Task 1: Prerequisites to build SAM with known DA (40 points)

In this task, you will implement the prerequisites for landmark-SAM. Your robot is driving around an environment obtaining observations to a number of landmarks. The position of these landmarks is not initially known, nor is the number of landmarks. For now, we'll simplify the problem: when the robot observes a landmark, you know which landmark it has observed (i.e., you have perfect data association).

For this problem, your landmark observations are $[range, bearing, markerId]$ tuples. Your robot state is $[x, y, \theta]$ (cm, cm, radians) and the motion control is $[\delta_{\text{rot1}}, \delta_{\text{trans}}, \delta_{\text{rot2}}]$ (radians, cm, radians).

Your task is to use the `mrob` library to handle the graphSLAM problem and provide a solution.

A. (10 pts) **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. For this, you will need to use the method `graph.add_node_pose_2d(x_0)`. Take a look at help for input required dimensions. This function, and all functions creating node variables, return their corresponding node id key, and requires as input an initial guess of the value, in the form of a np.array of 3 elements.

Now, we need to set an anchor factor to initialize x_0. For that, use `graph.add_factor_1pose_2d(x0, nodeId, W)`. The inputs are: observation, node id, returned by the previous function `add_node_pose_2d`, and information matrix of the observation. All these values are given in the initial conditions.

Check that you have actually added the node by using the command `graph.print(True)`, it will print all the information in the graph. If set to false, it will only plot the number of nodes and factors. Include in your report the output from print. Since no optimization has been done, you will observe that the residuals are not initialized (garbage values) and Jacobians are set to zero. State variables should be correct.

Hint: For these initial tasks, you may want to run a single iteration of the optimizer `python3 run.py -s -f sam -n 1`.

```
    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.nodes = []
        self.x_0 = initial_state.mu
        self.node_id = self.graph.add_node_pose_2d(self.x_0)
        self.nodes.append(self.node_id)
        self.information_matrix = inv(initial_state.Sigma)
        self.graph.add_factor_1pose_2d(self.x_0, self.node_id, self.information_matrix)
        self.graph.print(True)
```

```
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
 Chi2 error = 0 and neighbour Nodes 1
```

Initial state was provided. In this case residuals are not initialized, Jacobians are set to zero. State variables are set correctly.

B. (10 pts) **Odometry.** Update the function `sam.predict` (in the file `slam/sam.py`) 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)). Later, we will add the new factor corresponding to the odometry observation. We will use the function `graph.add_factor_2pose_2d_odom(u, nodeOriginId, nodeTargetId, W_u)`. Take a look at help. There is an interesting feature about odometry factor, it assumes that the node needs to be initialized given the last node state and the action u, so it does all required calculations without further intervention.

You need to calculate the covariance in state space and invert to obtain the infomration matrix `W_u`.

To check this, use the function `graph.get_estimated_state()`. It returns the list of all nodes estimated. Check out the values before and after adding the odometry factor. Include the information in your report.

```
def predict(self, u):
    next_node_id = self.graph.add_node_pose_2d(np.zeros(3))
    print('Before:\n', self.graph.get_estimated_state())
    self.mu = self.graph.get_estimated_state()[self.node_id]
    res = state_jacobian(self.mu.T[0], u)
    V = res[1]
    W_u = inv(V @ get_motion_noise_covariance(u, self.alphas) @ V.T)
    self.graph.add_factor_2poses_2d_odom(u, self.node_id, next_node_id, W_u)
    print('After:\n', self.graph.get_estimated_state())
    self.node_id = next_node_id
    self.nodes.append(self.node_id)
```

```
Before:
 [array([[180.],
        [ 50.],
        [  0.]]), array([[0.],
        [0.],
        [0.]])]
After:
 [array([[180.],
        [ 50.],
        [  0.]]), array([[190.],
        [ 50.],
        [  0.]])]
```

After adding odometers the state of the system changed due to input signal after one step.

C. (10 pts) **Landmark observations.** Update the function `sam.update` to add a landmark factor. Bear in mind that there are several observations per time step. In case that the landmark has not been previously observed, you should add a new landmark node to the graph: `graph.add_node_landmark_2d(np.zeros(2))`. Initialization is again automatic if indicated so in the factor. You may add the factor corresponding to the landmark observation: `graph.add_factor_1pose_1landmark_2d(z, nodeOriginId, nodeLandmakrId, W_z, initializeLandmark=True)`. Take a look at help. If `initializeLandmark = True`, then automatically the value of the landmark node is initialized according to the inverse of the observation function, as explained in L08 SLAM. If false (by default), it simply adds the observation without modifying the value of the landmark node (necessary for all other observation except a new landmark.

You need to calculate the information matrix from the beta parameters.

Print all nodes in the graph with poses and landmark.

```
def info(self):
    for node in self.nodes:
        print("Node ID: %i, Position estimation: %s" % (node, self.graph.get_estimated_state()[node].T))

    for landmark in self.landmarks.values():
        print('Landmark ID: %i, Landmark estimation: %s' % (landmark, self.graph.get_estimated_state()[landmark].T))


    print('Information matrix:\n', self.information_matrix)
```

```
Node ID: 0, Position estimation: [[180.  50.   0.]]
Node ID: 1, Position estimation: [[190.  50.   0.]]
Node ID: 4, Position estimation: [[200.  50.   0.]]
Node ID: 5, Position estimation: [[210.  50.   0.]]
Node ID: 6, Position estimation: [[220.  50.   0.]]
Node ID: 7, Position estimation: [[230.  50.   0.]]
Node ID: 8, Position estimation: [[240.  50.   0.]]
Node ID: 9, Position estimation: [[250.  50.   0.]]
Node ID: 10, Position estimation: [[260.  50.   0.]]
Node ID: 11, Position estimation: [[270.  50.   0.]]
Node ID: 12, Position estimation: [[280.  50.   0.]]
Landmark ID: 2, Landmark estimation: [[483.81277549  36.88220312]]
Landmark ID: 3, Landmark estimation: [[317.15117937  37.06121834]]
Landmark ID: 13, Landmark estimation: [[433.50811414 299.75953859]]
Information matrix:
 [[ 1.00000000e-02 -0.00000000e+00]
 [ 0.00000000e+00  3.28280635e+01]]
```

After 10 steps we have received estimated states of the robot position and landmarks observations.

D. (10) **Solve.** Modify the function `sam.solve` to include the solving routine `graph.solve()`. This function corresponds to a single iteration of the Gauss-Newton optimization.

Print the full graph after optimization. Comment on the results.

```
def solve(self, method):
    self.graph.solve(method)
```

`slam.solve(mrob.GN)`

```
Node ID: 0, Position estimation: [[1.80000000e+02 5.00000000e+01 8.21214912e-15]]
Node ID: 1, Position estimation: [[1.90000000e+02 5.00000082e+01 1.64242983e-06]]
Node ID: 4, Position estimation: [[1.99995685e+02 5.00032196e+01 6.39777941e-04]]
Node ID: 5, Position estimation: [[2.09965734e+02 5.00200288e+01 2.65617632e-03]]
Node ID: 6, Position estimation: [[2.19958266e+02 5.00602988e+01 5.35334246e-03]]
Node ID: 7, Position estimation: [[2.30002323e+02 5.01272901e+01 8.07510170e-03]]
Node ID: 8, Position estimation: [[2.40020497e+02 5.02165376e+01 9.75812142e-03]]
Node ID: 9, Position estimation: [[2.50030006e+02 5.03147706e+01 9.92584596e-03]]
Node ID: 10, Position estimation: [[2.60022534e+02 5.04080683e+01 8.77050981e-03]]
Node ID: 12, Position estimation: [[2.69995757e+02 5.04865565e+01 6.88476640e-03]]
Node ID: 13, Position estimation: [[2.79976656e+02 5.05472179e+01 5.35175858e-03]]
Landmark ID: 2, Landmark estimation: [[453.78781233 -25.42169477]]
Landmark ID: 3, Landmark estimation: [[318.28579162   0.81172561]]
Landmark ID: 11, Landmark estimation: [[469.56847945 297.92368253]]
```

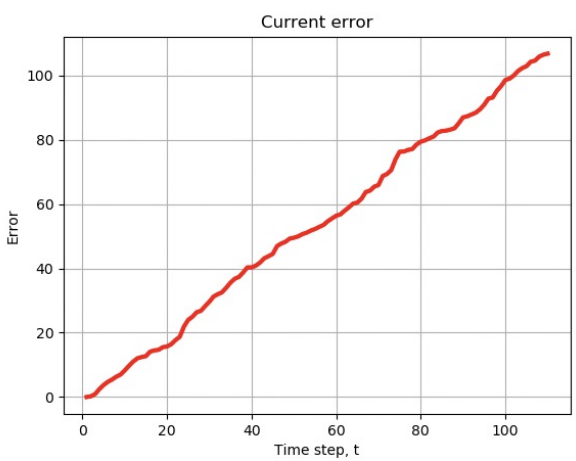
After including slam solve algorithm we received real robot and landmarks position estimations.

Task 2: SAM evaluation (40 points)

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

A. (5 pts) **Incremental Solution.** At each time iteration, solve the SAM problem. Monitor the current error in the graph at each iteration by using the function `graph.chi2()`. This function re-evaluates all residuals and calculates the current error. Plot in a graphic its result w.r.t time.

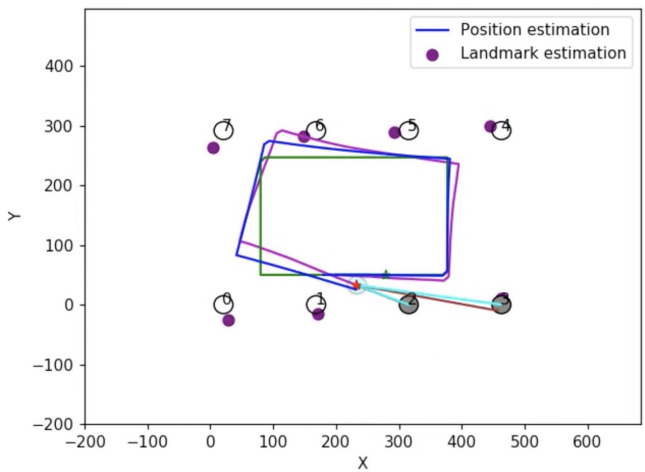
```
def plot_chi2(tp1, err):
    plt.figure()
    plt.plot(np.linspace(1, tp1, tp1), err, lw = 3, c = 'red')
    plt.title('Current error')
    plt.xlabel('Time step, t')
    plt.ylabel('Error')
    plt.grid(True)
    plt.show(block = True)
```



B. (10 pts) **Visualization.** Plot the current trajectory and landmark estimates in the `run.py` file.

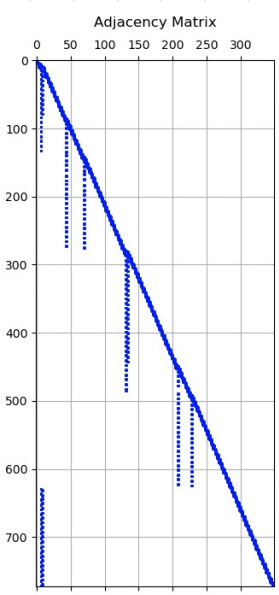
Note: you may need to create a data structure to keep track of the landmarks id's to plot them separately and all the state variables corresponding to poses.

```
if should_show_plots:
    plt.plot(position_states[:,0], position_states[:,1], 'blue', label='Position estimation')
    plt.scatter(landmarks_states[:,0], landmarks_states[:,1], c = 'purple', s = 50, label='Landmark estimation')
    plt.legend(loc='upper right')
    plt.draw()
    plt.pause(args.plot_pause_len)
```

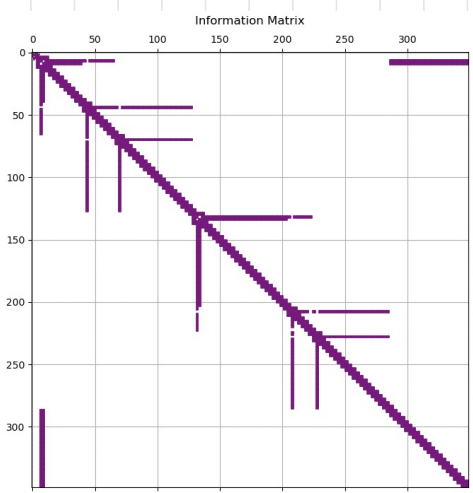


C. (5 pts) **Adjacency matrix.** Plot the current adjacency matrix at the last time step. For this, use the function `graph.get_adjacency_matrix()`, returning a sparse matrix. Comment on its structure. Also print the information matrix using the function `graph.get_information_matrix()`.

Note: use matplotlib `spi` function



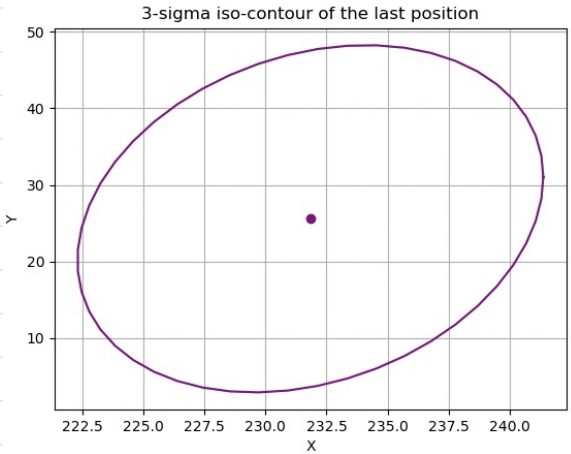

```
def plot_matrix(matrix, title, color):
    plt.figure(figsize=(10, 8))
    plt.spy(matrix, markersize = 1, color = color)
    plt.title(title)
    plt.grid(True)
    plt.show(block = True)
```



Adjacency matrix is a squared matrix, where are represented amount of states (number of columns), number of factors (amount of rows) and also relations between them.

D. (10 pts) **Covariance.** Plot the 3-sigma iso-contour of the covariance of the last pose.

Note: you may use the function `plot2cov` in tools.




```python
def plot_iso(slam, information_matrix):
    mean, _ = slam.get_states()
    cov = np.linalg.inv(information_matrix.todense())[-3:-1, -3:-1]
    plt.figure()
    plot2dcov(mean[-1, :2], cov, nSigma=3, color='purple')
    plt.grid()
    plt.title("3-sigma iso-contour of the last position")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show(block = True)
```

E. (10 pts) **Batch solution.** Disable solving solution at each iteration and solve only in the last time step. In this case, you would need to call multiple times `graph.solve()`, checking for convergence with the chi2 function. Fortunately, the Levenberg-Marquard algorithm `graph.solve(mrob.LM)` optimizes until convergence and adapts the update value of state variables as the iterations go on. Report on the number of iterations required (printed in console) and the final chi2 error achieved.

`slam.solve(method=mrob.LM)`

```
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
```

```python
for i in range(5):
  slam.solve(mrob.GN)
  print('\n')
  print('GNGraphSolve: i:', i+1, 'error: ', slam.graph.chi2())
```

```
GNGraphSolve: i: 1 error:  134.25259010738165

GNGraphSolve: i: 2 error:  107.16060510525982

GNGraphSolve: i: 3 error:  106.77378957525534

GNGraphSolve: i: 4 error:  106.77262175342969

GNGraphSolve: i: 5 error:  106.77262167906254
```

Gauss-Newton graph solution reaches the error of 106.773 in 3 iterations, while using Levenberg-Marquard method to the same result is obtained in 4 iterations. But GN algorithm needs manual cycle iterations, while LM is adopted to automatic calculations.