EECE 5550 Mobile Robotics HW #2

Instructor: Michael Everett

Due: Feb 7, 2023 at 11:00am

This assignment should be submitted via Gradescope. Your answers to both problems 1 & 2 should be submitted as the written portion. We do not plan to run your code this time (i.e., no autograder), but please also upload your code to Gradescope anyway.

Question 1: Bayesian inference with linear-Gaussian models

In this exercise we will study Bayesian estimation in linear-Gaussian models; as we will see later in the course, these play a fundamental role in robotic state estimation (most prominently in the celebrated Kalman filter).

We begin by recording a few useful facts. Recall that:

$$X \sim \mathcal{N}(\mu, \Sigma) \tag{1}$$

means that $X \in \mathbb{R}^n$ is a random variable that follows a Gaussian distribution with mean $\mu \in \mathbb{R}^n$ and covariance $\Sigma \in \mathbb{S}^n_{++}$. As we saw in class, X is described by the following probability density function:

$$\begin{gathered} p_X : \mathbb{R}^n \to \mathbb{R} \\ p_X(x) \triangleq \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\left(-\frac{1}{2}(x-\mu)^\mathsf{T}\Sigma^{-1}(x-\mu)\right). \end{gathered} \tag{2}$$

If we expand the quadratic form and ignore the normalization constant in (2), we find:

$$p_X(x) \propto \exp\left(-\frac{1}{2}\left(x^\mathsf{T}\Sigma^{-1}x - 2\mu^\mathsf{T}\Sigma^{-1}x\right)\right). \tag{3}$$

It follows from (3) that *any* function of the form:

$$f(z) = \frac{1}{c}\exp\left(-\frac{1}{2}\left(z^\mathsf{T}\Lambda z - 2\eta^\mathsf{T} z\right)\right) \tag{4}$$

with $c > 0$ is an unnormalized density for a Gaussian random variable $Z \sim \mathcal{N}(\bar{\mu}, \bar{\Sigma})$ with parameters:

$$\bar{\Sigma} = \Lambda^{-1}, \qquad \bar{\mu} = \bar{\Sigma}\eta. \tag{5}$$

Equations (4) and (5) give an alternative way of parameterizing a Gaussian probability density, called the *information* or *canonical form*.

Now, suppose that Θ is a random variable with prior distribution:

$$\Theta \sim \mathcal{N}(\mu_0, \Sigma_0), \tag{6}$$

and that we collect a set of m noisy linear measurements $\tilde{Y}_1, \dots, \tilde{Y}_m$ of Θ according to:

$$\tilde{Y}_i = A_i \Theta + b_i + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(\mu_i, \Sigma_i), \tag{7}$$

where A_i, b_i, μ_i, and Σ_i are known parameters for all $i = 1, \dots, m$. In this exercise, you will determine the posterior distribution for Θ given the measurements $\tilde{Y}_1, \dots, \tilde{Y}_m$.

(a) Use Bayes' Rule to express the posterior density $p(\Theta | \tilde{Y}_1, \dots, \tilde{Y}_m)$ in terms of the prior $p(\Theta)$ and the measurement likelihoods $p(\tilde{Y}_i | \Theta)$ for each individual measurement. You may leave your result in an unnormalized form.

(b) Derive an expression for the likelihood function $p(\tilde{Y}_i | \Theta)$ of the ith measurement. (Hint: Notice that you can easily solve (7) for ϵ_i.)

(c) Using your results from parts (a) and (b), derive the parametric form of the posterior density $p(\Theta | \tilde{Y}_1, \dots, \tilde{Y}_m)$. You should simplify your result by collecting linear and quadratic terms in Θ in the exponent. You may leave your result in an unnormalized form.

    (Hint: Since your result need not be normalized, any term appearing in an exponent that does *not* involve Θ can be be discarded by absorbing it into the normalization constant. You can use this fact to dramatically simplify your work.)

(d) You should be able to recognize your expression for $p(\Theta | \tilde{Y}_1, \dots, \tilde{Y}_m)$ in part (c) as an unnormalized Gaussian density in information form. This shows that the posterior distribution for Θ is Gaussian; that is, $\Theta | \tilde{Y}_1, \dots, \tilde{Y}_m \sim \mathcal{N}(\bar{\mu}, \bar{\Sigma})$ for some mean $\bar{\mu}$ and covariance $\bar{\Sigma}$. What are the mean $\bar{\mu}$ and covariance $\bar{\Sigma}$ of this distribution?

Problem 2: Route planning in occupancy grid maps

As we will see later in the semester, occupancy grid maps provide a convenient representation of a robot's environment that is particularly well-suited to route planning for navigation.

For example, Fig 1a shows a (probabilistic) occupancy grid map of a research lab constructed using the Cartographer SLAM system[1], and Fig. 1b the resulting estimate of free and occupied space obtained by thresholding these occupancy probabilities to binary values.

In this exercise, you will implement two of the graph-based planning algorithms that we discussed in class (A* search and probabilistic road maps) to perform route-planning in the (binary) occupancy grid map shown in Fig. 1b.

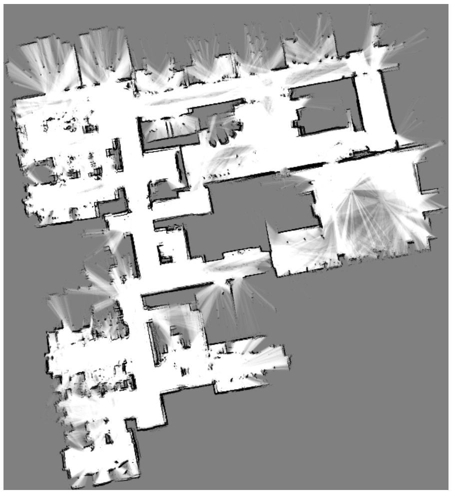
Note: Since the A* search algorithm requires the use of several data structures other than basic matrices (e.g. sets and priority queues), we recommend implementing the following exercise in a Python notebook (feel free to use colab to manage your dependencies again!).

(a) **A* search:** In the first part of this exercise, you will implement a general version of A^* search that is abstracted with respect to the choice of representation of the graph G. This will enable you to apply it to *both* occupancy grids (considered as 8-connected graphs) *and* "standard" graphs G (for use with probabilistic roadmaps).

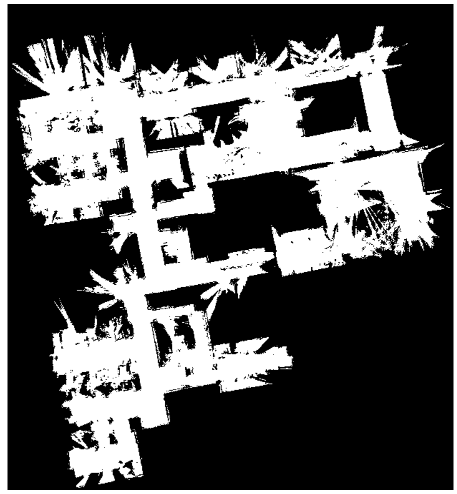
    The pseudocode for this version of A^* search is shown as Algorithm 1. The following objects appear in Algorithm 1:

    - "CostTo" is a map that assigns to each vertex v the cost of the shortest known path from the start node s to v.

[1] We will see later in the course how to construct these maps from raw sensor data.



(a) Probabilistic occupancy grid map



(b) Binary occupancy grid map

- "pred" is a map that associates to each vertex v its predecessor on the shortest known path from the start s to v.
- "EstTotalCost" is a map that assigns to each vertex v the sum $\text{CostTo}(v) + h(v, g)$, the sum of the cost of the best known path to v *and* the predicted cost of the best path from v to the goal g; this is the estimated cost of the optimal path from the start s to the goal g that passes *through* vertex v.
- Q is a priority queue in which elements with *lower* values are removed *first*.
- "RecoverPath" is a function that takes as input the start state s, goal state g, and populated predecessor map pred, and returns the sequence of vertices on the optimal path from s to g.

Tip: In Python, you can use the dictionary class to implement the maps CostTo, pred, and EstTotalCost, a list or set to hold the vertex set V, and a sorted list or heap of (priority, vertex) tuples[2] to implement the priority queue Q.

(i) Implement the RecoverPath function.

(ii) Using your implementation of RecoverPath, implement the complete A* search algorithm shown in Algorithm 1. Your code should accept as input the vertex set V, the start and goal vertices s and g, and function handles for N, w, and h.

(b) **Route planning in occupancy grids with A* search:** In this part of the problem, you will apply your A* search algorithm to perform route planning directly on the occupancy grid map Fig. 1b, considered as an 8-connected graph. Note that in this part of the exercise, we will *identify* each vertex $v \in V$ (cell) using its row and column (r, c) in the occupancy grid.

(i) Given an occupancy grid map M in the form of a 2D binary array, where a value of 0 indicates "occupied" and a value of 1 indicates "free" space, implement the function $N(v)$ that returns the set of unoccupied neighbors of a vertex v (remember that we can't drive the robot through occupied space!) The vertex v and its neighbors should be expressed in the form of (row, column) tuples $v = (r, c)$.

[2] Python compares tuples in lexicographic order, so placing the priority first ensures that (priority, vertex) tuples will be sorted by priority, as desired.

Algorithm 1 An abstracted implementation of A^* search

Input: Graph $G = (V, E)$ with vertex set V and edge set E, nonnegative weight function $w\colon V \times V \to \mathbb{R}_+$ for the edges, admissible A* heuristic $h\colon V \times V \to \mathbb{R}_+$ that returns the the estimated cost-to-go, a set-valued function $N\colon V \to 2^V$ that returns the neighbors of a vertex v in G, starting vertex $s \in V$, goal vertex $g \in V$.

Output: A least-cost path from s to g if one exists, or the empty set $\varnothing$ if no path exists.

```
function A_STAR_SEARCH(V, s, g, N, w, h)
    // Initialization
    for v ∈ V do
        Set CostTo[v] = +∞.
        Set EstTotalCost[v] = +∞.
    end for
    Set CostTo[s] = 0.                       ▷ Cost to reach starting vertex s is 0.
    Set EstTotalCost[s] = h(s, g).           ▷ Estimated cost-to-go from s to g
    Initialize Q = {(s, h(s, g))}.           ▷ Insert start vertex s with value h(s, g)
    // Main loop
    while Q is not empty do
        v = Q.pop()                          ▷ Remove least-value element from Q
        if v = g then                        ▷ We have reached the goal!
            return recoverPath(s, g, pred)   ▷ Reconstruct and return optimal path
        end if
        for i ∈ N(v) do                      ▷ For each of v's neighbors
            pvi = CostTo[v] + w(v, i)        ▷ Cost of path to reach i through v
            if pvi < CostTo[i] then
                // The path to i through v is better than the previously-known best path to i,
                // so record it as the new best path to i.
                Update pred[i] = v
                Update CostTo[i] = pvi       ▷ Update cost of best path to i
                Update EstTotalCost[i] = pvi + h(i, g)
                if Q contains i then
                    Q.setPriority(i) = EstTotalCost[i]      ▷ Update i's priority
                else
                    Q.insert(i, EstTotalCost[i])  ▷ Insert i into Q with priority EstTotalCost[i]
                end if
            end if
        end for
    end while
    return ∅                                 ▷ Return empty set: there is no path to goal
end function
```

(ii) We will consider the cost of moving from a cell $v_1 = (r_1, c_1)$ to an adjacent cell $v_2 = (r_2, c_2)$ to be the Euclidean distance between the cell centers. Implement a function $d\colon V \times V \to \mathbb{R}_+$ that accepts as input the tuples v_1 and v_2, and returns this Euclidean distance.

(iii) We saw in class that the straight-line Euclidean distance between two points provides an admissible A^* heuristic h for route-planning using the total path length as the cost; this means that we can do route planning using your distance function d from part (ii) as both the edge weight w *and* the heuristic h.

Using your implementations of d, N, and A^* search, find the shortest path in the occupancy grid Fig. 1b from the starting point $s = (635, 140)$ to the goal $g = (350, 400)$ [assuming 0-based indexing for rows and columns, as is standard in CS.] Plot this optimal path overlaid on the image, and calculate its total length.

Tip: In Python, you can use the Python Imaging Library to easily manipulate basic image data. The following code snippet will read the occupancy map file from disk, interpret it as a Python numpy array, and then threshold it to produce the binary array M required in part (i):

(c) **Route planning with probabilistic roadmaps:** Voxelized grids (like occupancy maps) provide simple and convenient models of robot configuration spaces, but as we will see in class their memory requirements scale *exponentially* in the dimension of the state space, making them far too costly to use for higher-dimensional planning problems.

Thus, *sampling-based planners* can sometimes provide a tractable alternative for planning in high-dimensional spaces. Recall that these methods *approximate* the configuration space C using a graph $G = (V, E)$ whose vertex set $V \subset C$ is a *randomly sampled subset* of points in C, and where two vertices $v_1, v_2 \in V$ are joined by an edge if v_2 is *reachable* from v_1 by applying a local controller.

In this part of the exercise, you will implement a sampling-based planner to perform route planning in the occupancy grid shown in Fig. 1b; more specifically, you will implement a *probabilistic roadmap* (PRM). Recall that we construct a PRM incrementally by sampling a new vertex $v_{new} \in C$, and then attempting to join v_{new} to nearby vertices $v \in G$ using a local planner (cf. Algorithms 2 and 3). In order to implement this approach, we must therefore specify:

- A method for sampling new vertices $v_{new} \in C$ (line 4 of Alg. 2)
- A suitable distance function $d \colon V \times V \to \mathbb{R}_+$ for characterizing "nearby" vertices (line 3 of Alg. 3)
- A local planner (line 4 of Alg. 3)

Algorithm 2 Construction of a probabilistic roadmap

Input: Desired number of sample points N, maximum local search radius d_{max}.
Output: A graph $G = (V, E)$ consisting of a vertex set $V \subseteq E$ of cardinality N, and edge set E indicating reachability via local control.

1: **function** CONSTRUCTPRM(N, d_{max})
2: Initialize $V = \varnothing$, $E = \varnothing$.
3: **for** $k = 1, \ldots, N$ **do**
4: Sample a new vertex $v_{new} \in C$.
5: ADDVERTEX(G, v_{new}, d_{max})
6: **end for**
7: **return** $G = (V, E)$
8: **end function**

(i) Implement a function that accepts as input the occupancy grid map M, and returns a vertex $v = (r, c)$ sampled *uniformly randomly* from the free space in M. [Hint: consider rejection sampling with a uniform proposal distribution.]

Algorithm 3 Adding a vertex v_{new} to the probabilistic roadmap $G = (V, E)$

1: **function** AddVertex(G, v_{new}, d_{max})
2: $\quad V \leftarrow V \cup \{v_{new}\}$. $\qquad \triangleright$ Add vertex v_{new} to G
3: $\quad$ **for** $v \in V$ satisfying $v \neq v_{new}$ and $d(v, v_{new}) \leq d_{max}$ **do** $\quad \triangleright$ Link v_{new} to nearby vertices
4: $\qquad$ Attempt to plan a path from v_{new} to v.
5: $\qquad$ **if** planning succeeds **then**
6: $\qquad\quad E \leftarrow E \cup \{(v, v_{new})\}$ $\qquad \triangleright$ Add edge $e = (v, v_{new})$ to G
7: $\qquad$ **end if**
8: $\quad$ **end for**
9: **end function**

(ii) We saw in class that is easy to plan straight-line paths between arbitrary points using a differential drive robot, since the robot can rotate in-place to face the correct direction before beginning to move. Therefore, we might consider using a *straight-line path planner* as our local planner in line 4 of Alg. 3. Using this approach, a point $v_2 \in V$ is reachable from v_1 if and only if the line segment joining v_1 and v_2 does not intersect any occupied cells in M.

Implement a function that performs this reachability check. Your function should accept as input the occupancy grid map M and two grid cells $v_1 = (r_1, c_1)$ and $v_2 = (r_2, c_2)$, and return a Boolean value indicating whether the line segment joining v_1 and v_2 in M is obstacle-free.

(iii) With the aid of your results from parts (c)(i) and (c)(ii), and your distance function implementation from (b)(iii), implement Algorithm 2 in the form of a function that accepts as input an occupancy grid map M, the desired number of samples N, and the maximum local search radius d_{max}, and returns a PRM G constructed from M.

Tip: You may find it convenient to model the PRM G using the Graph class in Python's NetworkX library. If you do so, you can record the location (row and column) of each vertex v by setting its `pos` attribute. Similarly, given any straight-line path joining two vertices $v_1, v_2 \in V$ found in line 4 of Alg. 3, you can store the length of this path as the *weight* of the edge $e_{12} = (v_1, v_2)$ joining v_1 and v_2 in G. The following code snippet provides a minimal working example:

(iv) Using your implementation of Algorithm 2, construct a PRM on the occupancy grid in Fig. 1b with $N = 2500$ samples and a maximum local search radius of $d_{max} = 75$ voxels. Plot the resulting graph overlaid on Fig. 1b. [Hint: you may find NetworkX's draw_networkx function useful here.]

(v) Recall that given a PRM G for a configuration space C, and start $s \in C$ and goal $g \in C$, we can plan a route from s to g by first *adding* s and g to the PRM, and then searching for a shortest path from s to g in G.

Using the PRM you constructed in part (v), find a path from $s = (635, 140)$ to $g = (350, 400)$. [Note: If s and g initially lie in separate connected components of G, you may need to sample and add more vertices to G until s and g are path-connected.] Plot this path overlaid on Fig. 1b, and calculate its total length.

Tip: In part (v), you may use NetworkX's implementation of A* search.