Problem set 3

Denis Logashov

1 Task 1

A. Constructor. The base class *sam.py* creates a factor graph by using $graph = mrob.FGraph()$ at line 22. In order to correctly initialize the problem, we ask you to add the first node, variable x_0, to the graph.

Now, we need to set an anchor factor to initialize x_0.

Output:

```
#####init#######
[179.99999932408264, 50.00000094688127, -7.177365971167249e-08]
0
Status of graph: 1Nodes and 1Factors.
Printing NodePose2d: 0, state =
         180
          50
-7.17737e-08
and neighbour factors 1
Printing Factor: 0, obs=
         180
          50
-7.17737e-08
 Residuals=
 1.7721e-316
 1.4908e-316
1.98912e-316
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
 Chi2 error = 0 and neighbour Nodes 1
```

Figure 1: Output of the initialization

B. Odometry. Update the function *sam.predict* to add an odometry factor. Also, modify the *run.py* consequently. For this, you need to add a new 2d pose node (see A), without need to be initialized (*np.zeros*(3)).

Output:

Before adding odometry, the state was equal to zeros, as we put *np.zeros*() as input to the *graph.add_node_pose_2d*(*x_0*) (Fig. 2).

```
#####pred#######
[array([[ 1.79999999e+02],
       [ 5.00000009e+01],
       [-7.17736597e-08]]), array([[0.],
       [0.],
       [0.]])]
```

Figure 2: Output before adding odometry factor

After adding odometry, the state will change according to the *u* (Fig. 3).

```
[array([[ 1.79999999e+02],
       [ 5.00000009e+01],
       [-7.17736597e-08]]), array([[ 1.89999999e+02],
       [ 5.00000002e+01],
       [-7.17736597e-08]])]
```

Figure 3: Output after adding odometry factor

C. Landmark observations. Update the function *sam.update* to add a landmark factor. Bear in mind that there are several observations per time step.

Output:

At the Fig. 4 we can see, that 2 landmarks added to the graph (last 2 arrays with x and y coordinates)

```
#####update#######
[array([[ 1.79999999e+02],
       [ 5.00000009e+01],
       [-7.17736597e-08]]), array([[ 1.89999999e+02],
       [ 5.00000002e+01],
       [-7.17736597e-08]]), array([[456.20285449],
       [ -9.13238147]]), array([[321.7413458 ],
       [ -3.41316767]])]
```

Figure 4: Output after adding landmark factors

D. Solve. Modify the function *sam.solve* to include the solving routine *graph.solve*().

Output:

As we can see from the Fig. 5 at the first step solve routine does not affect the graph

```
#####solve#######
[array([[ 1.79999999e+02],
       [ 5.00000009e+01],
       [-7.17736597e-08]]), array([[ 1.89999999e+02],
       [ 5.00000002e+01],
       [-7.17736597e-08]]), array([[456.20285449],
       [ -9.13238147]]), array([[321.7413458 ],
       [ -3.41316767]])]
```

Figure 5: Output after adding landmarks

2 Task 2

A. Incremental Solution. At each time iteration, solve the SAM problem. Plot in a graphic its result w.r.t time.

Output:

At the end ξ^2 achieved 106.76.

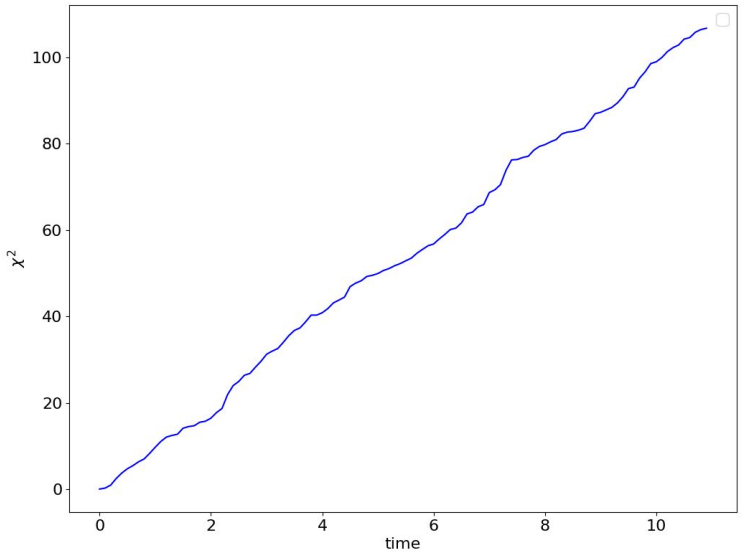

Figure 6: Errors with respect of time

B. Visualization.Plot the current trajectory and landmark estimates in the *run.py* file.

Output:

As we can see on the Fig. 7, we can slightly rotate the predicted positions and landmarks, and they will almost perfectly match the true position of the robot.

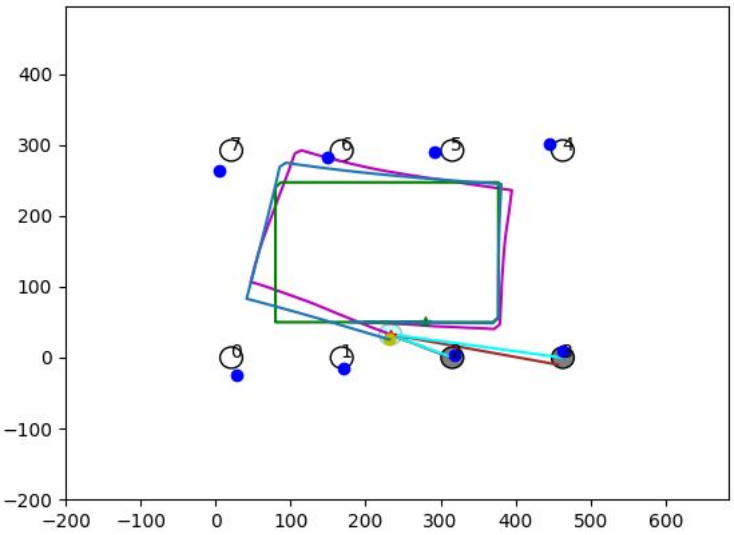

Figure 7: Trajectory and landmark estimates

C. Adjacency matrix. Plot the current adjacency matrix at the last time step. Also print the information matrix.

Output:

The Fig. 8 shows structure of adjacency matrix after last iteration. We can see vertical lines (marked by red lines), which corresponds to the connections between landmark node(column number) and state node (row number). These verical lines are J^{i_k} Jacobians and on the same row but on the diagonal part stored H^{i_k} Jacobian. Each others points corresponds to the connections between states.

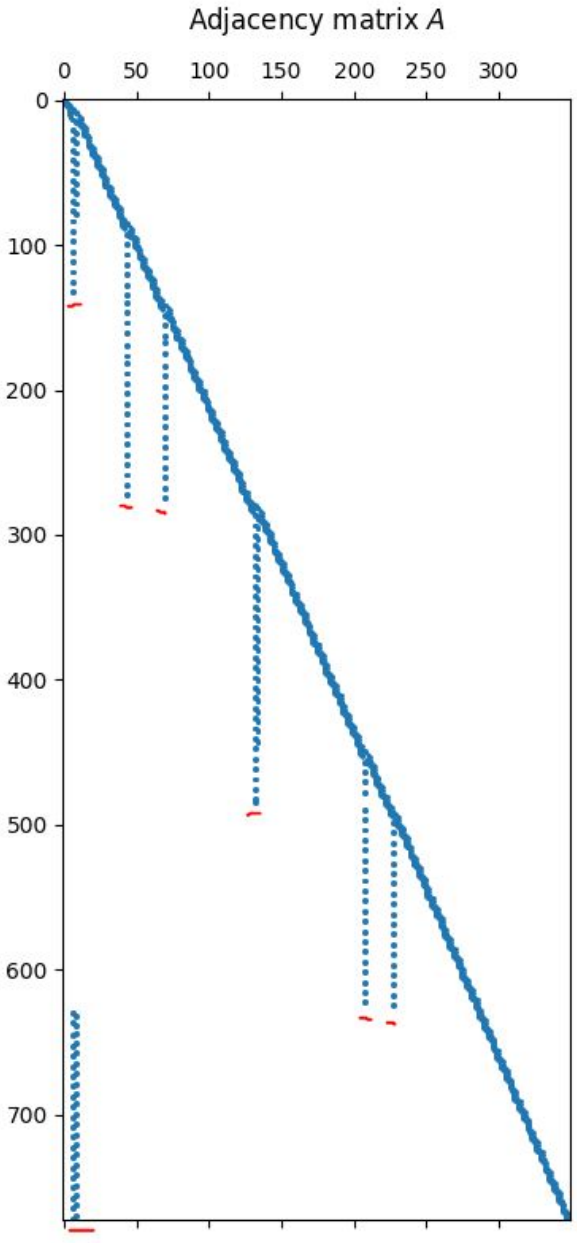

Figure 8: Adjacency matrix

Even with higly sparse information matrix we have dense covariance (inverse of the information (Fig. 9)) matrix(Fig. 10).

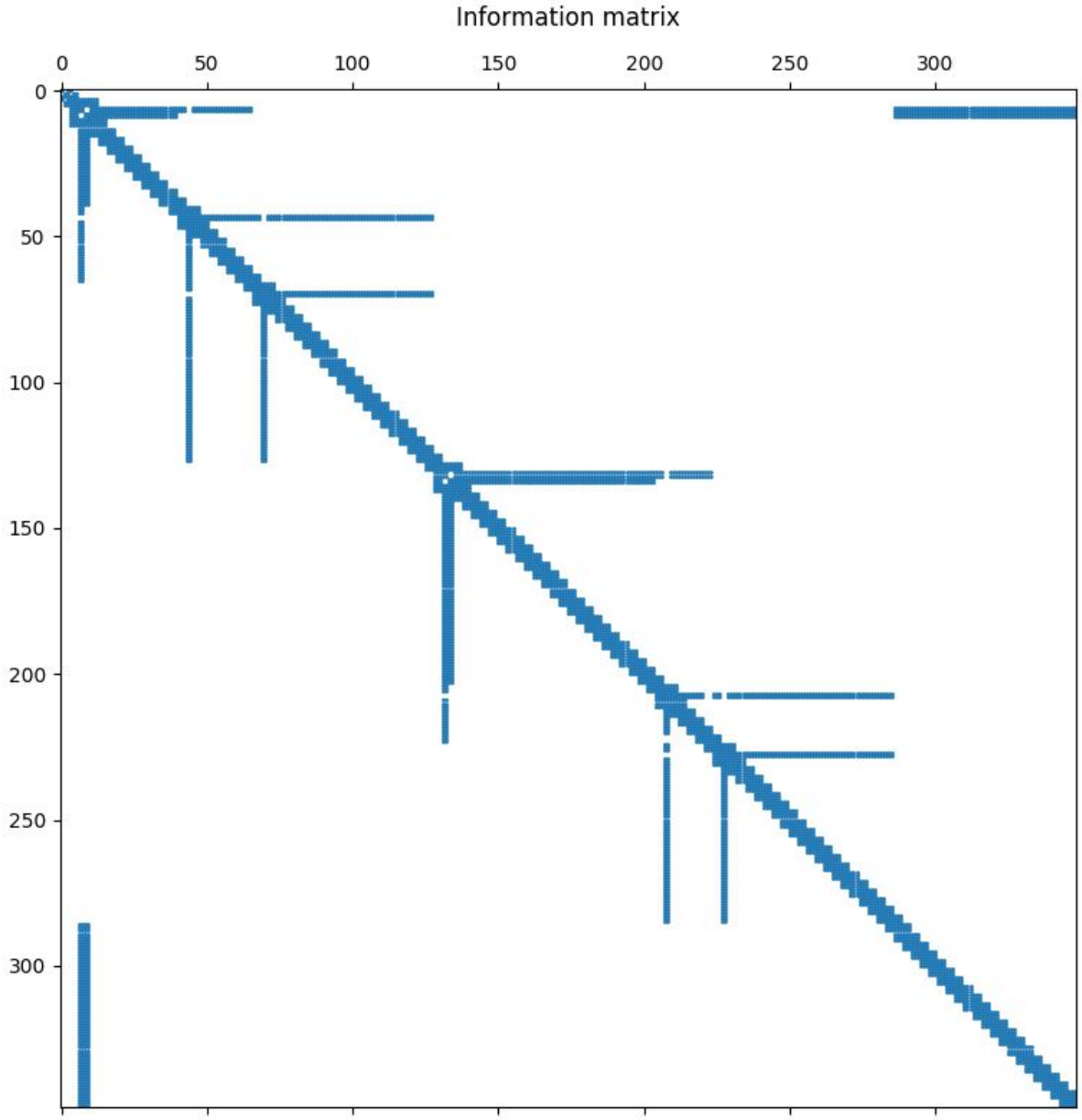

Figure 9: Information matrix

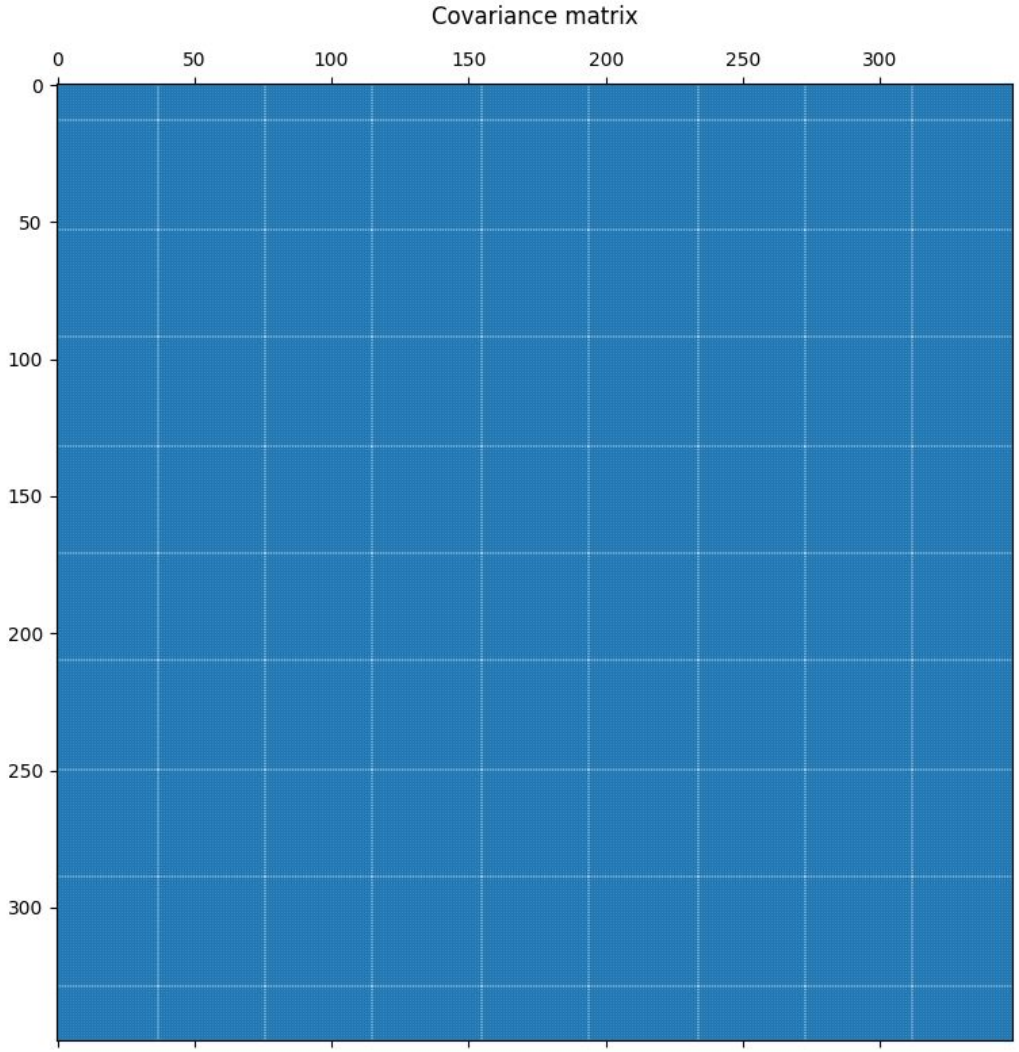

Figure 10: Covariance matrix

D. Covariance. Plot the covariance of the last pose.

Output:

As covariance depends only on the noise in the action state, then covariance of the last pose will be very small.

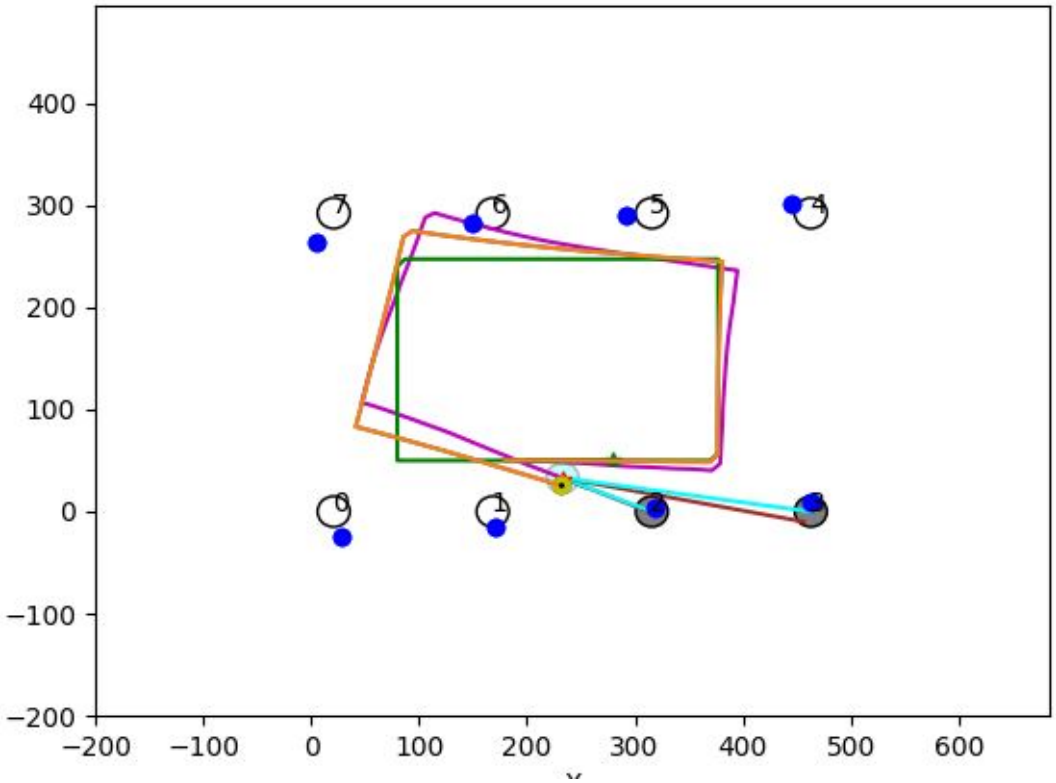

Figure 11: Covariance of the last pose (black circle)

E. Batch solution. Disable solving solution at each iteration and solve only in the last time step.

Output:

Using loop and Gauss-Newton algorithm ξ^2 achieved 106.78 (Fig. 12) for 5 steps. Fig. 13 shows trajectory obtained after solving.

```
number of iterations: 5
achived chi: 106.77262167906261
```

Figure 12: Achived chi for Gauss-Newton

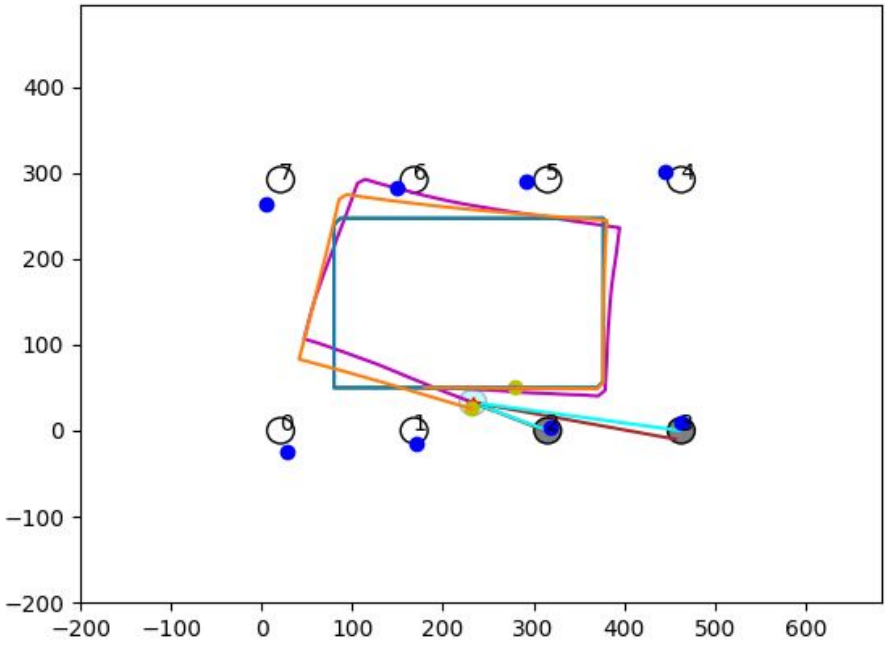

Figure 13: Trajectory after solution

Using Levenberg-Marquard algorithm ξ^2 achieved 106.77 (Fig. 14) for 4 steps.

```
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
```

Figure 14: Achived chi for Levenberg-Marquard