Kashirin_Aleksandr_PS3

March 11, 2022

Perception in Robotics

PS3: SLAM

Kashirin Aleksandr

Task 1: Prerequisites to build SAM with known DA

In order to check the code, this command is executed: `python run.py -s -f sam -n 1` from the terminal.

A: Constructor

1. Initialize initial state: `self.X_0 = initial_state.mu`
2. Add initial node: `self.orig_node_id = self.graph.add_node_pose_2d(self.X_0)`
3. Append initial node to the list of nodes: `self.Nodes.append(self.orig_node_id)`
4. Define information matrix of the initial node $\Lambda = \Sigma^{-1}$: `self.init_inf_matrix = inv(initial_state.Sigma)`
5. Add factor that related to the initial state and initial node: `add_factor_1pose_2d(self.X_0, self.orig_node_id, self.init_inf_matrix)`

```
aleksandr@aleksandr-HP-Pavilion-Gaming-Laptop-15-cx0xxx:~/Perception-in-Robotics-course-T3-2022-S
koltech/ps/PS3/PS3_code$ python run.py -s -f sam -n 1
Status of graph: 1Nodes and 1Factors.
Printing NodePose2d: 0, state =
180
 50
  0
and neighbour factors 1
Printing Factor: 0, obs=
180
 50
  0
 Residuals=
2.00268e-307
 2.4477e-307
2.89272e-307
and Information matrix
1e+12    -0    -0
    0 1e+12    -0
    0     0 1e+12
 Calculated Jacobian =
0 0 0
0 0 0
0 0 0
```

Picture 1. Results on Task 1 A

Conclusion: As the initial observation initial state was provided, no optimization is done. Hence residuals are not initialized. Jacobians are set to zero. State variables are set correctly.

B: Odometry

```
def predict(self, u):
    # Task 1. B: Begin
    # Add target node
    target_node_id = self.graph.add_node_pose_2d(np.zeros(3))
    #print('\n X_est before =', self.graph.get_estimated_state())
    # Get the last estimated state
    self.mu = self.graph.get_estimated_state()[self.orig_node_id]
    # Calculate Jacobian with respect to input signal
    _, V = state_jacobian(self.mu.T[0], u)
    # Calculate information matrix
    W_u = inv(V @ get_motion_noise_covariance(u, self.alphas) @ V.T)
    # Add factor to the graph associated with odometry model
    self.graph.add_factor_2poses_2d_odom(u, self.orig_node_id, target_node_id, W_u)
    #print('\n X_est after =', self.graph.get_estimated_state())
    # Reassign the original node
    self.orig_node_id = target_node_id
    self.Nodes.append(self.orig_node_id)
    # Task 1. B: End
```

Picture 2. Code with comments for task 1 B

Let us check the obtained results:

```
X_est before= [array([[180.],
      [ 50.],
      [  0.]]), array([[0.],
      [0.],
      [0.]])]

X_est after= [array([[180.],
      [ 50.],
      [  0.]]), array([[190.],
      [ 50.],
      [  0.]])]
```

Picture 3. Results on Task 1 B

Conclusion: As we see, after adding odometry factor, estimated state has been changed accordingly the control input.

C: Landmark observation

```python
def update(self, z):
    # Task 1. C: Begin
    self.obsv_id = z[:, 2].T # Vector of observed landmarks
    self.W_z = inv(self.Q)   # Information matrix on observation covariance
    # For each obtained observation
    for i in range(len(self.obsv_id)):
        # If landmark was already observed
        if self.lm_dict.get(int(self.obsv_id[i])):
            # Get node landmark id
            self.node_lm_id = self.lm_dict[self.obsv_id[i]]
            # Add factor to the graph assigned with observation
            self.graph.add_factor_1pose_1landmark_2d(z[i,:2], self.orig_node_id, self.node_lm_id, self.W_z, initializeLandmark=False)

        # Else landmark was not previously observed
        else:
            # Create a new landmark node and connect it with the new state
            self.node_lm_id = self.graph.add_node_landmark_2d(np.zeros(2))
            # Assign new landmark with observations
            self.lm_dict[self.obsv_id[i]] = self.node_lm_id
            # Add factor to the graph assigned with observation
            self.graph.add_factor_1pose_1landmark_2d(z[i,:2], self.orig_node_id, self.node_lm_id, self.W_z, initializeLandmark=True)

def info(self):
    # For each state node
    for i in range(len(self.Nodes)):
        # Print state node id and estimated state of the pose
        print(f'State Node ID:', self.Nodes[i], ', Estimated state: ', self.graph.get_estimated_state()[self.Nodes[i]].T)

    # For each landmark node
    for j in self.lm_dict.values():
        # Print landmark node id and estimated state of the landmark
        print(f'Landmark node ID:', j, ', Estimated landmark state:', self.graph.get_estimated_state()[j].T)
    # Task 1. C: End
```

Picture 4. Code with comments for task 1 C

```
State Node ID: 0 , Estimated state:  [[180.   50.    0.]]
State Node ID: 1 , Estimated state:  [[190.   50.    0.]]
State Node ID: 4 , Estimated state:  [[200.   50.    0.]]
State Node ID: 5 , Estimated state:  [[210.   50.    0.]]
State Node ID: 6 , Estimated state:  [[220.   50.    0.]]
State Node ID: 7 , Estimated state:  [[230.   50.    0.]]
State Node ID: 8 , Estimated state:  [[240.   50.    0.]]
State Node ID: 9 , Estimated state:  [[250.   50.    0.]]
State Node ID: 10 , Estimated state:  [[260.   50.    0.]]
State Node ID: 11 , Estimated state:  [[270.   50.    0.]]
State Node ID: 12 , Estimated state:  [[280.   50.    0.]]
Landmark node ID: 2 , Estimated landmark state: [[474.67790878  40.37893043]]
Landmark node ID: 3 , Estimated landmark state: [[325.34103402   2.57202215]]
Landmark node ID: 13 , Estimated landmark state: [[516.65926915 228.66882348]]
```

Picture 5. Results on Task 1 C

Conclusion: By running 10 steps we have received estimated states of the robot and observed landmarks.

D: Solve

```python
# Task 1. D: Begin
def solve(self):
    self.graph.solve(mrob.GN)

def graph_print(self):
    self.graph.print(True)
# Task 1. D: End
```

Picture 6. Code with comments for task 1 D

```
State Node ID: 0 , Estimated state:  [[1.80000000e+02 5.00000000e+01 6.82660464e-14]]
State Node ID: 1 , Estimated state:  [[1.90000000e+02 5.00000683e+01 1.36532093e-05]]
State Node ID: 4 , Estimated state:  [[ 2.00007613e+02  4.99994647e+01 -1.15997720e-04]]
State Node ID: 5 , Estimated state:  [[ 2.09969902e+02  4.99837696e+01 -2.94338411e-03]]
State Node ID: 6 , Estimated state:  [[ 2.19896735e+02  4.99435986e+01 -5.10963130e-03]]
State Node ID: 7 , Estimated state:  [[ 2.29800312e+02  4.98913910e+01 -5.53598006e-03]]
State Node ID: 8 , Estimated state:  [[ 2.39715416e+02  4.98376835e+01 -5.46670422e-03]]
State Node ID: 9 , Estimated state:  [[ 2.49618293e+02  4.97804510e+01 -6.23069200e-03]]
State Node ID: 10 , Estimated state:  [[ 2.59509292e+02  4.97132847e+01 -7.42128379e-03]]
State Node ID: 11 , Estimated state:  [[ 2.69457734e+02  4.96376109e+01 -7.80648922e-03]]
State Node ID: 12 , Estimated state:  [[ 2.79450270e+02  4.95681704e+01 -6.15531020e-03]]
Landmark node ID: 2 , Estimated landmark state: [[458.19931211 -16.52304938]]
Landmark node ID: 3 , Estimated landmark state: [[319.51100416   0.52149228]]
Landmark node ID: 13 , Estimated landmark state: [[441.59725956 329.18332867]]
```

Picture 7. Results on Task 1 D

Conclusion: By solving the graph at each step we have improved estimations on robot positions and landmark positions.

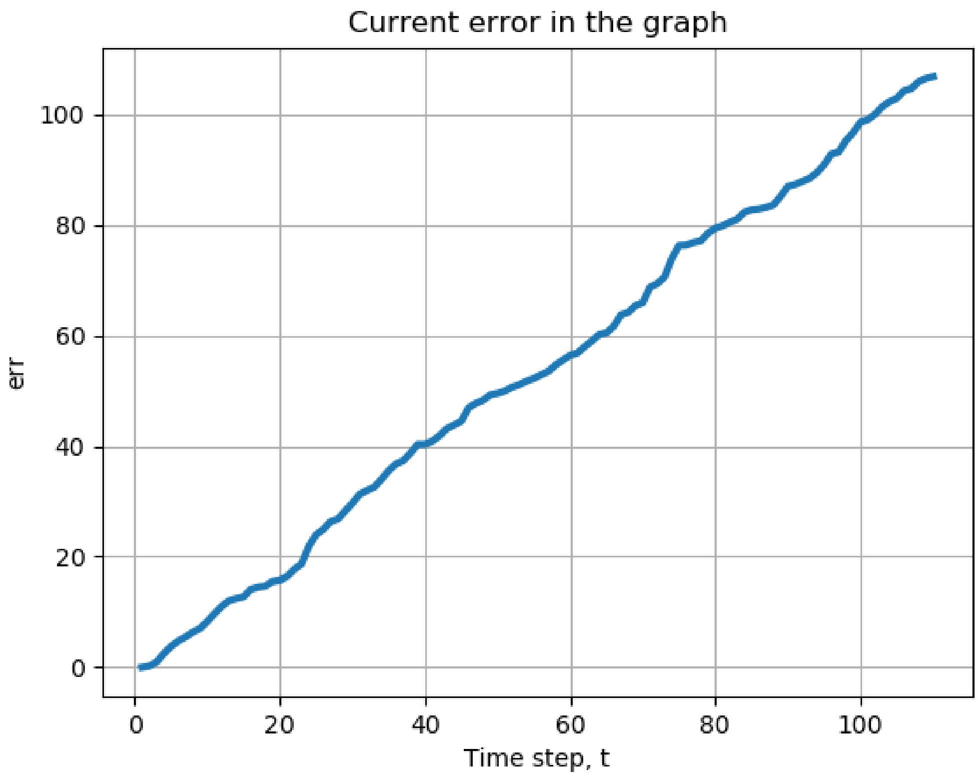
Task 2: SAM evaluation

For the following task, you will be evaluating the data provided in `slam-evaluation-input.npy`.

A: Incremental Solution

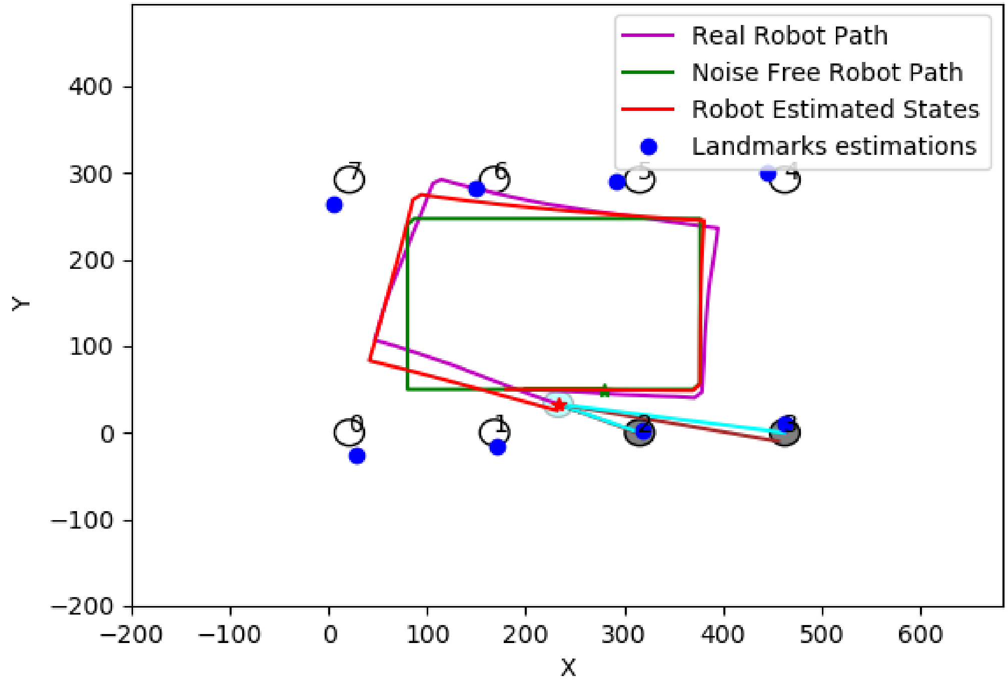
```python
 # Task 2. A: Begin
plt.figure()
plt.plot(np.linspace(1, tp1, tp1), err, linewidth=3)
plt.title('Current error in the graph')
plt.xlabel('Time step, t')
plt.ylabel('err')
plt.grid(True)
 # Task 2. A: End
plt.show(block=True)
```

Picture 8. Code with comments for task 2 A



Picture 9. Results on Task 2 A

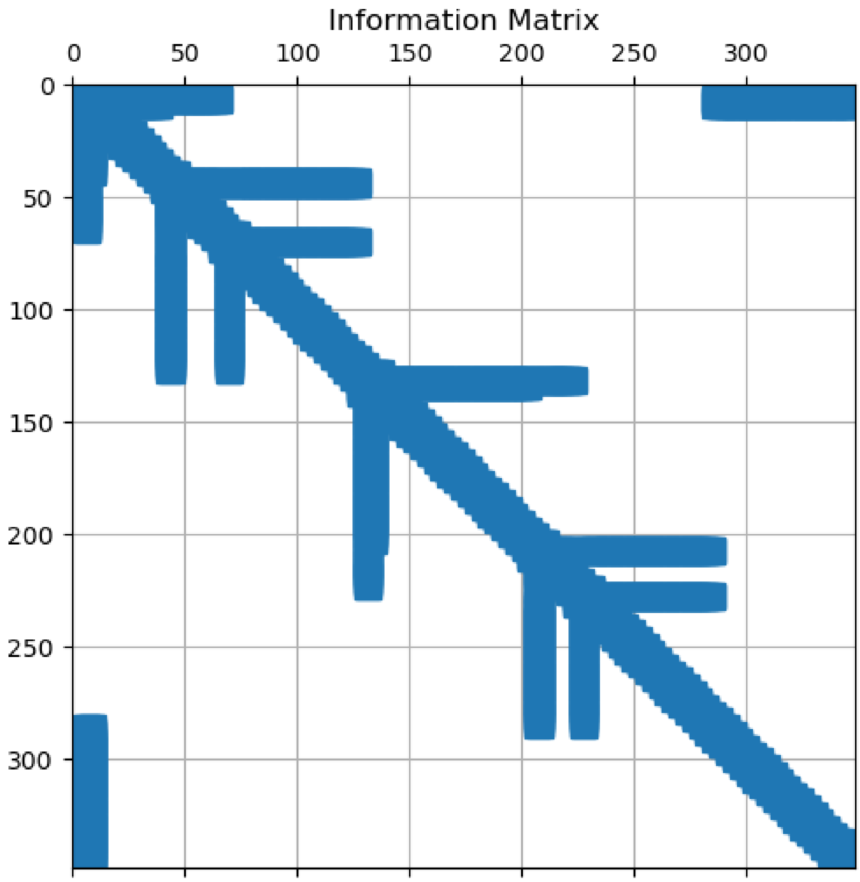
B: Visualization



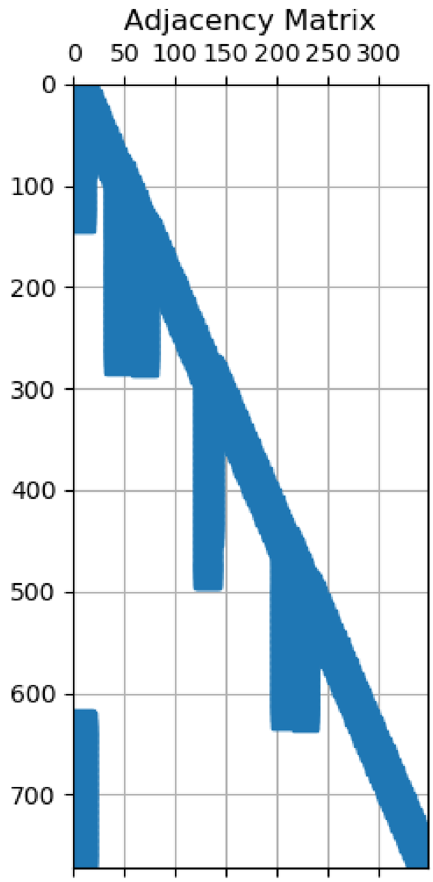
Picture 10. Results on Task 2 B

Video with solution is included in the folder.

C: Adjacency matrix



Picture 11. Results on Task 2 C: Information Matrix



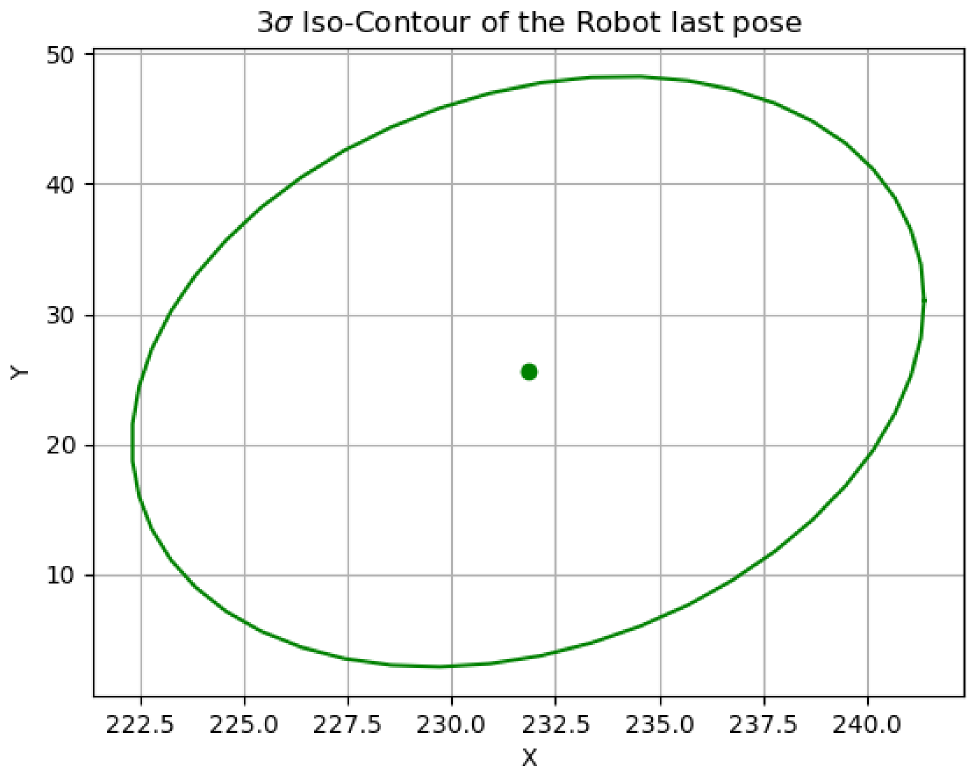
Picture 12. Results on Task 2 C: Adjacency Matrix

Conclusion: Adjacency matrix can be used in order to determine the amount of states (number of columns) and number of factors (amount of rows) and it also affects relations between them.

D: Covariance

```
# Task 2. D: Begin
mean, _ = slam.get_states()
cov = np.linalg.inv(info_mat.todense())[-3:-1, -3:-1]
plt.figure()
plot2dcov(mean[-1, :2], cov, nSigma=3, color='green')
plt.grid()
plt.title(r"3$\sigma$ Iso-Contour of the Robot last pose")
plt.xlabel("X")
plt.ylabel("Y")
# Task 2. D: End
```

Picture 13. Code with comments for task 2 D



Picture 14. Results on Task 2 D

E: Batch solution

```
Graph Solution: GN: i:1, chi2: 134.2525900560378

Graph Solution: GN: i:2, chi2: 107.16060510247696

Graph Solution: GN: i:3, chi2: 106.77378957534165
```

Picture 15. Results on Task 2 E: GN graph solution

```
FGraphSolve::optimize_levenberg_marquardt: iteration 1 lambda = 1e-05, error 755.516, and delta = 623.835
model fidelity = 0.975422 and m_k = 1279.11

FGraphSolve::optimize_levenberg_marquardt: iteration 2 lambda = 2.5e-06, error 131.682, and delta = 24.7425
model fidelity = 0.996971 and m_k = 49.6354

FGraphSolve::optimize_levenberg_marquardt: iteration 3 lambda = 6.25e-07, error 106.939, and delta = 0.166189
model fidelity = 0.995768 and m_k = 0.33379

FGraphSolve::optimize_levenberg_marquardt: iteration 4 lambda = 1.5625e-07, error 106.773, and delta = 0.000255864
```

Picture 14. Results on Task 2 E: LM graph solution

Conclusion: Gauss-Newton graph solution converges to the χ^2 error of 106.773 in 3 iterations, whereas Levenberg-Marquard converges to the same result in 4 iterations. However, GN algorithm requires manual cycle iterations adjusting, whereas LM is adaptive and can determine step to terminate the algorithm.