Principles of Robot Autonomy I
Homework 4
Due Thursday, November 14th (5:00pm)

Relevant links are listed for each problem.

You will submit your homework to Gradescope. Your submission will consist of a single pdf with your answers for written questions and relevant plots from code.

Your submission must be typeset in LaTeX.

Problem 1: Bundle Adjusment SLAM in 2D

In this problem you will use a standard open source SLAM package, GTSAM, to perform a bundle adjustment SLAM. Recall, bundle adjustment SLAM solves for both landmark locations and robot poses at the same time. Alternatively, Pose Graph Optimization (PGO) SLAM pre-processes raw sensor measurements in a "front end", and optimizes only robot poses in a "back end." We will use the Python interface to the GTSAM library, which is implemented in C++. We use a sparse 2D problem setup to ensure quick optimization.

1. Draw a simple factor graph by hand or with a computer program, such as PowerPoint, that meets the following criteria.
   - Three pose states x_0, x_1, and x_2.
   - Two landmarks, ℓ_0 and ℓ_1, marked by a 2D position.
   - Four range and bearing measurements as follows:
     - $y^{rb}_{0,0}$: between pose x_0 and landmark ℓ_0
     - $y^{rb}_{0,1}$: between pose x_0 and landmark ℓ_1
     - $y^{rb}_{1,1}$: between pose x_1 and landmark ℓ_1
     - $y^{rb}_{2,1}$: between pose x_2 and landmark ℓ_1
   - Odometry measurements $y^{o}_{i,i+1}$ between successive poses.
   - Loop closure constraint as odometry-like measurement $y^{lc}_{0,2}$ between pose states x_0 and x_2.

Please download the code and necessary packages by running following lines in a terminal window.

```
$ git clone https://github.com/PrinciplesofRobotAutonomy/AA274a-HW4-F24.git
$ cd AA274a-HW4-F24/P1-GTSAM
$ pip install -r requirements.txt
```

We now consider a more complex scenario. We will use the robot trajectory and landmarks in Figure 1. In the file `robot_history_5.csv`, we provide the ground truth history of the robot pose and measurements for a maximum sensor range of 5 meters. We will add noise and reconstruct the robot pose and landmark locations. The following steps will guide you in constructing the factor graph in GTSAM.

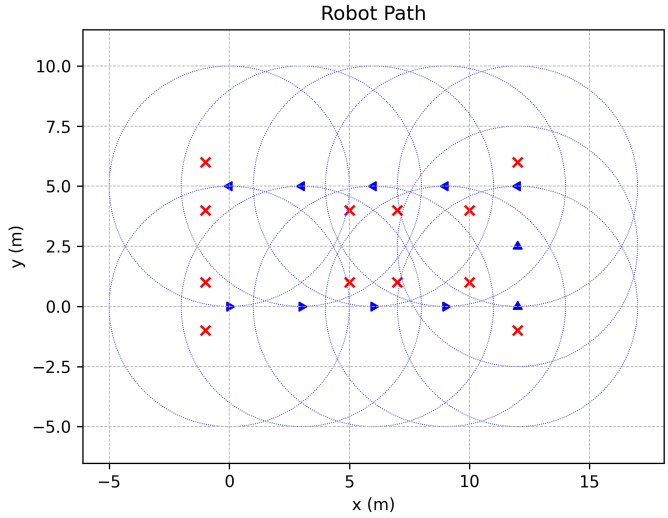


Figure 1: Robot Ground Truth Path. The landmarks are indicated as a red '×', the robot is a blue triangle oriented in the direction of travel, and the dotted blue circles indicate the maximum sensor range.

2. Ensure that the requirements.txt are installed properly, especially GTSAM. Please read through the provided code to load and process the files. Instantiate the factor graph with:

   ```
   graph = gtsam.NonlinearFactorGraph()
   ```

   This is an empty factor graph. To *add* a factor to the factor graph, we will call `graph.add(...)`. To simplify the notation, GTSAM offers shorthand variables `X` and `L` for state and landmark, respectively, to organize the SLAM factors. In the provided notebook, we have initialized lists for `X` and `L`. The first factor we want to add is the prior factor to initialize the map at the origin. Fill in the required code based on the factor description below.

   ```
   gtsam.PriorFactorPose2(X_start, pose, noise)
   ```

   For `X_start`, use the zero-th element of the `X` list. For `pose`, use the `gtsam.Pose2(x, y, theta)` function specifying the origin and zero angle (i.e., aligned with the x-axis).

3. Iterate through the provided data (`Pandas` is recommended). Use the factor type below to add the relative motion factors (i.e., from the odometry or relative state between successive states).

   ```
   gtsam.BetweenFactorPose2(X_prev, X_curr, rel_pose, motion_noise)
   ```

   Use the factor type below to add the bearing and range factors (i.e., from the sensor model) per each landmark in view at the given time step.

   ```
   gtsam.BearingRangeFactor2D(X_curr, L_curr, bearing, range, noise
   ```

   For the bearing measurement, use `gtsam.Rot2.fromDegrees(bearing_angle)`. Lastly, we will use the `BetweenFactorPose2` method to add a loop closure constraint between the second and eighth elements (zero-indexed) of the `X` list with position offset of 0 horizontally, 5 vertically, and 180 degrees in rotation. How many factors are in your factor graph (print `graph` to verify)?

4. We will initialize the values from the motion alone. Instantiate the initial estimate with

   ```
   initial_estimate = gtsam.Values()
   ```

   You can input an initial estimate with `initial_estimate.insert(X_or_L_var, pose_or_point)`. Be sure to use `gtsam.Pose2(x, y, theta)` for the pose and `gtsam.Point2(x, y)` for the landmark.

   Run the following code block to plot your initial estimate data and include the plot in your write-up.

5. Run the Levenberg Marquardt in GTSAM in the notebook. GTSAM will provide estimates of `X` and `L` through the optimizer and will provide the marginal Gaussian distributions in `gtsam.Marginals`. Plot the optimized estimates for `X` and `L` along with the 2D Gaussian Confidence Ellipse (1-sigma). Include the plot in your write-up.

6. What do you notice about the orientation of the solution and the shape and orientation of the confidence ellipses? Why does this occur?

7. Switch out `robot_history_5.csv` for `robot_history_3.csv` (maximum sensor range of 3 meters) and `robot_history_10.csv` (maximum sensor range of 10 meters). Using the same initialization for `X` and `L`, reconstruct the factor graphs. What are the factor graph sizes in each case?

   Hint: You should only have to change the file input. All the other functions you wrote should still be applicable!

8. Running the following code block to plot the optimized estimates for `X` and `L` along with the 2D Gaussian Confidence Ellipse (1-sigma) in each case. What do you notice about the final results at different sensor ranges? Explain your result with the connectivity of the resulting factor graphs.

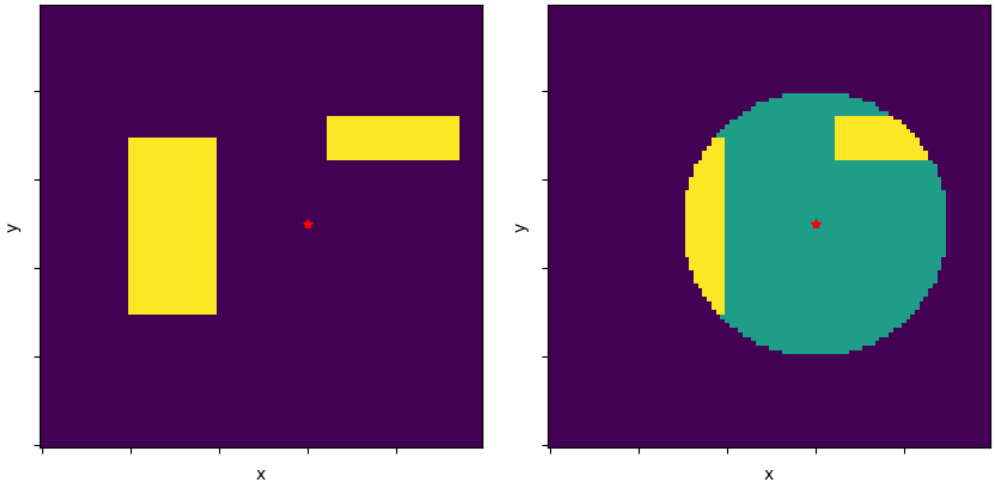
Problem 2: Frontier Exploration

Objective: Implement a frontier exploration algorithm to decide where to travel in a map.

Setup: For this problem, we will use a Google Colab notebook linked here:
https://drive.google.com/file/d/10-1O0KMwEDijUwd42H7OyHF4YJe8NIio/view?usp=sharing.
Please make a copy of this notebook to your drive to make changes.

Map Representation: We will use a Stochastic Occupancy grid that we've used in previous sections to keep track of the states that are known vs. unknown, and occupied vs. unoccupied.

For this problem, when a state in the grid is within the sensing radius of the robot, the state is known. When the state is known, it is assigned with a probabilistic value of whether there is an object occupying that grid cell. These probabilities range between 0 and 1, and for the purposes of this question, we will use the heuristic that a cell is considered occupied if the probability of occupancy is greater than or equal to 0.5. If a cell is unknown, (has not been sensed), then the value of the cell is -1.



(a) Ground Truth Map with obstacles (b) Stochastic Occupancy Map

Figure 2: Obstacles are shown in yellow, the current state of the robot is marked with a red star.

Exploration Heuristics: With a given Stochastic Occupancy map, we want to decide what states to explore to balance gathering new information (traveling to unknown states), while remaining in regions where we don't believe there are any obstacles (traveling to unoccupied states). To balance these two things, we will look for states that satisfy the following conditions:

1. The percentage of unknown cells surrounding a cell in some window should be greater than or equal to 20% of the surrounding cells.
2. The number of known, occupied cells surrounding a cell in some window should be 0.
3. The percentage of known, unoccupied cells surrounding a cell in some window should be greater than or equal to 30% of the surrounding cells.

Cells that satisfy these three heuristics will be considered valid cells to explore the frontier. A visualization of an example for defining these windows is shown in Figure 3. Note that the heuristics in the simple example shown in Figure 3 are different from the heuristics listed in the problem statement.

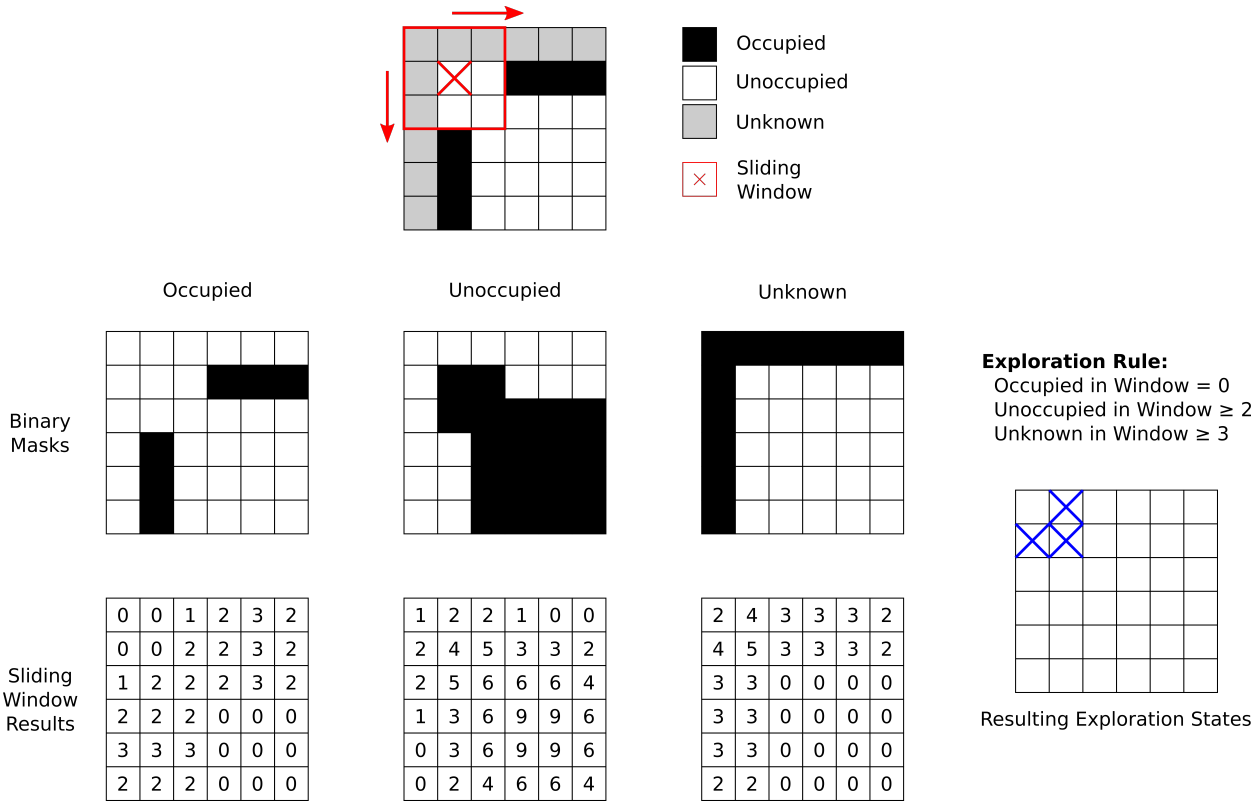


Figure 3: Visualization of using Convolutions for Heuristics for a sample problem

(i) 💻 Write the `explore` function in the notebook according to these heuristics and return the resulting plot, showing the robot's Stochastic Occupancy Map and the cells that satisfy the exploration heuristic.

(ii) 💻 We will choose a state from the valid frontier states that is closest to the current state of the robot. What is the euclidean distance of the closest frontier state to the robot's current state? Include this calculation in your `explore` function, and report the distance value in your write-up submission.

[Section Prep]: Frontier Exploration

Note: This portion of the homework is **not graded**, but should be completed before Section the week of 11/18 to test in hardware. All the URLs are highlighted in blue. Make sure you click on them as they are important references and documentation!

Please also run `update_tb_ws` (anywhere in your terminal) somewhat frequently. The provided library code asl-tb3-utils from the course staff is under rapid development changes.

Objective

In this assignment, you will practice writing ROS2 node and launch files from scratch, while adding autonomous exploration capabilities to the turtlebot stack you have been building throughout the quarter. **The requirements are as follows**,

1. Everything should be integrated into one single launch file, similar to the `navigator.launch.py` you wrote from HW2. Note that the launch file **does not** need to launch either the simulator or the hardware bringup on the turtlebot.
2. On launch, your robot should start exploring the map autonomously until the closed environment is mostly mapped out. The exploration does not have to end with a perfectly watertight map but there should not be any large unexplored regions inside the closed environment.
3. The robot should not run into any obstacles any time during the exploration.
4. The launch should also include RViz to visualize the reconstructed map in real time.
5. The autonomous exploration logic needs to be its own standalone ROS2 node.

The solution to this problem is quite open-ended as you have a lot of freedom when designing your own ROS2 node. We have provided the following guidelines to help you build this.

ROS2 Node

1. In your frontier exploration node constructor, you need to create various subscribers and publishers to interface with the navigator. Take a closer read at the base navigator class to see what topics are being subscribed and published.
2. Adding to the previous point, the navigator publishes a boolean message to the `/nav_success` topic. A true message is published when the robot reaches the commanded navigation pose, and a false message is published when planning (or re-planning) fails. Your exploration node needs to listen to this topic to figure out when to send out the next navigation command.
3. Code from Problem 2 can be ported into this ROS2 node with minimal edits. Make sure your code works well in the notebook from Problem 2 before testing this in the ROS2 stack as it will be harder to debug with everything running together.
4. You might want to subscribe to `/state` and `/map` topics to receive the latest robot pose and map information for calculating eligible frontier locations.
5. `StochOccupancyGrid2D` can be useful for processing map data. The base navigator class gives an example of how to import this class and construct an occupancy object given a map message.

ROS2 Launch File

1. The launch file should be largely similarly to `navigator.launch.py` from HW2.
2. Add your frontier exploration node into your launch file.
3. It may or may not be helpful to delay launching your exploration node with `launch.actions.TimerAction`. An example on how to use it can be found here.

Testing in Simulation

You can test your implementations in several simulated maps. Up to this point, you should be familiar with launching the simulator. Here are some suggested sim environments to test with:

1. `ros2 launch asl_tb3_sim root.launch.py`
2. `ros2 launch asl_tb3_sim maze.launch.py`