Task 0. Modifying run.py

Before starting the problem set, we have to include calls of $predict$, $update$, and $solve$ methods of SAM class in **run.py** file.

```
Q = np.diag(beta**2)

slam = Sam(
    initial_state=initial_state,
    alphas=alphas,
    slam_type=args.filter_name,
    data_association=args.data_association,
    update_type=args.update_type,
    Q=Q,
)
```

```
# Control at the current step.
u = data.filter.motion_commands[t]
# Observation at the current step.
z = data.filter.observations[t]

slam.predict(u)

slam.update(z)
slam.solve()
```

Also plotting of SLAM solution was implemented.

```python
nodes = np.array(
    [i.flatten() for i in slam.graph.get_estimated_state()], dtype=object
)

obs_states = list(slam.landmark_ids.values())
non_obs_states = [i for i in range(len(nodes)) if i not in obs_states]

nodes_for_plot_obs_states = [nodes[i] for i in obs_states]
nodes_for_plot_non_obs_states = [nodes[i] for i in non_obs_states]

plt.plot(
    [i[0] for i in nodes_for_plot_non_obs_states],
    [i[1] for i in nodes_for_plot_non_obs_states],
    "blue",
)

plt.scatter(
    [i[0] for i in nodes_for_plot_obs_states],
    [i[1] for i in nodes_for_plot_obs_states],
    color="orange",
)

obs_since_last_state = (len(nodes) - 1) - non_obs_states[-1]

prev_node = nodes[slam.prev_node]
inf_matrix = slam.graph.get_information_matrix()
plot2dcov(
    np.array(prev_node[:-1]),
    np.linalg.inv(inf_matrix.toarray())[
        -3 - 2 * obs_since_last_state : -1 - 2 * obs_since_last_state,
        -3 - 2 * obs_since_last_state : -1 - 2 * obs_since_last_state,
    ],
    "b",
    nSigma=3,
)
```

Task 1. Prerequisites to build SAM with known DA

A. Constructor

In $\_\_init\_\_$ method of Sam class we need to add initial state to graph. Here is how it was done.

```
def __init__(
    self,
    initial_state,
    alphas,
    state_dim=3,
    obs_dim=2,
    landmark_dim=2,
    action_dim=3,
    *args,
    **kwargs,
):
    super(Sam, self).__init__(*args, **kwargs)
    self.state_dim = state_dim
    self.landmark_dim = landmark_dim
    self.obs_dim = obs_dim
    self.action_dim = action_dim
    self.alphas = alphas

    self.graph = mrob.FGraph()

    self.landmark_ids = {}
    self.chi2 = []

    self.prev_node = self.graph.add_node_pose_2d(initial_state.mu)
    self.graph.add_factor_1pose_2d(
        initial_state.mu, self.prev_node, np.linalg.inv(initial_state.Sigma)
    )
    self.graph.print(True)
```

And here is the confirmation that it went correct (output after running **python3 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=
 6.94204e-310
 6.94204e-310
-4.96527e-140
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
```

B. Odometry

Next we add odometry factor ($predict$ method).

```python
def predict(self, u):
    print(f"\nEstimated state:\n{self.graph.get_estimated_state()}\n")
    J_u = state_jacobian(
        self.graph.get_estimated_state()[self.prev_node].flatten(), u
    )[-1]
    new_node = self.graph.add_node_pose_2d(np.zeros(3))
    self.graph.add_factor_2poses_2d_odom(
        u,
        self.prev_node,
        new_node,
        np.linalg.inv(J_u @ get_motion_noise_covariance(u, self.alphas) @ J_u.T),
    )
    self.prev_node = new_node
```

After one step we obtain the following state estimation:

```
  0%|
Estimated state:
[array([[180.],
       [ 50.],
       [  0.]])]
```

C. Landmark observations

Next we add landmark factor ($update$ method)

```
def update(self, z):
    for observation in z:
        obs_lm_id = int(observation[-1])
        initializeLandmark = np.logical_not(self.landmark_ids.get(obs_lm_id, 0))
        if initializeLandmark:
            self.landmark_ids[obs_lm_id] = self.graph.add_node_landmark_2d(
                np.zeros(2)
            )
        self.graph.add_factor_1pose_1landmark_2d(
            observation[:-1],
            self.prev_node,
            self.landmark_ids[obs_lm_id],
            np.linalg.inv(self.Q),
            initializeLandmark,
        )
```

To check estimated states in points with no new landmarks, estimations of landmark positions when they appear, and to print information matrix calculated from β parameters, the following lines were inserted in **run.py** file.

```
for idx in non_obs_states:
    print(
        f"Node ID: {idx}\t->\tPosition: {slam.graph.get_estimated_state()[idx].T}"
    )

for idx in obs_states:
    print(
        f"Landmark ID: {idx}\t->\tPosition: {slam.graph.get_estimated_state()[idx].T}"
    )

print(f"Information matrix:\n{slam.Q}")
```

Here is the result after 10 steps.

```
Node ID: 0       ->      Position: [[180.  50.   0.]]
Node ID: 1       ->      Position: [[190.  50.   0.]]
Node ID: 4       ->      Position: [[200.  50.   0.]]
Node ID: 5       ->      Position: [[210.  50.   0.]]
Node ID: 6       ->      Position: [[220.  50.   0.]]
Node ID: 7       ->      Position: [[230.  50.   0.]]
Node ID: 8       ->      Position: [[240.  50.   0.]]
Node ID: 9       ->      Position: [[250.  50.   0.]]
Node ID: 10      ->      Position: [[260.  50.   0.]]
Node ID: 11      ->      Position: [[270.  50.   0.]]
Node ID: 12      ->      Position: [[280.  50.   0.]]
Landmark ID: 2   ->      Position: [[451.86293741 -58.76395757]]
Landmark ID: 3   ->      Position: [[331.2011023   -5.86739347]]
Information matrix:
[[1.0000000e+02 0.0000000e+00]
 [0.0000000e+00 3.0461742e-02]]
```

D. Solve

Now we include graph solving routine into $solve$ method. There is also filling of χ_2 array necessary for task 2A.

```python
def solve(self):
    self.graph.solve()
    self.chi2.append(self.graph.chi2())
```

This is the graph after 10 steps (same as in previous task, but including $solve$)

```
Node ID: 0      ->      Position: [[ 1.80000000e+02  5.00000000e+01 -2.23086276e-15]]
Node ID: 1      ->      Position: [[ 1.90000000e+02  4.99999978e+01 -4.46172555e-07]]
Node ID: 4      ->      Position: [[ 1.99981619e+02  4.99978248e+01 -4.36851677e-04]]
Node ID: 5      ->      Position: [[2.09972868e+02 4.99991514e+01 6.34156440e-04]]
Node ID: 6      ->      Position: [[2.19951888e+02 5.00142872e+01 2.30229492e-03]]
Node ID: 7      ->      Position: [[2.29969196e+02 5.00473799e+01 4.24098104e-03]]
Node ID: 8      ->      Position: [[2.40034984e+02 5.01067271e+01 7.57774534e-03]]
Node ID: 9      ->      Position: [[2.50051543e+02 5.01913309e+01 9.24263398e-03]]
Node ID: 10     ->      Position: [[2.60064062e+02 5.02829022e+01 9.09045826e-03]]
Node ID: 11     ->      Position: [[2.70030812e+02 5.03772135e+01 9.72242154e-03]]
Node ID: 12     ->      Position: [[2.80010103e+02 5.04788435e+01 1.05494537e-02]]
Landmark ID: 2  ->      Position: [[460.81470696 -12.48467408]]
Landmark ID: 3  ->      Position: [[323.85693188  14.3558632 ]]
Information matrix:
[[1.0000000e+02 0.0000000e+00]
 [0.0000000e+00 3.0461742e-02]]
```

$solve$ method adds 'feedback' to our system, so we can see changes of estimated states, which are tiny on nodes, but big on landmarks.

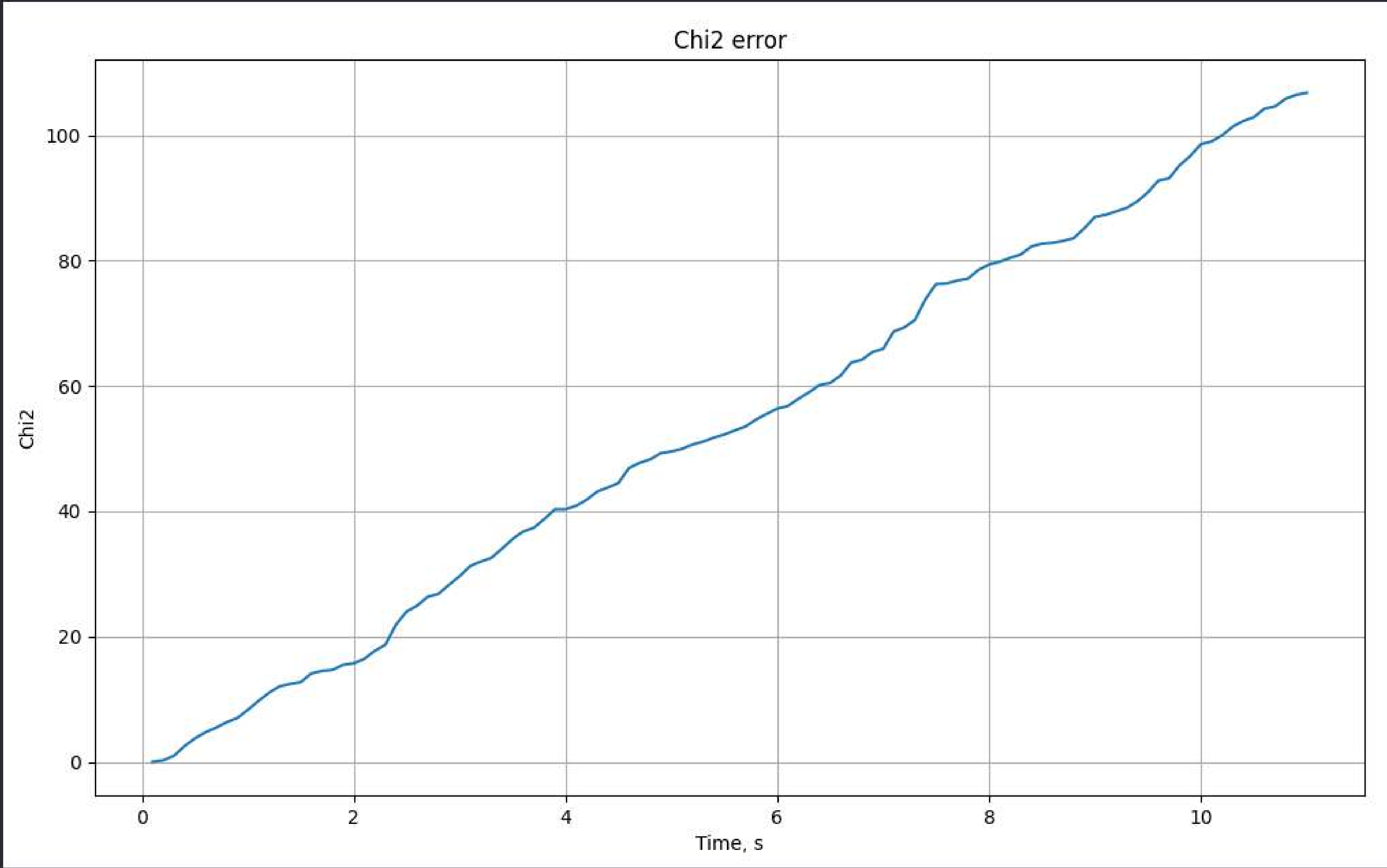
Task 2. SAM evaluation

A. Incremental solution

To plot χ_2 error over time the following function was added to **plot.py** file

```
def plot_chi2(chi2: List[float], dt: float) -> None:
    """
    Plot SAM error over time

    Args:
        chi2 (List[float]): SAM error
        dt (float): Time between measurements
    """
    plt.figure(figsize=(12, 7))
    plt.plot(np.linspace(dt, dt * len(chi2), len(chi2)), chi2)
    plt.title("Chi2 error")
    plt.xlabel("Time, s")
    plt.ylabel("Chi2")
    plt.grid()
    plt.show()
```

Then we call this function in **run.py** and get the following plot on evaluation input dataset.

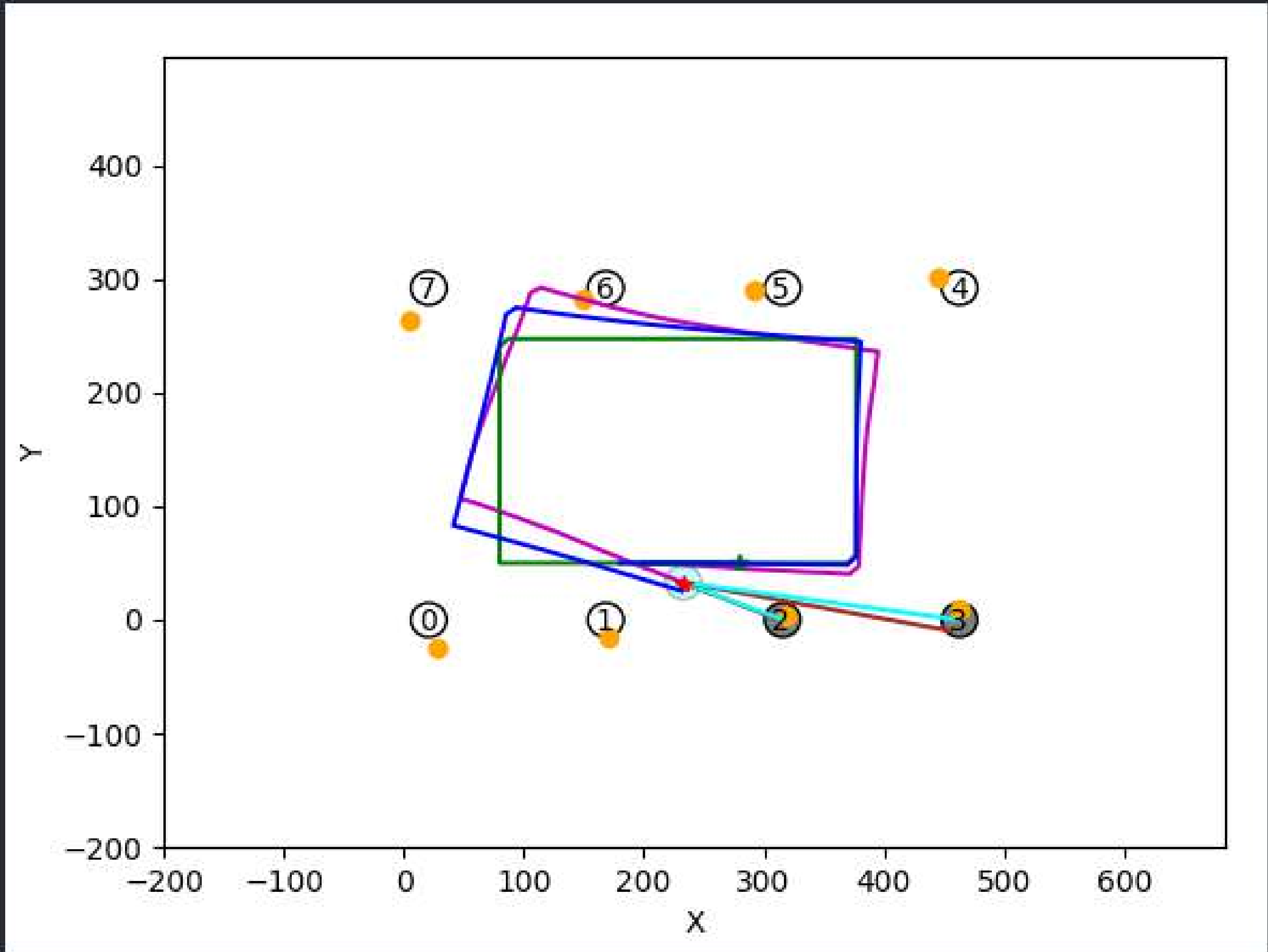


B. Visualization

To plot results of SLAM we should add the following code to **run.py**.

```
nodes = np.array(
    [i.flatten() for i in slam.graph.get_estimated_state()], dtype=object
)

obs_states = list(slam.landmark_ids.values())
non_obs_states = [i for i in range(len(nodes)) if i not in obs_states]

nodes_for_plot_obs_states = [nodes[i] for i in obs_states]
nodes_for_plot_non_obs_states = [nodes[i] for i in non_obs_states]

obs_since_last_state = (len(nodes) - 1) - non_obs_states[-1]

prev_node = nodes[slam.prev_node]
inf_matrix = slam.graph.get_information_matrix()
plot2dcov(
    np.array(prev_node[:-1]),
    np.linalg.inv(inf_matrix.toarray())[
        -3 - 2 * obs_since_last_state : -1 - 2 * obs_since_last_state,
        -3 - 2 * obs_since_last_state : -1 - 2 * obs_since_last_state,
    ],
    "b",
    nSigma=3,
)

if should_show_plots:
    plt.plot(
        [i[0] for i in nodes_for_plot_non_obs_states],
        [i[1] for i in nodes_for_plot_non_obs_states],
        "blue",
    )

    plt.scatter(
        [i[0] for i in nodes_for_plot_obs_states],
        [i[1] for i in nodes_for_plot_obs_states],
        color="orange",
    )
    plt.draw()
    plt.pause(args.plot_pause_len)
```

The video is in attached **sam.mp4** file. Below there is an figure of trajectory and landmarks evaluated positions after last step.

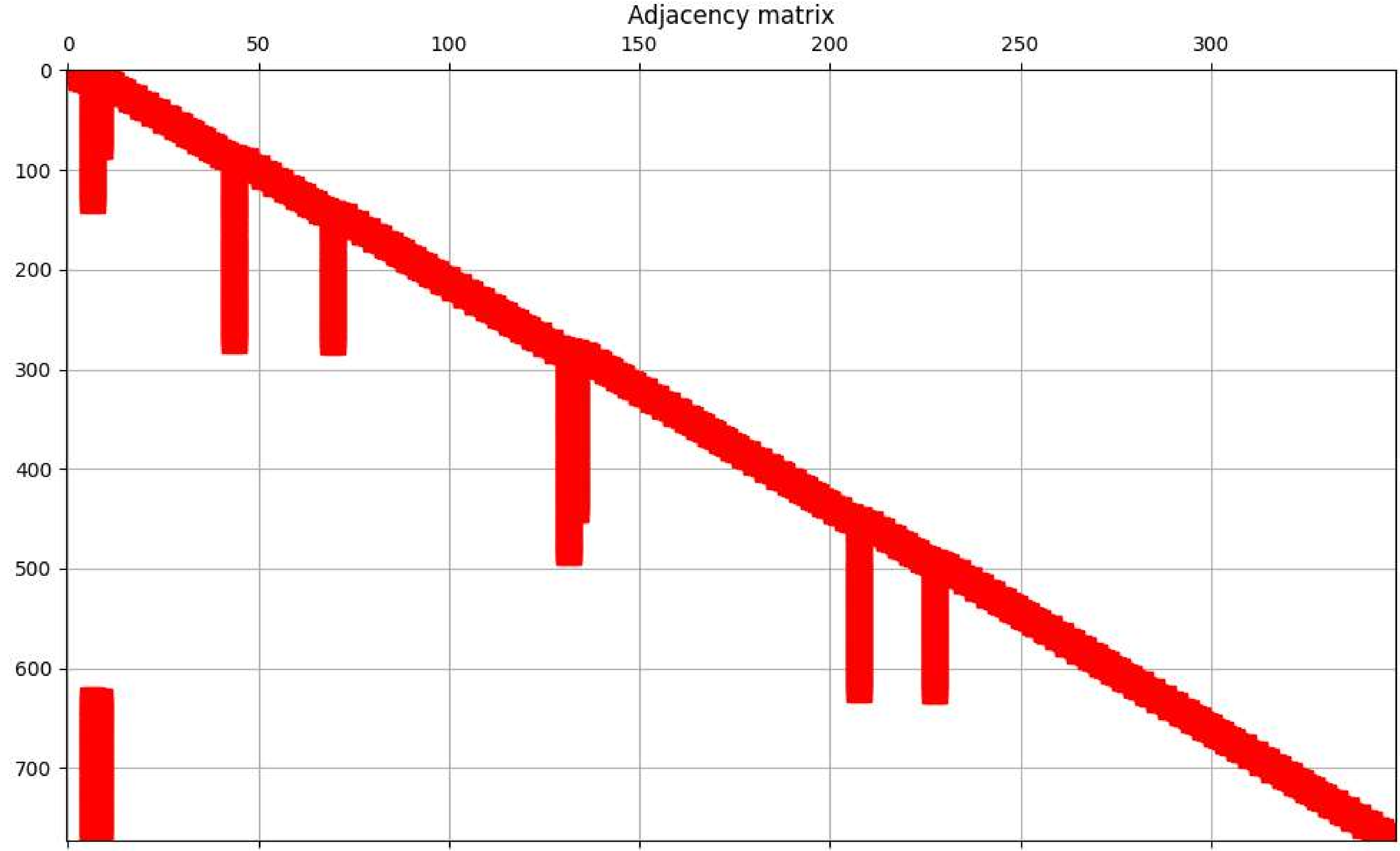


C. Adjacency matrix

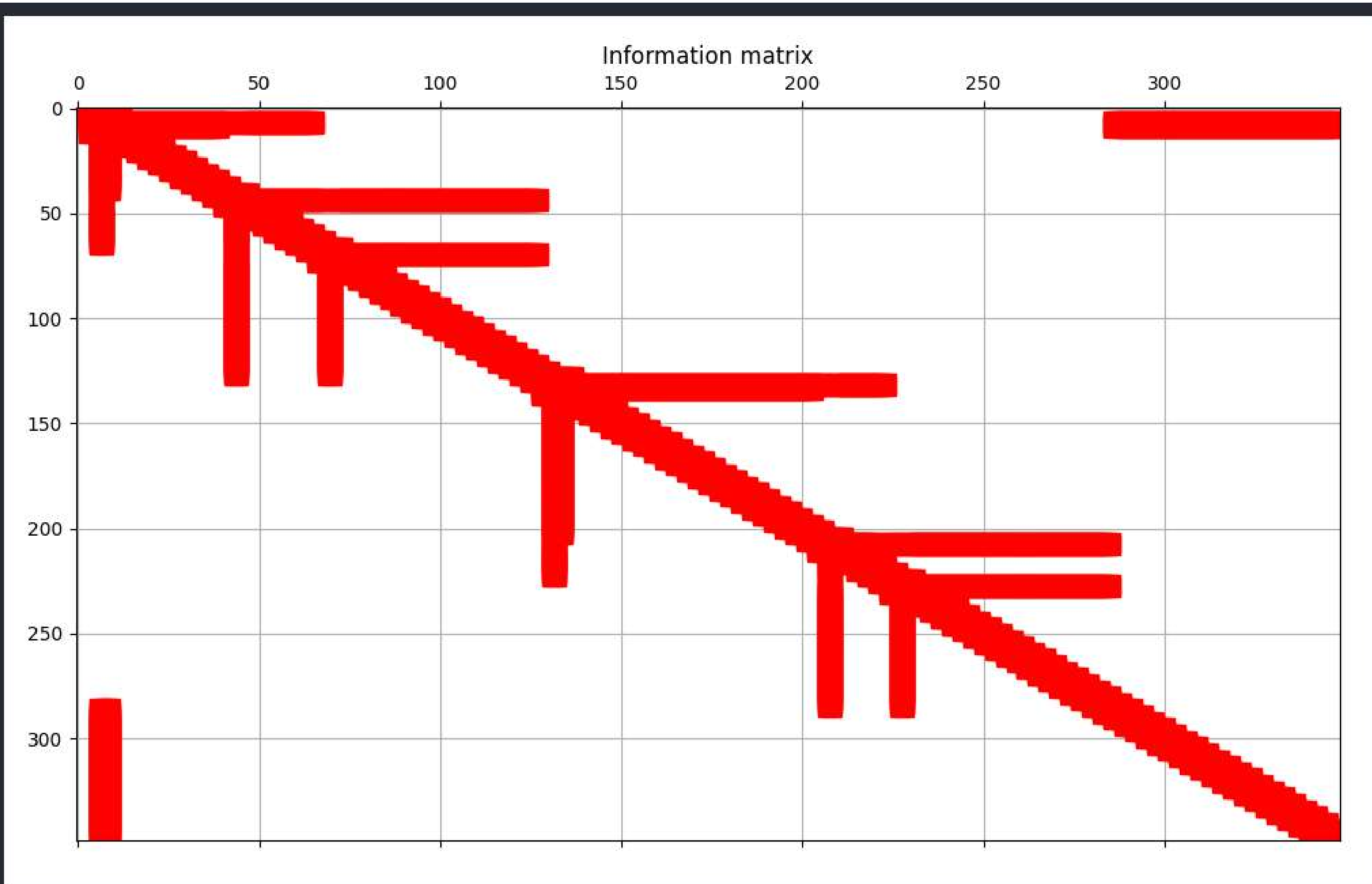
To plot adjacency and information matrices the following function was added to **plot.py** file

```python
def plot_matrix(matrix: spmatrix, name: str) -> None:
    """
    Plot sparse matrix

    Args:
        matrix (spmatrix): Sparse matrix to plot
        name (str): Name of figure that is displayed as a title
    """
    plt.figure(figsize=(12, 7))
    plt.title(name)
    plt.spy(matrix, aspect="auto", color="red")
    plt.grid()
    plt.show()
```

Below there are plots of both matrices on the last step.



Adjacency matrix
0
50
100
150
200
250
300
0
100
200
300
400
500
600
700

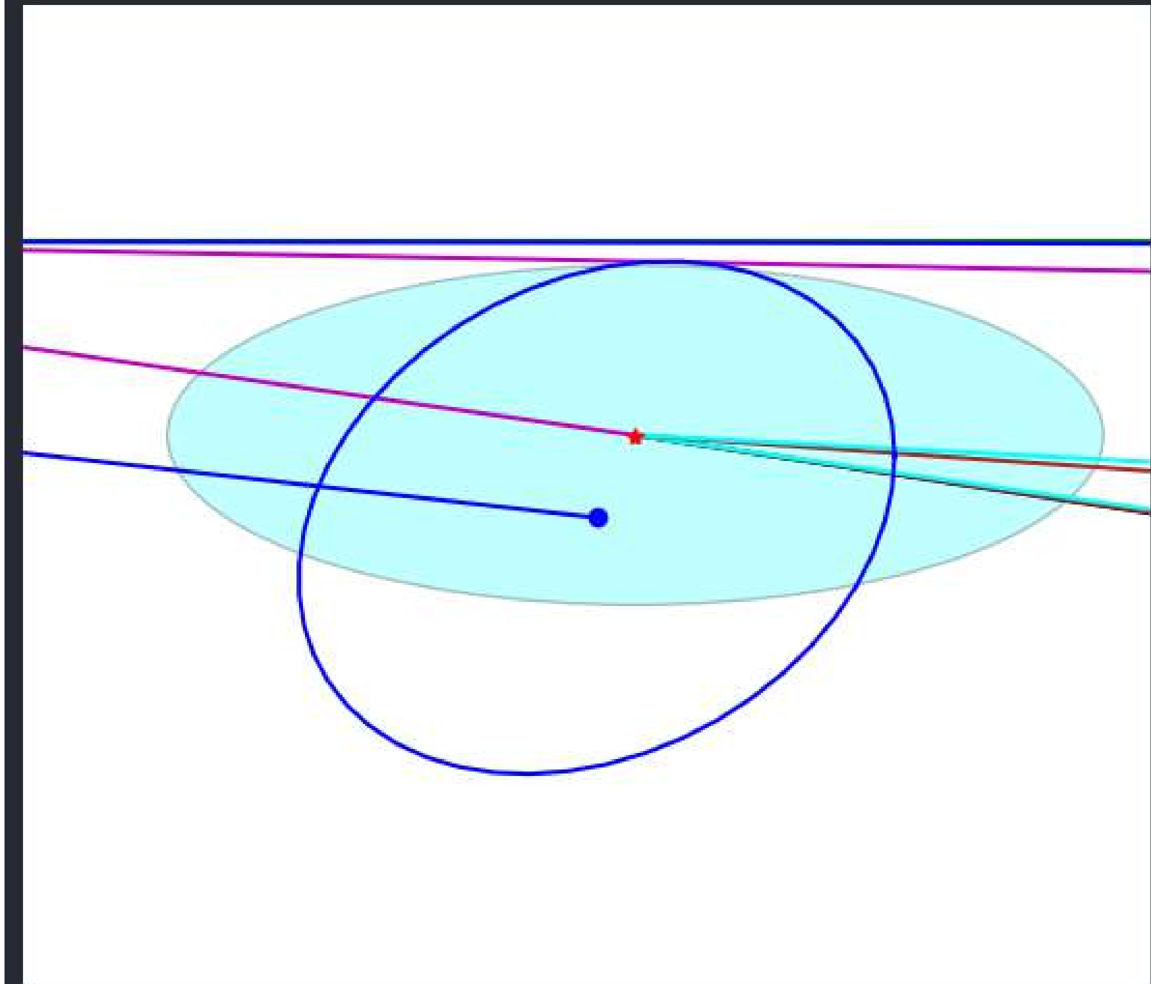


Adjacency matrix is a symmetrical matrix that shows connections between nodes in graph. Informaiton matrix is lower triangular, therefore all of its values are located below the main diagonal.

To plot 3σ iso-contour of the covariance we should add the following lines to **run.py**.

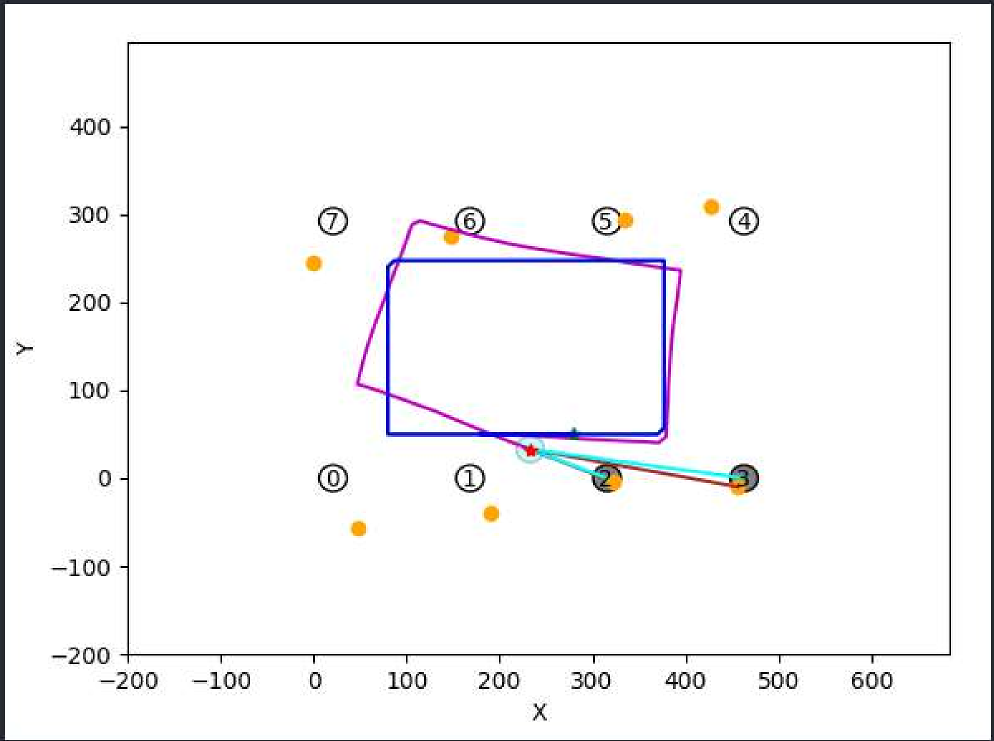
```
plot2dcov(
    np.array(prev_node[:-1]),
    np.linalg.inv(inf_matrix.toarray())[
        -3 - 2 * obs_since_last_state : -1 - 2 * obs_since_last_state,
        -3 - 2 * obs_since_last_state : -1 - 2 * obs_since_last_state,
    ],
    "b",
    nSigma=3,
)
```

Below there is a result.



E. Batch solution

For batch solution we'll use Levenberg-Marquard algorithm for optimization on last step only. Below there is a figure of trajectory and landmark estimated positions in this case.



LM algorithm required 4 iterations to reach convergence in error value of 106.773.

```
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
```

GN algorithm required 3 iterations for the same goal.

```
Iteration 1 -> error: 134.25259011494413
Iteration 2 -> error: 107.16060510852145
Iteration 3 -> error: 106.77378957537209
Iteration 4 -> error: 106.77262175342977
Iteration 5 -> error: 106.77262167906254
Iteration 6 -> error: 106.77262167674498
Iteration 7 -> error: 106.772621677101
Iteration 8 -> error: 106.77262167713049
Iteration 9 -> error: 106.77262167713104
Iteration 10 -> error: 106.77262167713126
```