.)` and the cumulative distribution of X to generate `x`.

Question provided 1.4. A utility function for graphs.

```
[deg]=graph_degree(graphVector)
```

Description: The function returns an array with the degree $\deg(n)$ of each vertex n in the graph `graphVector`.

Input arguments

- `graphVector` (dim. $[N\text{Nodes} \times 1]$, type `struct`): the structure describing a graph, as specified in Homework 4.

Output arguments

- `deg` (dim. $[N\text{Nodes} \times 1]$): the number of neighbors for each element in `graphVector`.

Question code 1.2. This function performs connection sampling to generate candidates for growing the tree.

```
[graphVector,xSample]=sampleTree_extend(graphVector,radius,world)
```

Description: The function performs the following steps:

- 1) Uses `graph_degree(.)` to obtain the array of degrees `deg` of the nodes in the tree.
- 2) Computes an array `fDistr` such that `fDistr(i)` is proportional to $1/(\deg(i)+1)$.
- 3) Use `rand_discrete(.)` to sample a node with index `idxSample` according to `fDistr`.
- 4) Use `sphereworld_sample(.)` with the optional arguments to generate a sample location `xSample` around the location of `idxSample` according to a Gaussian

distribution with variance `radius`.

- 5) Use `graph_nearestNeighbors` () (with `kNeighbors=1`) to find the index `idxNear` of the single closest neighbor of `xSample`.
- 6) Use `prm_localPlannerIsCollision` () to check whether `xSample` can be connected to the closest neighbor. If there is no collision, add `xSample` to `graphVector`, setting its backpointer to `idxNear`, and updating the `neighbor` and `neighborCost` fields of `graphVector[idxNear]`.
- 7) Repeat the steps above from 3) until a sample is successfully added, or up to 100 times.

Input arguments

- `graphVector` (dim. $[N\text{Nodes} \times 1]$, type `struct`): the structure describing the tree to extend, as specified in Homework 4.
- `radius`: variance of the sampling around a given node.
- `world` (dim. $[N\text{Spheres} \times 1]$, type `struct array`): see definition in Homework 3. This is needed to be able to check for collisions.

Output arguments

- `graphVector` (dim. $[N\text{Nodes}+1 \times 1]$, type `struct`): same as the corresponding input argument, but with an additional node (unless the extension fails).
- `xSample` (dim. $[2 \times 1]$): the location of the additional node (empty if the extension fails).

Question optional 1.1. The function `sampleTree_extend` () as described above implements a variant of EST. Add an optional argument to this function to transform the algorithm into RRT, RRT with greedy search, or RRT with greedy search and sampling biased toward the goal.

Question code 1.3. In this question you will build around the function `sampleTree_extend` () to generate a complete planner.

```
[xPath,graphVector]=sampleTree_search(world,xStart,xEnd,goalDistThreshold)
```

Description: Implement a tree-based sampling search algorithm. The function performs the following steps:

- 1) Initializes a `graphVector` containing only a vertex at `xStart`.
- 2) Call `sampleTree_extend` () to grow the tree with a new sample with location `xSample`.
- 3) If the distance between `xSample` and `xEnd` is less than `goalDistThreshold`, call `prm_localPlannerIsCollision` () to try to connect the goal to the tree:
 - (a) If the connection does *not* succeed, repeat from step 2), up to `NTrials=1000` times.

- (b) If the connection succeeds, add `xEnd` to `graphVector` (with the correct backpointer), and then use `graph_path()` to return the path from `xStart` to `xEnd`.

Input arguments

- `world` (dim. $[NSpheres \times 1]$, type `struct array`): see definition in Homework 3.
- `xStart` (dim. $[2 \times 1]$), `xEnd` (dim. $[2 \times 1]$): the coordinates of the initial and final locations.
- `goalDistThreshold` (dim. $[1 \times 1]$): threshold for trying to connect the goal to a newly generated extension of the tree. If too large, the planner will spend too much time checking if the goal can be reached from the tree. If too small, the planner will be unlikely to generate a sample close enough to the goal to connect it.

Output arguments

- `xPath` (dim. $[2 \times NPath]$): array where each column contains the coordinates of the points of the path found from `idxStart` to `idxEnd`. If the planner fails, it returns an empty array.
- `graphVector` (dim. $[NNodes \times 1]$, type `struct`): the structure describing the tree built by the algorithm, as specified in Homework 4.

Question report 1.1. In this section you will test your planner.

`sampleTree_search_test()`

Description:

- 1) Loads the variables `world`, `xStart` and `xEnd` from the file `sphereWorld.mat`.
- 2) For each starting location (column) in `xStart`, and each goal location (column) in `xGoal`, calls `sampleTree_search()` to find a path.
- 3) Calls `sphereworld_draw()` and then overimposes all the trajectories from the start locations to the goal location.

Start with `goalDistThreshold` and `radius` both around 1, and then try to decrease them until you are satisfied with the performance. In the report, include a table with the values you tried, and the images from the final values.

Question optional 1.2. Modify and run `sampleTree_search_test()` so that they run the respective planners multiple times on the same data, and show how the variance of the length of the paths found across different trials.

Problem 2: Homework feedback

Question report 2.1. Indicate an estimate of the number of hours you spent on this homework (possibly broken down by problem). Explain what you found hard/easy about this homework, and include any suggestion you might have for improving it.

Hint for question code 1.1: To generate the samples, you can use the function `line_linspace ()` from Homework 2, with `a=x2-x1` and `b=x1` (what should you use for `tMin` and `tMax`?)