Principles of Robot Autonomy: Homework 4

[Chetan Reddy] [Narayanaswamy]

11/14/2024

Other students worked with: None
Time spent on homework: 6 hours

Problem 1:

Part (1)

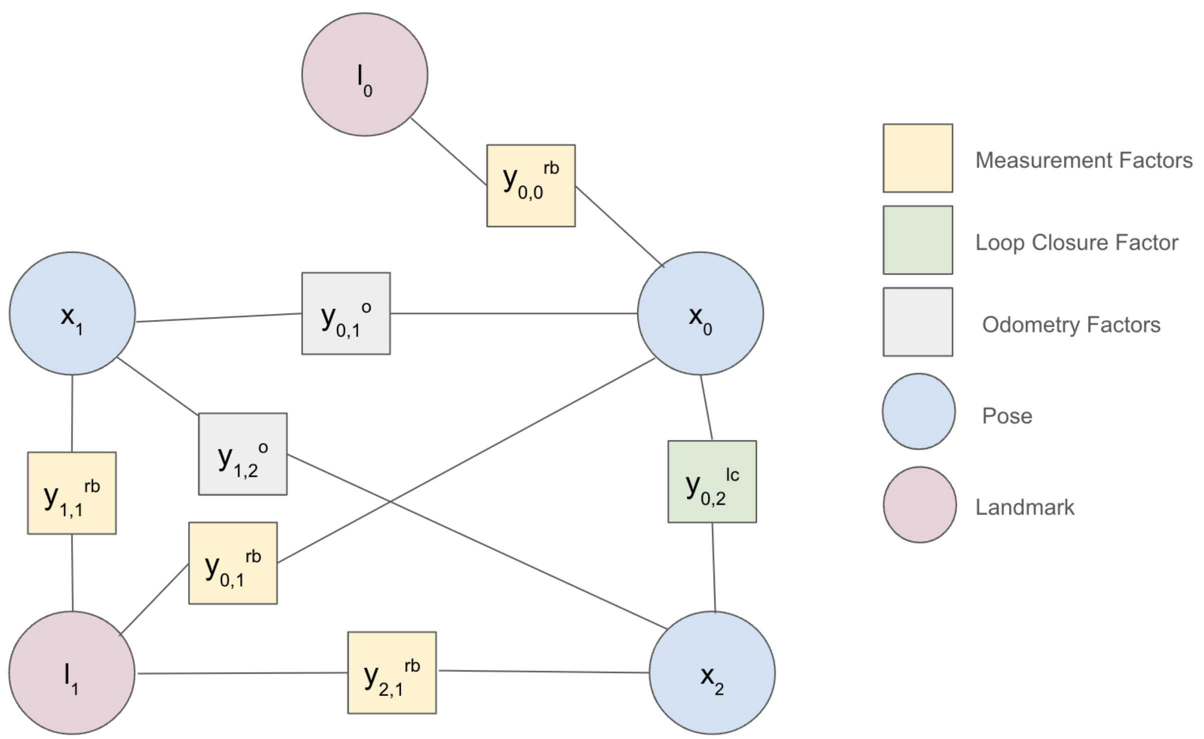


Figure 1: Factor Graph

- The measurement factors in yellow will be dependent on the measurements: $y_{0,0}^{rb}, y_{0,1}^{rb}, y_{1,1}^{rb}, y_{2,1}^{rb}$
- The odometry factors in gray will be dependent on $y_{i,i+1}^{o}$
- The loop closure factor (which is just like an odometry factor) will be dependent on $y_{0,2}^{lc}$

Part (2)

Shown in Code

Part (3)

There are **60 factors** in the factor graph for the *robot_history_5.csv* file.

Part (4)

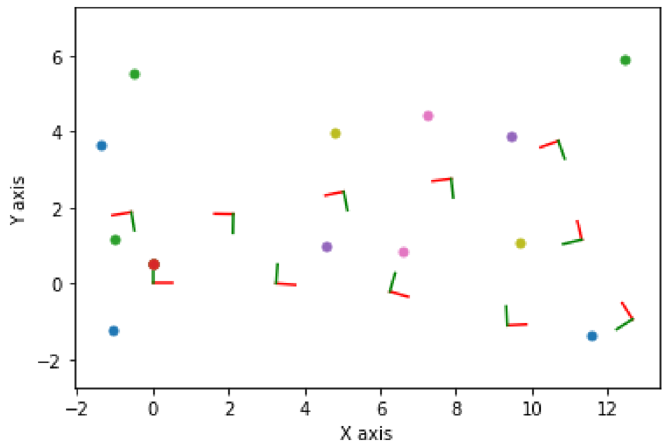


Figure 2: Initial Estimate of the Poses and Landmarks

Part (5)

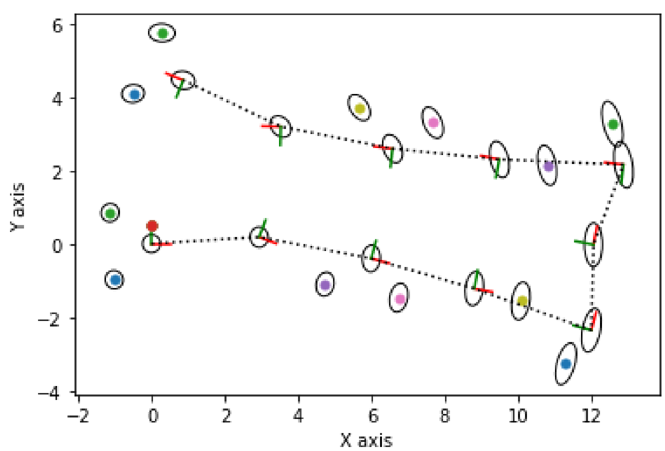


Figure 3: Optimised Estimates for X and L along with 1-sigma confidence ellipses

Part (6)

The estimates seem to be tilted from the true values. This is perhaps because of the noisy initial orientation estimate which accumulates and drifts from the true values. In the code, when the noise is added, the orientation noise in ODOMETRY_NOISE is fairly high as well. The ellipses are somewhat stretched along the y-axis depicting more uncertainty.

Increasing the number of loop closures or the factors will fix this issue as seen later with sensor range = 10.

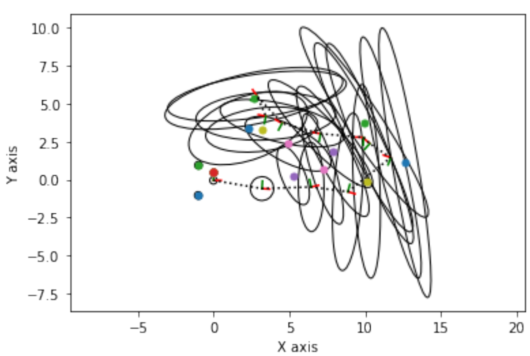
Part 7

The factor graph sizes are as follows

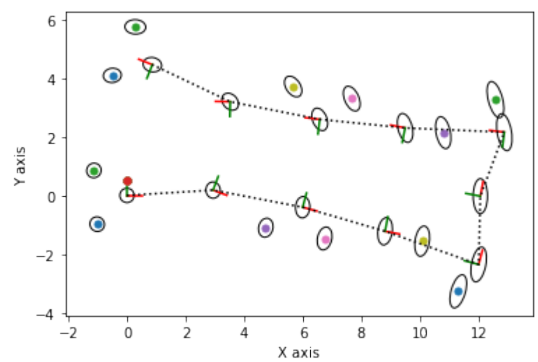
- *robot_history_3.csv*: 32
- *robot_history_5.csv*: 60
- *robot_history_6.csv*: 76
- *robot_history_10.csv*: 114

Part (8)

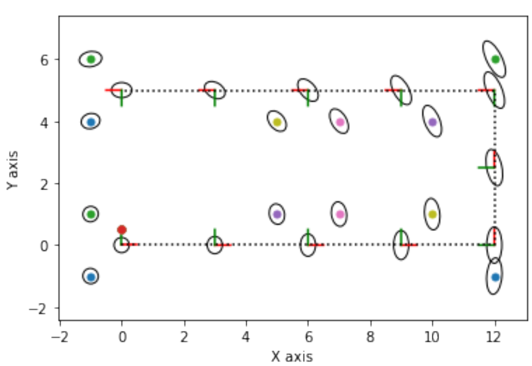
The accuracy of reconstruction clearly increases as the number of factors are increased as there is more data flowing in. In this case, the number of factors increases when the sensor range is made higher as this would detect more landmarks.



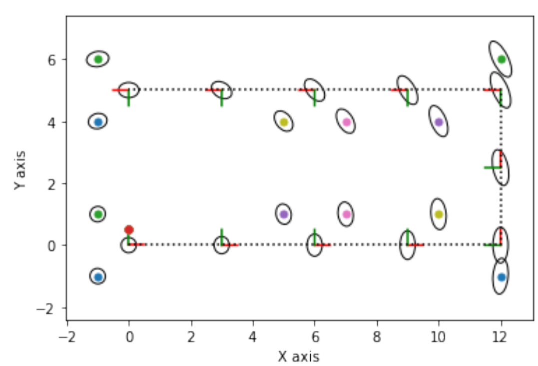
(a) Sensor Range = 3 metres



(b) Sensor Range = 5 metres



(c) Sensor Range = 6 metres



(d) Sensor Range = 10 metres

Figure 4: Optimised Estimates of Pose and Landmarks for different Sensor Rangees

Problem 2:

Part (i)

Shown in code

Part (ii)

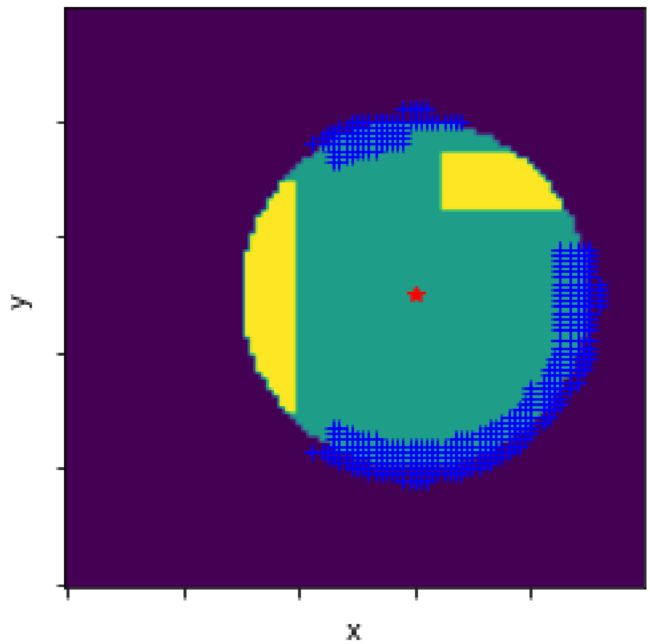


Figure 5: Frontier States shown in blue

The euclidean distance of the closest frontier state = **2.5961**

Appendix A: Code Submission

Problem 1

```
### TODO (Q1 Part 7) ###
robot_gt_df = pd.read_csv('robot_history_5.csv') # this is GROUND TRUTH (gt)!
# robot_gt_df = pd.read_csv('robot_history_3.csv') # this is GROUND TRUTH (gt)!
# robot_gt_df = pd.read_csv('robot_history_10.csv') # this is GROUND TRUTH (gt)!

###
```

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

# Create an empty nonlinear factor graph
### TODO (Q1. part 2) ###
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 (Q1. part 2) ###
pose = gtsam.Pose2(0, 0, 0)
prior_factor = gtsam.PriorFactorPose2(Xs[0], pose, PRIOR_NOISE)
graph.add(prior_factor)

###

# Loop through the noisy data (i.e., `robot_noisy_df`)
# In the loop,
#    get the relative motion with pose_{i} - pose_{i - 1}
#    the measurements are in the noisy data
# Don't forget the loop closure constraint
######### TODO (Q1. part 3) #############
for i in range(0,n_time_steps):
    row_prev = robot_noisy_df.iloc[i-1]
    row_present = robot_noisy_df.iloc[i]

    ### Adding Odometry factor
    if i>=1: # Adding odometry factor from the 1st element (by zero index)
        rel_pose = gtsam.Pose2(row_present.x-row_prev.x, row_present.y - row_prev.y, np.deg2rad(row_present.theta - row_prev.theta))

        X_prev = Xs[i-1]
        X_curr = Xs[i]
        relative_motion_factor = gtsam.BetweenFactorPose2(X_prev, X_curr, rel_pose, ODOMETRY_NOISE)
        graph.add(relative_motion_factor)

    ### Adding Landmark related factors

    ## Retrieving Landmarks within the range of the robot which have non-nan values
```

```python
    for j in range(n_landmarks):
        range_j = row_present["range_{}".format(j)]
        if not pd.isna(range_j):
            X_curr = Xs[i]
            L_curr = Ls[j]
            bearing_angle_j  = row_present["bearing_{}".format(j)]
            bearing_j = gtsam.Rot2.fromDegrees(bearing_angle_j) #Bearing for landmark j

            measurement_factor = gtsam.BearingRangeFactor2D(X_curr, L_curr, bearing_j, range_j, MEASUREMENT_NOISE)
            graph.add(measurement_factor)



LOOP_CLOSURE_NOISE = ODOMETRY_NOISE

X_loop1 = Xs[2]
X_loop2 = Xs[8]
rel_pose_loop = gtsam.Pose2(0,5,np.deg2rad(180))
loop_closure_factor = gtsam.BetweenFactorPose2(X_loop1,X_loop2,rel_pose_loop, LOOP_CLOSURE_NOISE)
graph.add(loop_closure_factor)



################################################

# Print the factor graph to see all the nodes
### TODO (Q1. part 3) ####
print(graph)

###
```

```python
# Set-up a values data structure for the initial estimate
### TODO (Q1. part 4) ###
initial_estimate = gtsam.Values()
###

# Set the initial poses from the noisy odometry alone
# Hint, you already have this in `robot_noisy_df`
### TODO (Q1. part 4) ###
X_init_mat = robot_noisy_df[["x","y","theta"]].values
for x_ind, X in enumerate(Xs):
    X_hat = X_init_mat[x_ind]
    X_pose = gtsam.Pose2(*X_hat)
    initial_estimate.insert(X, X_pose)

###

# Sample random values for the initial landmark positions
# In reality, you would have to estimate these from odometry,
# but to not over-complicate the problem, just use noisy
# ground-truth.
# In reality, the initialization is very important for the
# graph optimization to converge to a good solution!!
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]))
    # Add the initial estimates of the landmarks
    ### TODO (Q1. part 4) ###
 ###
    L_point = gtsam.Point2(*point_init)
    initial_estimate.insert(L, L_point)

    ###

# Print the initial estimates to verify
### TODO (Q1. part 4) ###
 ###
print(initial_estimate)
###
```

```
lm_params = gtsam.LevenbergMarquardtParams()

# uncomment the two lines below
### TODO (Q1. part 5)  ###
optimizer = gtsam.LevenbergMarquardtOptimizer(graph, initial_estimate,lm_params)
result = optimizer.optimize()
###

# Print the results
### TODO (Q1. part 5) ###
# Plot the initial poses and landmarks.
for x_ind, x_key in enumerate(Xs):
    gtsam_plot.plot_pose2(0, result.atPose2(x_key), 0.5)

for l_ind, l_key in enumerate(Ls):
    gtsam_plot.plot_point2(0, result.atPoint2(l_key), 0.5)


plt.axis('equal')
plt.show()
###
```

```
### TODO (Q1. part 5) ###
marginals = gtsam.Marginals(graph,result)
print(marginals.marginalCovariance(Xs[1]))
print(marginals.marginalCovariance(Ls[1]))
```

```


###
```

Problem 2

```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.

    HINTS:
    - Function `convolve2d` may be helpful in producing the number of unknown, and number of occupied states in a window of a specified cell
    - Note the distinction between physical states and grid cells. Most operations can be done on grid cells, and converted to physical states at the end of the function with `occupancy.grid2state()`
    """

    window_size = 13    # defines the window side-length for neighborhood of cells to consider for heuristics
    ########################### Code starts here ###########################
    unknown_binary = (occupancy.probs==-1).astype(int)
    occupied_binary = (occupancy.probs>=0.5).astype(int)
    unoccupied_binary = (occupancy.probs<0.5).astype(int)*(occupancy.probs>=0).astype(int)

    mask = np.ones((window_size,window_size))
    unknown_nums = convolve2d(unknown_binary,mask,mode='same', boundary='fill')
    occupied_nums = convolve2d(occupied_binary,mask,mode='same', boundary='fill')
    unoccupied_nums= convolve2d(unoccupied_binary,mask,mode='same', boundary='fill')

    frontier_mask1 = unknown_nums>=0.2*window_size*window_size
    frontier_mask2 = occupied_nums==0
    frontier_mask3 = unoccupied_nums>=0.3*window_size*window_size
    frontier_mask = frontier_mask1*frontier_mask2*frontier_mask3

    # print(np.where(occupied_nums)[0][:10])
    # print(np.where(occupied_nums)[1][:10])

    frontier_states_indices = np.array(np.where(frontier_mask)).T
    frontier_states_indices[:,[0,1]] = frontier_states_indices[:,[1,0]]
    # frontier_states_indices[:,1] = occupancy.se
    frontier_states = occupancy.grid2state(frontier_states_indices)

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

```

# Call to explore function
state_xy = explore(occupancy)
grid_xy = occupancy.state2grid(state_xy)

# Plot Stochastic Occupancy grid with frontier to explore
fig,ax = plt.subplots(1)
ax.imshow(occupancy.probs, origin='lower')
ax.plot(current_state[0]/resolution, current_state[1]/resolution, 'r*')
ax.plot(grid_xy[:,0], grid_xy[:,1], 'b+')
ax.set_ylabel('y')
ax.set_xlabel('x')
ax.set_yticklabels([])
ax.set_xticklabels([])
plt.show()
```