Project 4 - Landmarks, Data Association, and SLAM

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. Triangulation

A. Projecting Into Image Space

Question: What are the camera matrices P_a and P_b? I will accept either the final matrix, or the matrix written in terms of its component matrices (the intrinsic and extrinsic matrices), as long as these are explicitly defined.
Answer: P_a and P_b are made of the product of *intrinsic matrix* K and the *extrinsic matrix* $[R|t]$. K consists the *focal length* f and the translation between the image coordinate and the Camera's coordinate. $[R|t]$ consists of the transformation of the camera coordinate and the world coordinate.

$$P_i = K[R_i|t_i]$$

i can be a or b. P_a and P_b share the intrinsic matrix K, but they have their own extrinsic matrices $[R_i|t_i]$. The function

```
def get_projected_point(P, X):
    X = np.append(X, 1.)
    T =  P @ X
    return T/T[2]
```

Fig. 1: Function *get_projected_point*

$get\_projected\_point(P, X)$ is implemented as shown in Fig. 1. The function takes in a camera matrix P and a 3D scene point X, and return the 2D position of the point on the image. Those returned points are shown in Fig. 2.

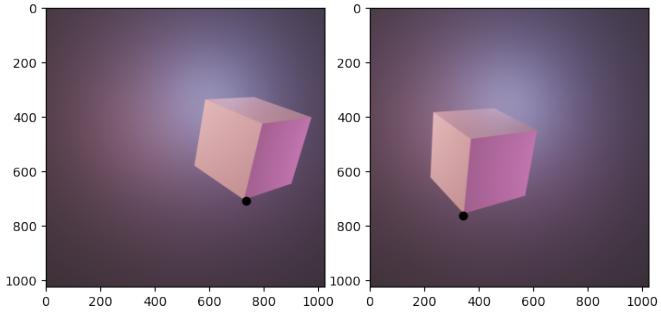

Fig. 2: The 2D projected points

B. Determining the Size of the Cube

Task: Pick two corners of the cube and include the (x, y) image coordinates for both $image\_a$ and $image\_b$ and the 3D world coordinate (X, Y, Z) in your writeup

Answer: We pick two corners on image_a $(p1\_a, p2\_a)$ and two corners on image_b $(p1\_b, p2\_b)$ at the position:

- p1_a = [970,403]
- p2_a = [603,450]
- p1_b = [905,655]
- p2_b = [565,672]

We built a function *get_3D_coordinate* to do the transform from 2D (image) to 3D (the world coordinate), which is shown in Fig. 3. We calculate each row of matrix A by multiplying the points and two camera matrices. We use the function *np.linalg.svd* to calculate the smallest eigenvector, which is the 3D point in homogeneous coordinate. To convert it back to Euclidean coordinate, we divide the whole vector for the last element $v[3, 3]$, and return the first three elements.

```
def get_3D_coordinate(Pa, Pb, point_a, point_b):
    row1 = point_a[1]*Pa[2] - Pa[1]
    row2 = Pa[0] - point_a[0]*Pa[2]
    row3 = point_b[1]*Pb[2] - Pb[1]
    row4 = Pb[0] - point_b[0]*Pb[2]
    A = np.array([row1, row2, row3, row4])
    _,_, v = np.linalg.svd(A, full_matrices=True)
    return v[3]/v[3,3]
```

Fig. 3: The code

The 3D coordinates of the two corners are:

- p1 = [1.87269315, -0.2496902, -0.21355011]
- p2 = [1.74420064, 0.77509012, 0.20990598]

Question: What is the side length of the cube shown in the two images above?
Answer: The side length is approximate 1.116 and the cube volume is $V = 1.39$

C. Incorporating Noise (Pt. 1)

Question: Describe a procedure for how you would compute the mean and covariance of the 3D position of the corner of the cube given the noise model defined above.
Answer: With assuming that the standard deviation of 5 pixel, we make procedure as shown in Fig. 4. We generate a number of samples with the input points as the mean values and the standard deviation is 5 pixel. After getting the sample, input it function *get_3D_coordinate* to calculate the corresponding 3D point, which is appended to a list. From the list, we can calculate the mean value, and covariance of the 3D point by two numpy functions $np.mean$ and $np.cov$.

```python
def get_mean_cov_3D(Pa, Pb, m1, m2):
    coor_3D = []
    for jj in range(50):
        x1 = m1[0] + random.randint(-5, 5)
        y1 = m1[1] + random.randint(-5, 5)
        x2 = m2[0] + random.randint(-5, 5)
        y2 = m2[1] + random.randint(-5, 5)

        X = get_3D_coordinate(Pa, Pb, [x1,y1], [x2,y2])
        coor_3D.append(X[:3])
    coor_3D = np.array(coor_3D).T
    mu_3D = np.mean(coor_3D, axis=1)
    cov_3D = np.cov(coor_3D)
    return mu_3D, cov_3D
```

Fig. 4: get_mean_cov_3D function

The first and second corner coordinates and their variance are shown below:

```
The mean of the first corner:
[ 1.86825523 -0.24809141 -0.22272294]
The covariance of the first corner:
[[ 2.01590787e-04 -6.15369421e-05  6.38882260e-04]
 [-6.15369421e-05  1.22777889e-04 -3.13473245e-04]
 [ 6.38882260e-04 -3.13473245e-04  3.03573355e-03]]
```

Fig. 5: The first corner

Question: Why might the mean of this result differ from the value you computed in the previous question?
Answer: The coordinate means should be different from the values from the last part. It is caused by a set of random points in 2D coordinate generating with the standard deviation of 5 pixels. In this function, we determine the mean and its covariance from a set of sample, not from a single point.

```
The mean of the second corner:
[1.74334207 0.77385069 0.19595061]
The covariance of the second corner:
[[0.00039142 0.00021111 0.00130572]
 [0.00021111 0.00020566 0.00088397]
 [0.00130572 0.00088397 0.00598238]]
```

Fig. 6: The second corner

D. Incorporating Noise (Pt.2)

Question: Describe a procedure for how you would compute the mean and variance of the length of a side of the cube from observations of two of the corners and the noise model from above.
Answer: From the means and covariances of the two 3D points, we can calculate the mean distance and its variance. The mean distance $d(m1, m2)$ is the euclidean distance between the two mean points, and the variance is determined by the formula:

$$Var(d) = (\frac{\delta d}{\delta x_1})^2\sigma_{x_1}^2 + (\frac{\delta d}{\delta y_1})^2\sigma_{y_1}^2 + (\frac{\delta d}{\delta z_1})^2\sigma_{z_1}^2 + \\ + (\frac{\delta d}{\delta x_2})^2\sigma_{x_2}^2 + (\frac{\delta d}{\delta y_2})^2\sigma_{y_2}^2 + (\frac{\delta d}{\delta z_2})^2\sigma_{z_2}^2$$

The detail of the function is shown in Fig. 7.

The mean and and (1D) variance for the length of the cube are:

```python
def get_mean_var_distance(m1, cov1, m2, cov2):
    dis_mu = np.sqrt(np.sum( np.square(m_p1-m_p2) for (m_p1, m_p2) in zip(m1, m2)) )
    var1 = 2*np.square(m1[0] - m2[0])*(np.square(cov1[0,0]) + np.square(cov2[0,0]))
    var2 = 2*np.square(m1[1] - m2[1])*(np.square(cov1[1,1]) + np.square(cov2[1,1]))
    var3 = 2*np.square(m1[2] - m2[2])*(np.square(cov1[2,2]) + np.square(cov2[2,2]))
    var = np.sqrt((var1+var2+var3)/np.square(dis_mu))
    return dis_mu, var
```

Fig. 7: get_mean_var_distance

- μ = 1.21
- The variance = 0.0034

II. Simple Simultaneous Localization and Mapping

A. Motion Only

As adding the robot poses and motion to GTSAM (no landmarks), and the robot motion is shown in Fig. 8. We can see that at the end of the trajectory, the uncertainty of the robot position is so huge.

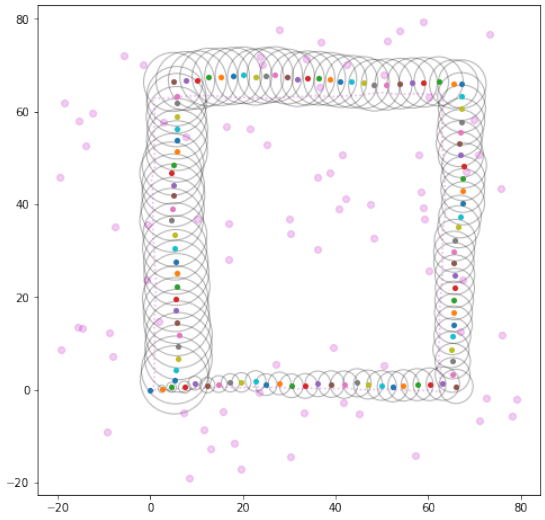

Fig. 8: The robot motion without landmarks

B. Adding Landmarks

Question: What type of factors relate the landmarks and the poses?
Answer: We observe the landmarks from the current robot position, thus the related factors between them are *BetweenFactorPoint2*. The position difference between them are the amount stored in observation.
Question: Are there factors (edges in the graph) between landmarks? If so, what are they? If not, explain why
Answer: In my opinion, there is no factors between landmarks. The landmarks are observed by the robot's sensor, and determined by the robot pose. Therefore, we need more constraints to connect these landmarks.

After adding the landmarks and their factors to the graph, we can improve the localization and mapping of both robot pose and landmarks position. The result is shown in Fig. 9.
Question: You may notice that the uncertainty over the positions of some of the landmarks is greater or less than the others. Explain why this may be

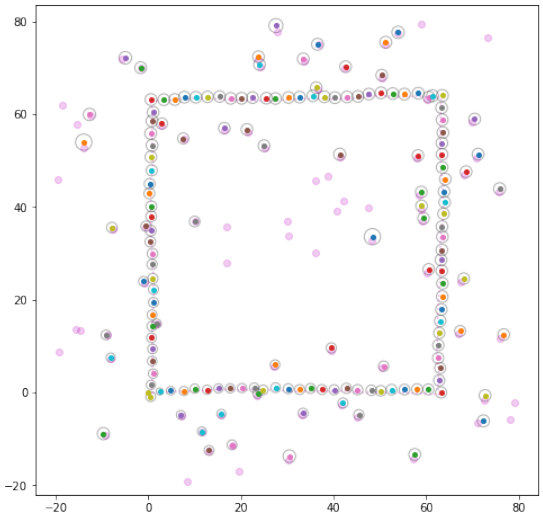

Fig. 9: The robot motion with landmarks

Answer: From the Fig. 9, the landmarks are close to the robot trajectory, the uncertainty is small, and the one are far from the trajectory, the uncertainty get larger. In my opinion, the reason for that is the number of those landmarks being observed. The more time they are observed, there are more factors relating them to the robot poses, so we can improve and update the landmarks' position. It makes the landmarks' position more certainty.

III. Data Association

A. Naive (Gaussian) Data Association

Question: You may notice that a few of the landmarks (magenta circles) have two observed landmarks (black dots) near them (Fig. 10). Explain why this occurs.

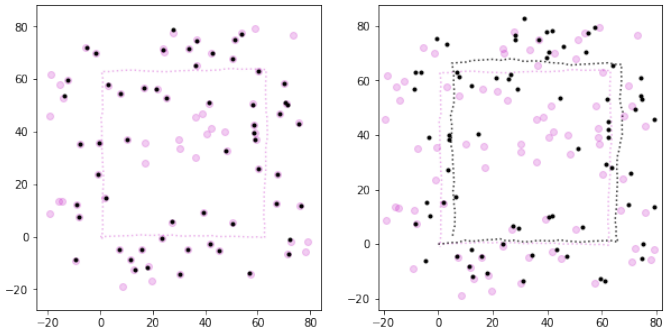

Fig. 10: Naive Data Association

Answer: That happens because of the condition to add a new landmark. As having a new observation, we calculate the probability of that observation overlap the landmark set. If the probability is smaller than a threshold, we will consider it as a new landmark. As the robot travels, and detects the same landmark, due to its motion and sensing errors, it makes the probability is very small. Such that, two observation are added to the landmark set for the same landmark.

Question: Increase the association threshold from 4 to 16 and observe what happens for the case with perfect positions (the left plot) (Fig. 11). Explain what happens and why

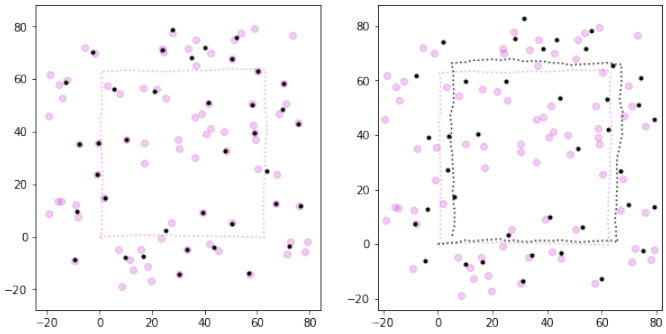

Fig. 11: Naive Data Association - threshold=16

Answer: As we reduce the probability threshold (by increase the association threshold from 4 to 16), we will reduce significantly the chance of making a new landmark. Although the observations are far from the landmarks in the set, with the probability of $np.exp(-16^2/2) = 2.5710^{-56}$ is too small to add a new landmark.

B. Linear Sum Assignment

The code for function *update_landmakrs_lsa* is shown in Fig. 12. We create a *cost_matrix*, and calculate the negative log probability of each observation, and fill into the matrix. Using the function *scipy.optimize.linear_sum_assignment*, we can get the minimum combination for all the rows of the *cost_matrix*. Based on the combination, if a observation is assigned to a landmark, that landmark will gain more detection. If no, a new landmark is created; Then we update the landmark's position.

```
def update_landmarks_lsa(landmark_set, observation,
                         pose, pose_id, association_thresh=4):

    cost_matrix = np.zeros((len(observation), \
                            len(observation)+ len(landmark_set)))
    for ii, obs in enumerate(observation):
        cost_matrix[ii, :len(landmark_set)] = \
              [-np.log(l.prob_of_obs(obs, pose)) for l in landmark_set]
        cost_matrix[ii, len(landmark_set):] = association_thresh**2/2

    row_ind, col_ind = \
             scipy.optimize.linear_sum_assignment(cost_matrix)

    for jj, (row, col) in enumerate(zip(row_ind, col_ind)):
        if col >= len(landmark_set):
            landmark_set.append(Landmark(observation[row], pose, pose_id))
        else:
            landmark_set[col].add_detection(observation[row], pose, pose_id)

    for landmark in landmark_set:
        landmark.update_position()
```

Fig. 12: Function *update_landmakrs_lsa*

Plots: The result of function *update_landmakrs_lsa* is shown in Fig. 13. With the association threshold of 16, with linear sum assignment, we still can make the number of landmarks similar with the function *update_landmakrs* with association threshold of 4. Moreover, for the noisy motion model (the right figure), the landmarks position are mostly overlapped with the

true landmark's position. It means our SLAM with linear sum assignment works very good.

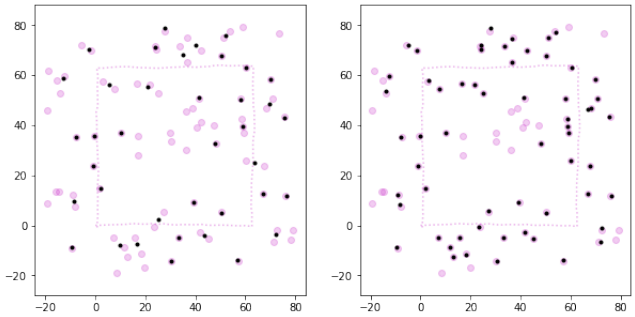

Fig. 13: The landmarks with Hungarian Sum

IV. SLAM WITH NOISY ASSOCIATIONS

The code of the function *update_position_gtsam* is shown in Fig. 14. The first step we create the factors relating the robot's poses based on the motion model. The robot poses are marked by X, and the factors used is $BetweenFactorPoint2$; then the robot's poses are initialized. After that, we handle the landmark's position. A *for* loop runs through the landmarks set, and we initialize the landwark's position by their initial positions. The next important step is to create the factors relating the landmarks and the robot' pose. As the robot moves, it can observe multiple landmarks, and each landmarks can associate with multiple robots poses. To handle those factors, we make another *for* loop inside the third *for* loop to run through all the landmarks's detection. Then we add the factor between the landmark and its related robot's poses.

```
graph.add(gtsam.PriorFactorPoint2(X(0), gtsam.Point2(0, 0), PRIOR_NOISE))
initial_estimate.insert(X(0), gtsam.Point2(0,0))
x_noisy = np.cumsum(np.concatenate(([0], dxs), axis=0))
y_noisy = np.cumsum(np.concatenate(([0], dys), axis=0))

# Add the odometry factors
for ii, (dx_n, dy_n) in enumerate(zip(dxs, dys)):
    graph.add(gtsam.BetweenFactorPoint2(
        X(ii), X(ii+1), gtsam.Point2(dx_n, dy_n), ODOMETRY_NOISE))

# Set the initial estimates for the poses
for ii, (x_n, y_n) in enumerate(zip(x_noisy, y_noisy)):
    initial_estimate.insert(X(ii+1), gtsam.Point2(x_n, y_n))

# Add the observations for the landmarks
for ii, landmark in enumerate(landmarks):
    # Set the initial position of the landmarks
    initial_estimate.insert(L(ii),
          gtsam.Point2(landmark.position[0], landmark.position[1]))
    for jj, obs in enumerate(landmark.detections):
        graph.add(gtsam.BetweenFactorPoint2(X(obs[2]), L(ii),
              gtsam.Point2(obs[0][0], obs[0][1]), LANDMARK_NOISE))
```

Fig. 14: Function *update_position_gtsam*

The result of the algorithm is shown in Fig. 15. Based on only the noisy robot motion and its observations, we can build a map with quite accurate robot's poses and landmark's position. The uncertainty is also small, and similar to the SLAM with known landmark's indices.

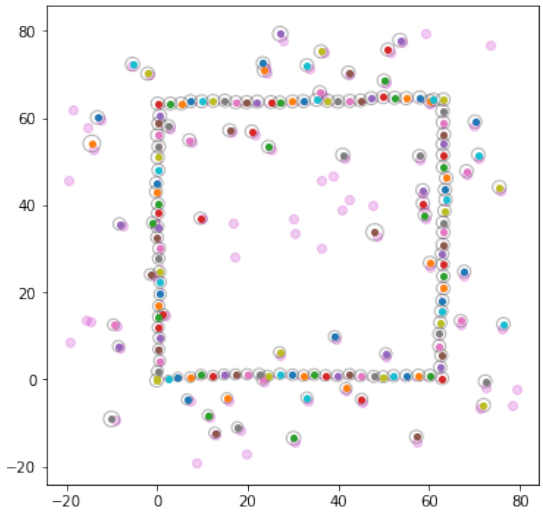

Fig. 15: SLAM and noisy Associations