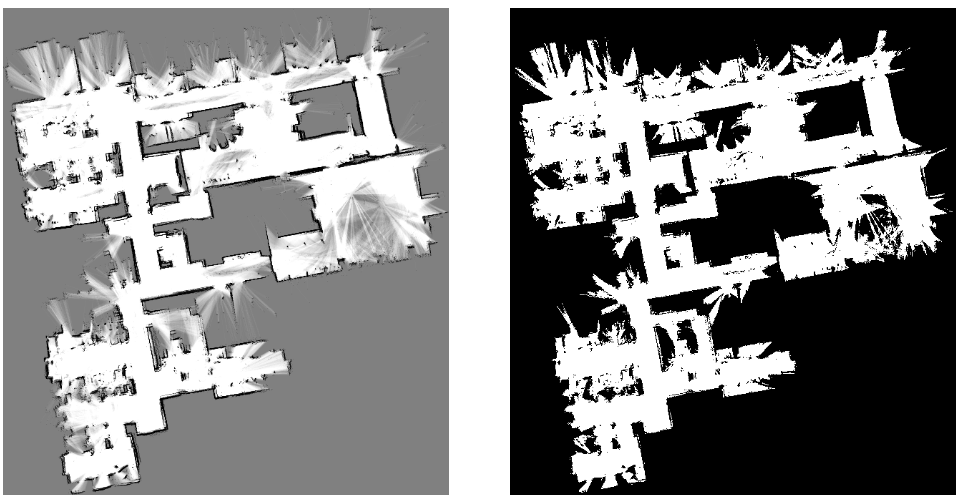
EECE 5550 Mobile Robotics - Section 1
Lab #4

Due: Dec 5, 2025 by midnight

Problem 1: Route planning in occupancy grid maps

As we learned in our course, 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, and Fig. 1b the resulting estimate of free and occupied space obtained by thresholding these occupancy probabilities to binary values.



(a) Probabilistic occupancy grid map

(b) Binary occupancy grid map

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.

(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.
- "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[1] 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 $v = (r, c)$.

(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 $g = (350, 400)$

[1] 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 $v \in V$ **do**
        Set CostTo[v] $= +\infty$.
        Set EstTotalCost[v] $= +\infty$.
    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 $i \in$ 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 $\varnothing$ ▷ Return empty set: there is no path to goal
end function

[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):

```
# Load the PIL and numpy libraries
```

```python
from PIL import Image
import numpy as np

# Read image from disk using PIL
occupancy_map_img = Image.open('occupancy_map.png')

# Interpret this image as a numpy array, and threshold its values to {0,1}
occupancy_grid = (np.asarray(occupancy_map_img) > 0).astype(int)
```

(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: $\quad V \leftarrow V \cup \{v_{new}\}$. $\quad\triangleright$ 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: $\quad\quad$ Attempt to plan a path from v_{new} to v.
5: $\quad\quad$ **if** planning succeeds **then**
6: $\quad\quad\quad E \leftarrow E \cup \{(v, v_{new})\}$ $\quad\triangleright$ Add edge $e = (v, v_{new})$ to G
7: $\quad\quad$ **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:

```python
# Import the NetworkX library
import networkx as nx

# Create empty graph
G = nx.Graph()

# Add vertex v1 at position (r1, c1)
G.add_node(1, {'pos' : (r1, c1)})

# Add vertex v2 at position (r2, c2)
G.add_node(2, {'pos' : (r2, c2)})

# Add an edge between v1 and v2 with weight w12
G.add_edge(1, 2, weight=w12)
```

(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 (iv), 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.

Problem 2: PID altitude control

Consider the following second order dynamical system that models the altitude dynamics of a drone:

$$\ddot{h} = \frac{4k_T u}{m} - g, \qquad IC: h(0) = 0, \dot{h} = 0, \tag{1}$$

where $m = 65g$ is the mass of the drone, $k_T = 5.276 \times 10^{-4}$ is the thrust coefficient, $g = 9.81m/s^2$ is the gravitational constant, $h \in \mathbb{R}$ is the altitude, and $u \in \mathbb{R}$ is the control input. Let u be designed as

$$u = PID + \frac{mg}{4k_T}, \tag{2}$$

where PID is the proportional-integral-derivative control architecture, and the last element is for the perfect gravity cancellation as we discussed in the class. The objective of the drone is to reach a reference altitude of $r = 1m$ and hover there.

(a) Design a P control using (2) for values of $K_p = 5, 15, 50$, plot h and $\dot{h}$, discuss the results.

(b) Design a PD control using (2) such that the closed-loop system is underdamped and the settling time is approximately 3 seconds, plot h and $\dot{h}$, discuss the results and justify why the system is underdamped.

(c) Design a PD control using (2) such that the closed-loop system is overdamped and the settling time is approximately 3 seconds, plot h and $\dot{h}$, discuss the results and justify why the system is overdamped.

(d) In this part, you will consider some uncertainty in the actuators so the actual control applied to the system will be less than the designed control input. First, reproduce the results of part (b) by considering $u' = 0.95u$ where u is defined as in (2). Then design a PID control using the same K_p, K_d gains computed in (b) and some nonzero K_i gain while the actual control applied to the system is in the form of u'. Plot h and $\dot{h}$, and discuss the differences between the responses obtained by the PD and PID controls.

Problem 3: Nonlinear feedback and stability analysis

In this problem, you will devise a feedback controller to stabilize a damped driven pendulum in the upright position. Recall the equations of motion for a damped pendulum driven by an external torque τ:

$$ml^2\ddot{\theta} = -mgl\sin(\theta) - \mu\dot{\theta} + \tau, \tag{3}$$

where m is the mass of the bob, l is the length of the pendulum, μ is the damping parameter (due to friction), g is the gravitational acceleration, and τ is the external torque applied to the pendulum's pivot.

(a) Defining $x = (\theta, \dot{\theta}) \in \mathbb{R}^2$, rewrite the second-order equations of motion (3) in the form of a *first*-order differential system for x:

$$\dot{x} = f(x, \tau). \tag{4}$$

Note that your function $f\colon \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$ should express the derivative $\dot{x}$ as a function of *both* the state x *and* the external torque $\tau \in \mathbb{R}$.

(b) Let

$$\begin{aligned} g\colon \mathbb{R}^2 &\to \mathbb{R}^2 \\ g(x) &\triangleq f(x, 0) \end{aligned} \tag{5}$$

denote the equations of motion for the *free* system (i.e., the dynamics of the system when *no external torque* is applied). Show that the upright configuration $x^* = (\pi, 0)$ is an *unstable* stationary point for the free system.

Linear control synthesis: In the next part of this exercise, you will apply linearized stability analysis to devise a PD controller to stabilize the pendulum in the upright configuration.

(c) We will assume a PD controller of the form:

$$\begin{aligned} \tau\colon \mathbb{R}^2 &\to \mathbb{R} \\ \tau(x) &\triangleq k_p \sin(x_1) + k_d x_2, \end{aligned} \tag{6}$$

where $k_p, k_d \in \mathbb{R}$ are controller gains (to be determined). Using the control law (6), derive an explicit expression for the *closed-loop* dynamics of the system in the form of an *autonomous* ODE:

$$\dot{x} = c(x) \triangleq f(x, \tau(x)). \tag{7}$$

(d) Using your result in part (c), derive conditions on the controller gains k_p, k_d that are sufficient to guarantee that $x^* = (\pi, 0)$ is a (locally) asymptotically stable stationary point of the closed-loop system. [Hint: Show that x^* is a stationary point for the closed-loop system, and then apply linearized stability analysis to derive sufficient conditions on the controller gains.]

Controller design via linearized stability analysis [as you did in part (d)] is convenient in that it provides an easy method of constructing *locally* asymptotically stabilizing controllers using only a bit of linear algebra. However, because this approach is based upon (local) *linearization*, it doesn't directly provide any information about the *size* of the neighborhood around x^* over which the resulting controller works.

Nonlinear control synthesis: As an alternative approach, in the remainder of this question, you will apply the theory of Lyapunov functions to devise a *nonlinear* feedback controller whose *invariant subsets* we can explicitly characterize.

(e) Consider a candidate Lyapunov function of the form:

$$
\begin{gathered}
V : \mathbb{R}^2 \to \mathbb{R} \\
V(x) \triangleq -mgl(1 + \cos(x_1)) + \alpha mgl(1 - \cos^2(x_1)) + \frac{1}{2} ml^2 x_2^2
\end{gathered} \tag{8}
$$

where $\alpha \in \mathbb{R}$ is a free parameter (to be determined). Calculate the gradient and Hessian of V, and derive a sufficient condition on α for the point $V(x^*) = 0$ to be an isolated local minimizer of V.

(f) Your results in part (e) show that there exists some neighborhood $U \in \mathbb{R}^2$ containing x^* such that $V(x) \geq 0$ for all $x \in U$, and x^* is the *unique* point in U satisfying $V(x^*) = 0$. In order to show that V is a valid Lyapunov function, we must identify a control law such that V is nonincreasing along the trajectories of the system. To that end, derive a closed-form expression for the time derivative $\dot{V}$ of V:

$$
\dot{V} = \frac{d}{dt} [V(x)] = \nabla V(x) \cdot f(x, \tau). \tag{9}
$$

You may leave your answer in terms of the state x and the control τ (to be determined).

The result of part (f) suggests that we consider the following nonlinear feedback controller:

$$
\begin{gathered}
\tau : \mathbb{R}^2 \to \mathbb{R} \\
\tau(x) \triangleq -2\alpha mgl \sin(x_1) \cos(x_1).
\end{gathered} \tag{10}
$$

Our goal now will be to apply Lyapunov theory to show that this indeed stabilizes the upright position $x^* = (\pi, 0)$, and to characterize the *invariant subsets* around x^*.

(g) Using (10), derive an explicit expression for the closed-loop dynamics of the system:

$$
\dot{x} = c(x) \triangleq f(x, \tau(x)) \tag{11}
$$

under the control law (10).

(h) Using your result in part (g), find the set S of stationary points for the closed-loop system under the control law (10).

(i) Show that under the control law (10), the function V satisfies:

$$
\dot{V} \leq 0. \tag{12}
$$

Also, what is your conclusion about the stability of the stationary point $x^* = (\pi, 0)$ under the nonlinear control law (10)?