Principles of Robot Autonomy: Homework 04

Purushotham Mani
11/12/24

Other students worked with: None
Time spent on homework: approx. 2.5 Hrs excluding report

Problem 1: Bundle Adjusment SLAM in 2D

- Simple factor graph:

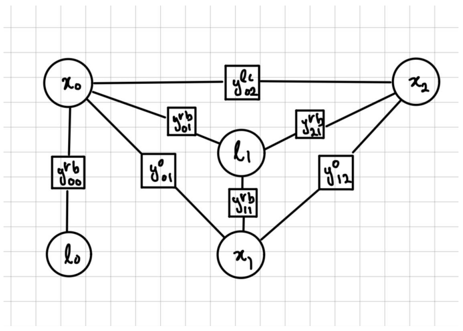


Figure 1: Simple Factor Graph

- Initial estimate data:

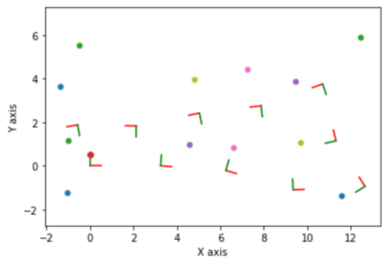
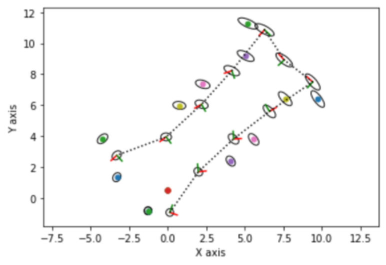
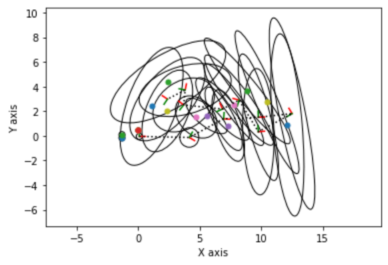


Figure 2: Initial Estimates

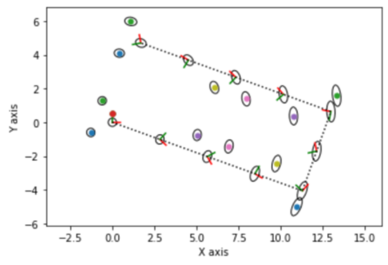
- Optimized estimates for X and L along with the 2D Gaussian Confidence Ellipse (1-σ)



(a) robot_history_5



(b) robot_history_3



(c) robot_history_10

- The orientation of the solution is at an angle from the ground truth. This is because of the noise in the sensor readings and because the factors only contain relative information between poses and not global information.

  That is why you can see in robot_history_10 that the optimized estimates of X and L are almost perfect except for the orientation. One way we could solve this issue is if we had added the prior factor for more than just initial pose.

  The shape and orientation of the gaussian is such that the minor axis of the confidence ellipse points towards the initial pose. Indicating that it's more certain in that direction.

  This is because of the loop closure constraint and because the factors store relative information between poses (meaning they are more certain in the direction of sensor measurement).

  You can also notice that the confidence ellipse are larger farther away from the initial pose.

- Factor Graph Size: · robot_history_5 : 60 · robot_history_3 : 32 · robot_history_10 : 114

- With a smaller sensor range, the SLAM system will have fewer observations per time step, resulting in lower connectivity in the factor graph. Consequently, the position estimates may be less accurate, and confidence ellipses as seen in the figure will be larger, reflecting increased uncertainty.

  An increased sensor range (e.g., in the 10-meter scenario) increases the number of factors (edges) in the graph, more landmarks are visible from each pose, increasing the connectivity in the factor graph, effectively improving the system's accuracy by adding more redundancy and reducing uncertainty.

Problem 2: Frontier Exploration

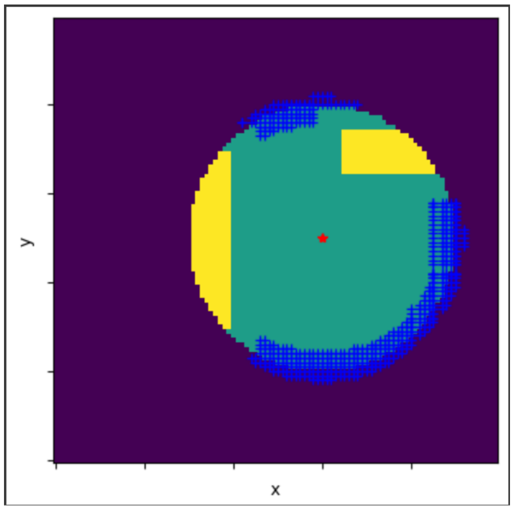


Figure 4: Frontier States

- Euclidean distance to the closest frontier state = 2.596

Appendix A: Code Submission

0.1 Problem 1: Bundle Adjusment SLAM in 2D

0.1.1 Building the factor graph

```python
# Re-import here so that you can rerun this cell to debug without rerunning
# everything
from gtsam.symbol_shorthand import L, X

# Create an empty nonlinear factor graph
### TODO (Q4.b) ###
graph = gtsam.NonlinearFactorGraph()
###

Xs = [X(i) for i in range(n_time_steps)]
Ls = [L(i) for i in range(n_landmarks)]

# Add a prior factor at the origin to set the origin of the SLAM problem
### TODO (Q4.b) ###
# graph.add(...)
prior_factor = gtsam.PriorFactorPose2(Xs[0],gtsam.Pose2(0,0,0),PRIOR_NOISE)
graph.add(prior_factor)
###

### TODO (Q4.c) ####
for j in range(n_landmarks):
    if not np.isnan(robot_noisy_df[f"bearing_{j}"].iloc[0]):
        bearing = gtsam.Rot2.fromDegrees(np.rad2deg(robot_noisy_df[f"bearing_{j}"].iloc[0]))
        f2 = gtsam.BearingRangeFactor2D(Xs[0], Ls[j], bearing, robot_noisy_df[f"range_{j}"].iloc[0],
        ↪  MEASUREMENT_NOISE)
        graph.add(f2)

for i in range(1,n_time_steps):
    x_c,y_c,theta_c = robot_noisy_df[["x", "y", "theta"]].iloc[i].to_numpy()
    x_p,y_p,theta_p = robot_noisy_df[["x", "y", "theta"]].iloc[i-1].to_numpy()
    f1 = gtsam.BetweenFactorPose2(Xs[i-1], Xs[i], gtsam.Pose2(x_c-x_p,y_c-y_p,theta_c-theta_p),
    ↪  ODOMETRY_NOISE)
    graph.add(f1)
    for j in range(n_landmarks):
        if not np.isnan(robot_noisy_df[f"bearing_{j}"].iloc[i]):
            bearing = gtsam.Rot2.fromDegrees(np.rad2deg(robot_noisy_df[f"bearing_{j}"].iloc[i]))
            f2 = gtsam.BearingRangeFactor2D(Xs[i], Ls[j], bearing, robot_noisy_df[f"range_{j}"].iloc[i],
            ↪  MEASUREMENT_NOISE)
            graph.add(f2)


f3 = gtsam.BetweenFactorPose2(Xs[1],Xs[7],gtsam.Pose2(0,5,np.deg2rad(180)), ODOMETRY_NOISE)
graph.add(f3)
###

# Print the factor graph to see all the nodes
### TODO (Q4.c) ####
print(graph)
###
```

0.1.2 Initialize the optimization problem

```python
### TODO (Q4.d) ###
initial_estimate = gtsam.Values()
###
### TODO (Q4.d) ###
for i in range(n_time_steps):
    x,y,theta = robot_noisy_df[["x", "y", "theta"]].iloc[i].to_numpy()
    initial_estimate.insert(Xs[i], gtsam.Pose2(x,y,theta))
###
l_init_vec = [
    (-1, -1),
    (-1, 1),
    (5, 1),
    (7, 1),
    (10, 1),
    (12, -1),
    (12, 6),
    (10, 4),
    (7, 4),
    (5, 4),
    (-1, 4),
    (-1, 6)
]
for l_ind, L in enumerate(Ls):
    l_hat = l_init_vec[l_ind]
    point_init = (np.random.normal(l_hat[0], ODOMETRY_NOISE_NUMPY[0]),
                  np.random.normal(l_hat[1], ODOMETRY_NOISE_NUMPY[0]))
    ### TODO (Q4.d) ###
    initial_estimate.insert(L, gtsam.Point2(point_init[0],point_init[1]))
    ###
### TODO (Q4.d) ###
print(initial_estimate)
###
```

0.1.3 Graph Optimization

```python
lm_params = gtsam.LevenbergMarquardtParams()
### TODO (Q4.e)  ###
optimizer = gtsam.LevenbergMarquardtOptimizer(graph, initial_estimate,lm_params)
result = optimizer.optimize()
### TODO (Q4.e) ###
print(result)
###

### TODO (Q4.e) ###
marginals = gtsam.Marginals(graph, result)
for x_ind, X in enumerate(Xs):
    print(x_ind," ",marginals.marginalCovariance(X))

for l_ind, L in enumerate(Ls):
    print(l_ind," ",marginals.marginalCovariance(L))
###
```

0.2 Problem 2: Frontier Exploration

```python
def explore(occupancy):
    """ returns potential states to explore
    Args:
        occupancy (StochasticOccupancyGrid2D): Represents the known, unknown, occupied, and unoccupied
        ↪  states. See class in first section of notebook.

    Returns:
        frontier_states (np.ndarray): state-vectors in (x, y) coordinates of potential states to explore.
        ↪  Shape is (N, 2), where N is the number of possible states to explore.
    """

    window_size = 13    # defines the window side-length for neighborhood of cells to consider for
    ↪  heuristics
    ########################### Code starts here ###########################
    occupied_mask = np.where(occupancy.probs >= 0.5, 1, 0)
    unknown_mask = np.where(occupancy.probs == -1, 1, 0)
    unoccupied_mask = np.where((occupancy.probs < 0.5) & (occupancy.probs>=0), 1, 0)

    kernel = np.ones((window_size, window_size)) / window_size**2
    occupied=convolve2d(occupied_mask, kernel, mode='same')
    unoccupied=convolve2d(unoccupied_mask, kernel, mode='same')
    unknown=convolve2d(unknown_mask, kernel, mode='same')

    frontier_mask = np.where((occupied==0) & (unoccupied>=0.3) & (unknown>=0.2),1,0)
    frontier_states = np.transpose(np.nonzero(np.transpose(frontier_mask)))
    frontier_states = occupancy.grid2state(frontier_states)

    print((np.min(np.linalg.norm(frontier_states-current_state,axis=1))))
    ########################### Code ends here ###########################
    return frontier_states

```