PS3-SLAM

PERCEPTION IN ROBOTICS – TERM 3, 2021

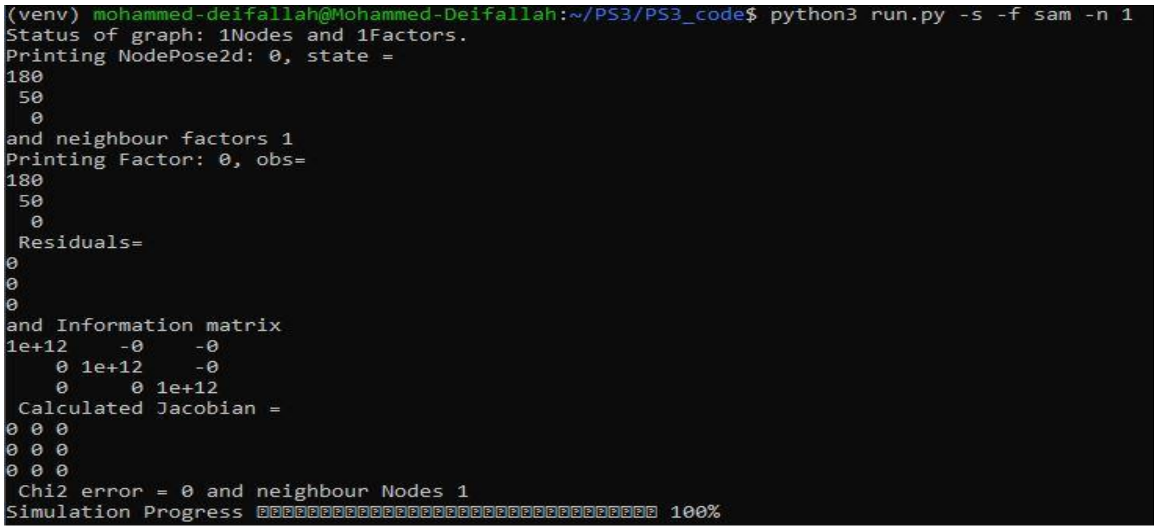
Mohammed Deifallah | 13/03/2021

Task 1: Prerequisites to build SAM with known DA

For this part, the command "python3 run.py -s -f sam -n 1"

TASK 1.A

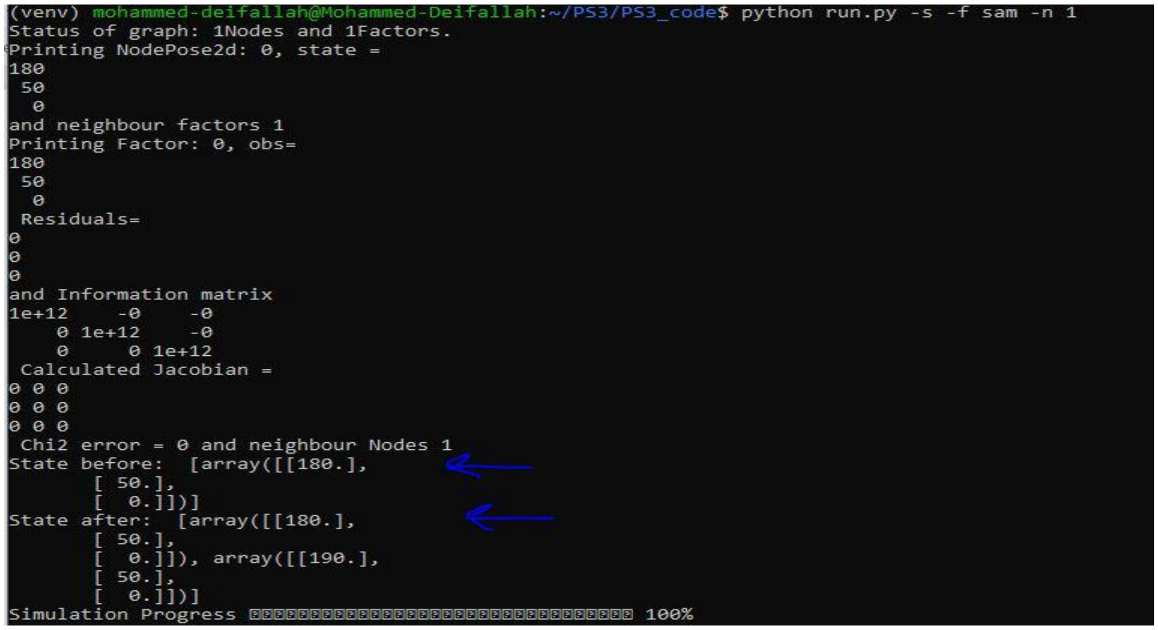
As shown below, the graph has only the initial node, while the Jacobian has garbage values (all zeros).



```
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$ 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=
0
0
0
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
Simulation Progress ████████████████████████████████ 100%
```

TASK 1.B

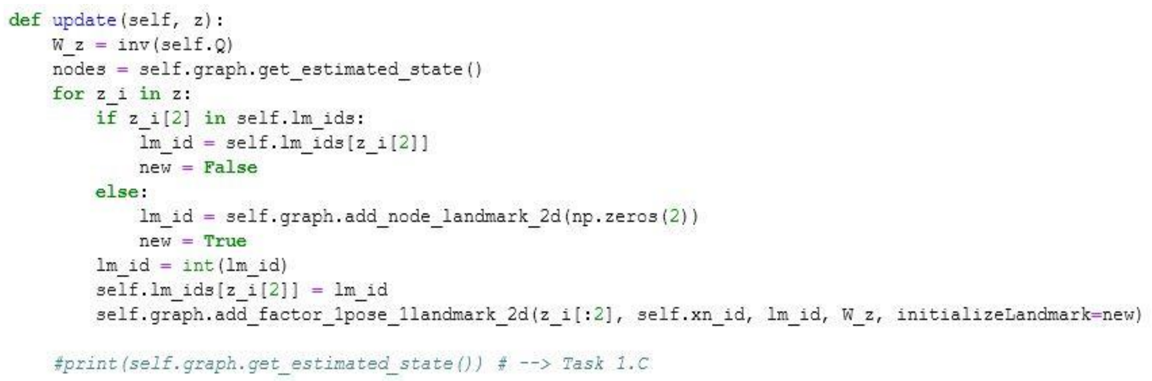
In order to show the effect after the change in prediction function, the following screenshot shows the state before and after adding a new pose node and odometry factor as the two blue arrows shows:



```
(venv) mohammed-deifallah@Mohammed-Deifallah:~/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=
0
0
0
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
State before:  [array([[180.],
       [ 50.],
       [  0.]])]
State after:  [array([[180.],
       [ 50.],
       [  0.]]), array([[190.],
       [ 50.],
       [  0.]])]
Simulation Progress ████████████████████████████████ 100%
```

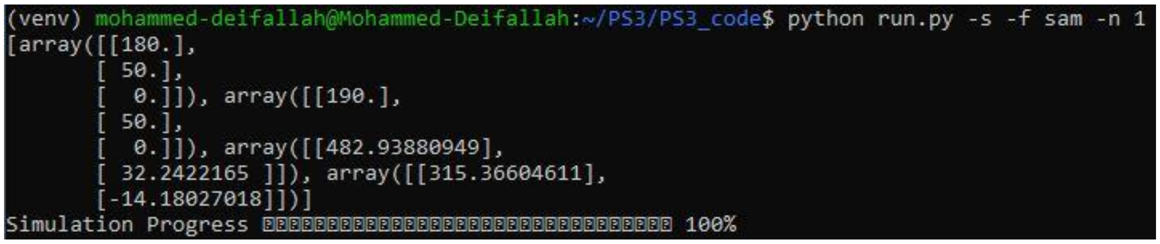
TASK 1.C

Here’s how the matrix Q is calculated at the beginning of the program from the values of betas: `Q = np.diag([*(beta**2)])`, and a new data structure python dictionary) is added to SAM class, so that it keeps track of existing landmarks and adding new ones, where each landmark ID is mapped to its node ID in the Factor Graph. The following code snippet shows how the update function is designed for this subtask:



```
def update(self, z):
    W_z = inv(self.Q)
    nodes = self.graph.get_estimated_state()
    for z_i in z:
        if z_i[2] in self.lm_ids:
            lm_id = self.lm_ids[z_i[2]]
            new = False
        else:
            lm_id = self.graph.add_node_landmark_2d(np.zeros(2))
            new = True
        lm_id = int(lm_id)
        self.lm_ids[z_i[2]] = lm_id
        self.graph.add_factor_1pose_1landmark_2d(z_i[:2], self.xn_id, lm_id, W_z, initializeLandmark=new)

    #print(self.graph.get_estimated_state()) # --> Task 1.C
```

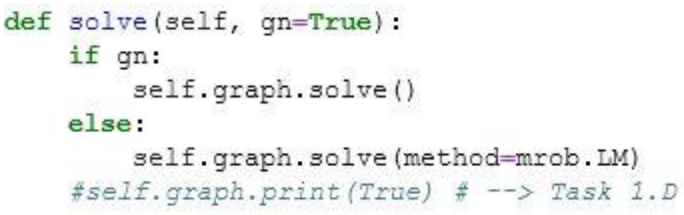
Uncommenting the last line prints the following output in the console:



```
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$ python run.py -s -f sam -n 1
[array([[180.],
       [ 50.],
       [  0.]]), array([[190.],
       [ 50.],
       [  0.]]), array([[482.93880949],
       [ 32.2422165 ]]), array([[315.36604611],
       [-14.18027018]])]
Simulation Progress  100%
```

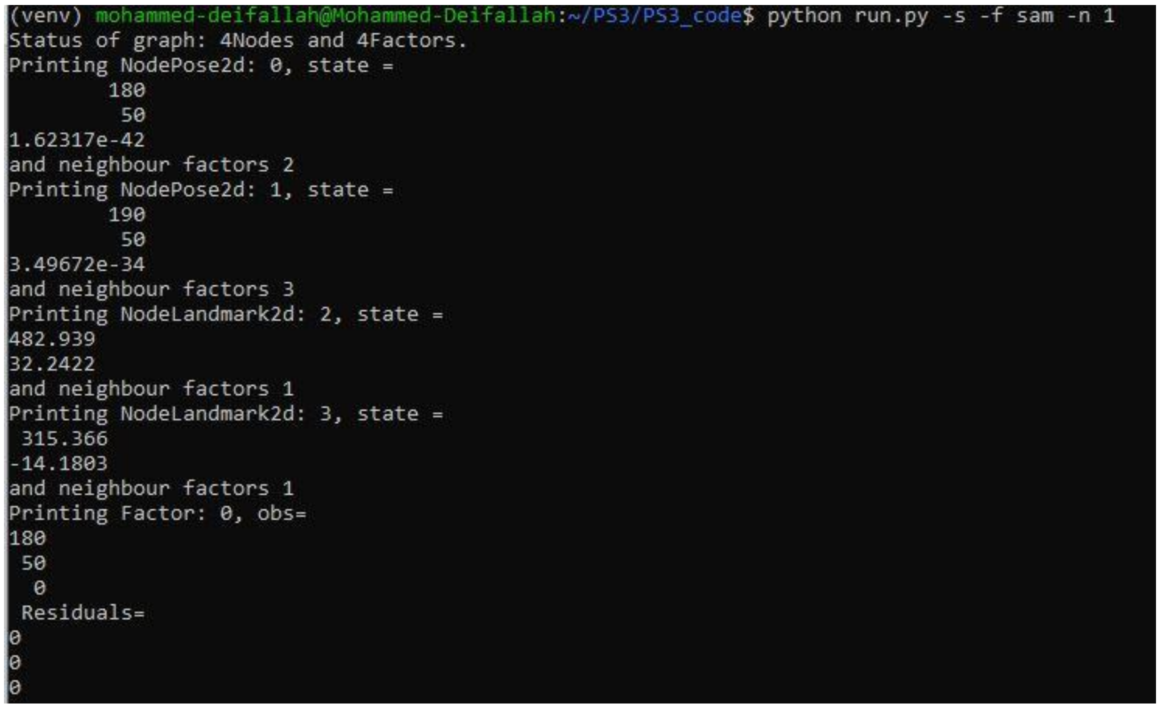
TASK 1.D

For this task, a new function is added to our SAM class called solve to wrap the actual functionality of mrob.FGraph.solve function. This encapsulation is applied in all upcoming subtasks to strength the OOP principles even if it’s not a big issue here. Also, it’s designed in such way as shown in the figure below, so that it can support different optimization methods:

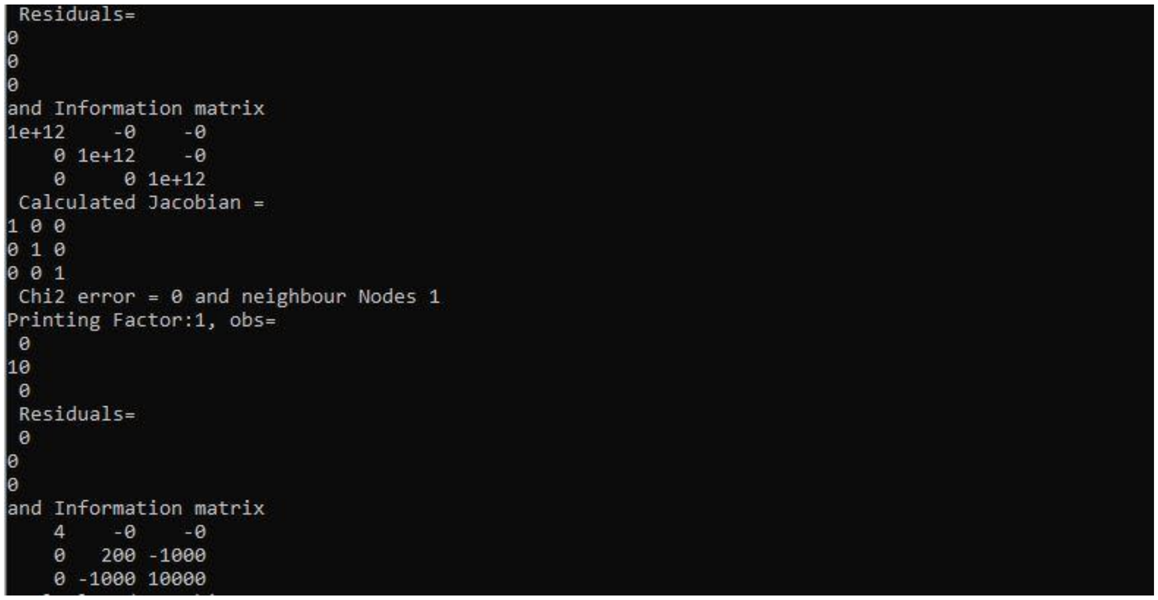


```
def solve(self, gn=True):
    if gn:
        self.graph.solve()
    else:
        self.graph.solve(method=mrob.LM)
    #self.graph.print(True) # --> Task 1.D
```

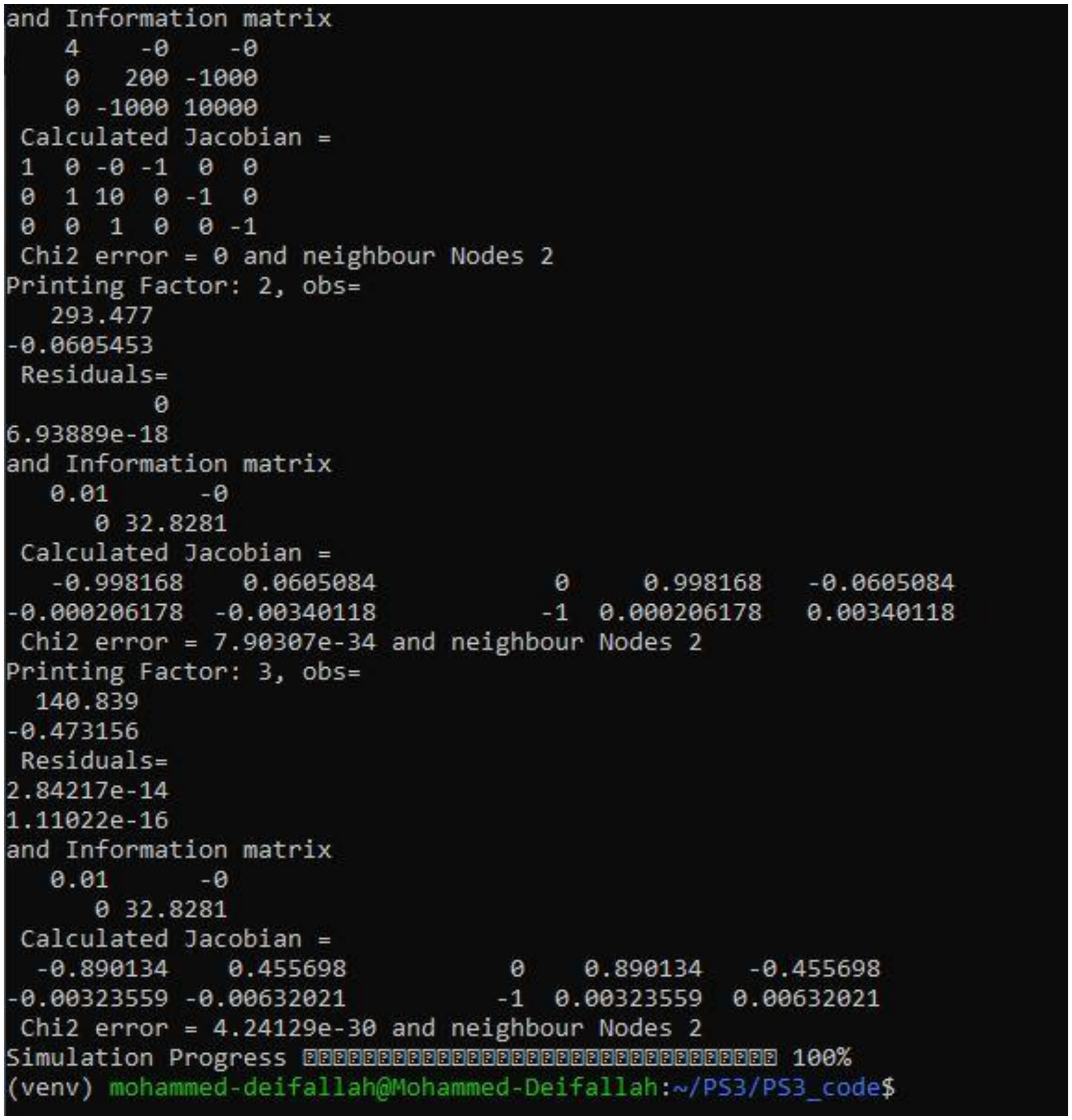
Uncommenting the commented line at the bottom prints the full information about the graph as follows:



```
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$ python run.py -s -f sam -n 1
Status of graph: 4Nodes and 4Factors.
Printing NodePose2d: 0, state =
        180
         50
1.62317e-42
and neighbour factors 2
Printing NodePose2d: 1, state =
        190
         50
3.49672e-34
and neighbour factors 3
Printing NodeLandmark2d: 2, state =
482.939
32.2422
and neighbour factors 1
Printing NodeLandmark2d: 3, state =
 315.366
-14.1803
and neighbour factors 1
Printing Factor: 0, obs=
180
 50
  0
 Residuals=
0
0
0
```



```
 Residuals=
0
0
0
and Information matrix
1e+12    -0    -0
    0 1e+12    -0
    0     0 1e+12
 Calculated Jacobian =
1 0 0
0 1 0
0 0 1
 Chi2 error = 0 and neighbour Nodes 1
Printing Factor:1, obs=
 0
10
 0
 Residuals=
 0
0
0
and Information matrix
    4    -0    -0
    0   200 -1000
    0 -1000 10000
```



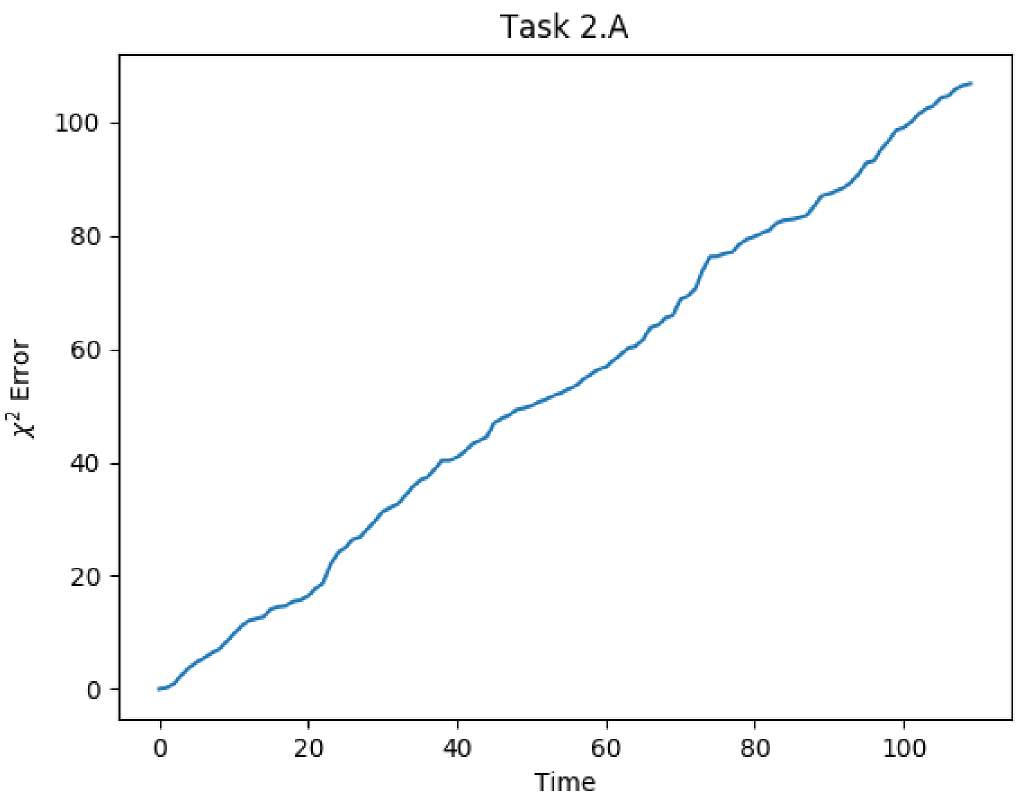
```
and Information matrix
    4    -0    -0
    0   200 -1000
    0 -1000 10000
 Calculated Jacobian =
 1  0 -0 -1  0  0
 0  1 10  0 -1  0
 0  0  1  0  0 -1
 Chi2 error = 0 and neighbour Nodes 2
Printing Factor: 2, obs=
   293.477
-0.0605453
 Residuals=
         0
6.93889e-18
and Information matrix
   0.01      -0
      0 32.8281
 Calculated Jacobian =
   -0.998168    0.0605084            0     0.998168   -0.0605084
-0.000206178  -0.00340118           -1  0.000206178   0.00340118
 Chi2 error = 7.90307e-34 and neighbour Nodes 2
Printing Factor: 3, obs=
  140.839
-0.473156
 Residuals=
2.84217e-14
1.11022e-16
and Information matrix
   0.01      -0
      0 32.8281
 Calculated Jacobian =
  -0.890134    0.455698           0    0.890134   -0.455698
-0.00323559 -0.00632021          -1  0.00323559  0.00632021
 Chi2 error = 4.24129e-30 and neighbour Nodes 2
Simulation Progress ████████████████████████████████ 100%
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$
```

Task 2: SAM Evaluation

For this part, the following command is used: ”python run.py -s -f sam -i slam-evaluation-input.npy -m sam.mp4” in order to load the data from the input file, unlike data generation approach which is followed in the previous part.

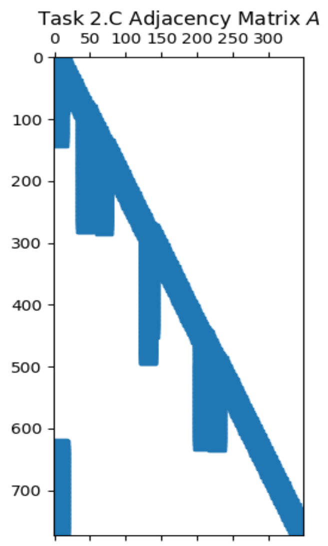
TASK 2.A

An array is added to the main loop to store the incremental values of χ^2 error over the time, calculated by a new wrapper function in our SAM class, then the following plot is drawn to show the increasing linear function of the error:



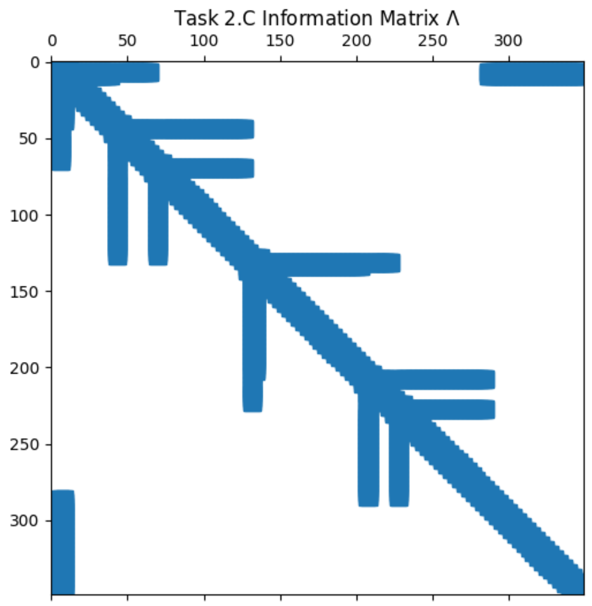
TASK 2.C

After the main loop is finished, a new plot is added to show the resultant adjacency matrix as shown:



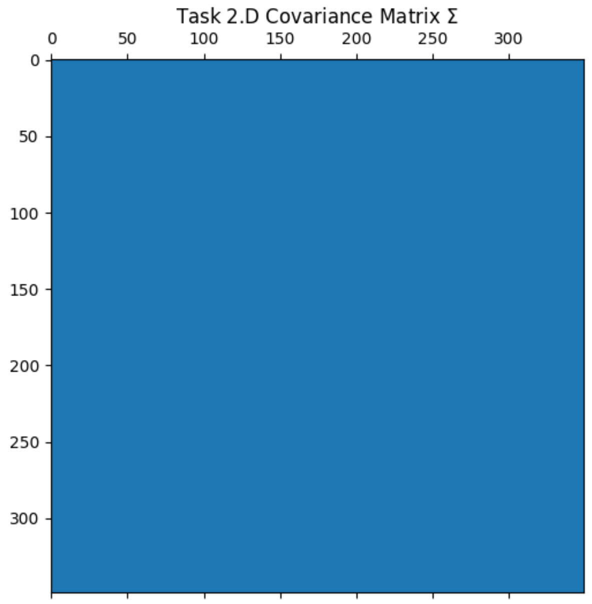
The graph shows how **<u>sparse</u>** the adjacency matrix is, as it's almost **diagonal**.

In addition to the adjacency matrix, a new graph is plotted to show the information matrix as follows:



TASK 2.D

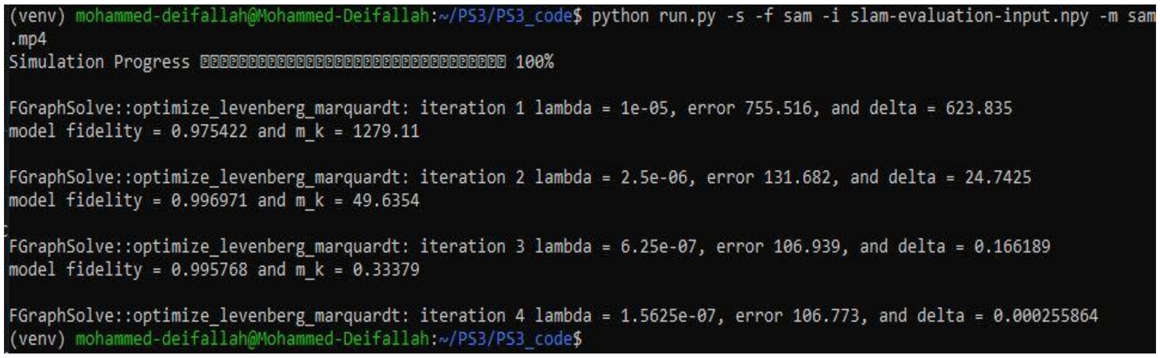
As the information matrix is shown to be significantly sparse, the covariance matrix Σ is supposed to be dense as mentioned in the lecture. This task asks to prove that in practice, and this is what results from plotting the inverse of the information matrix:



As shown in the figure above, it's a completely dense matrix.

TASK 2.E

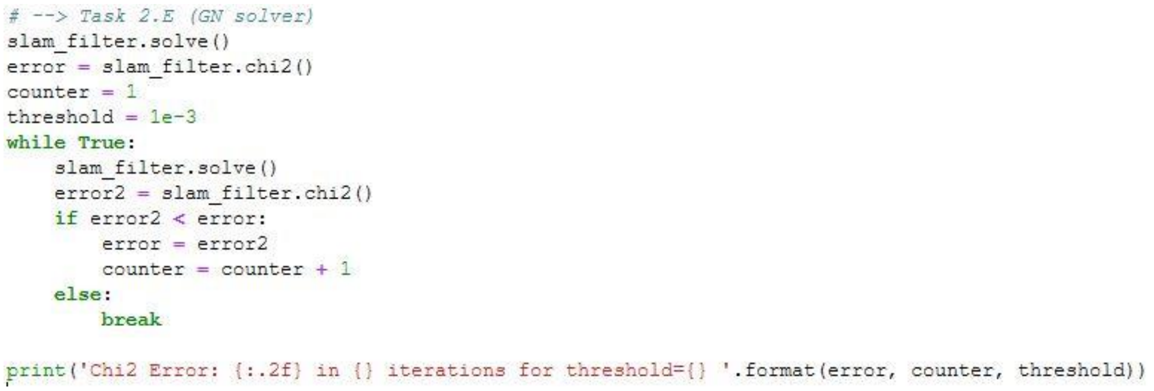
This was an interesting task for trying multiple optimizers till the algorithm converges. By solving the problem only once at the end using LM solver, the following console snippet shows the output:



```
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$ python run.py -s -f sam -i slam-evaluation-input.npy -m sam
.mp4
Simulation Progress ████████████████████████████████ 100%

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
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$
```

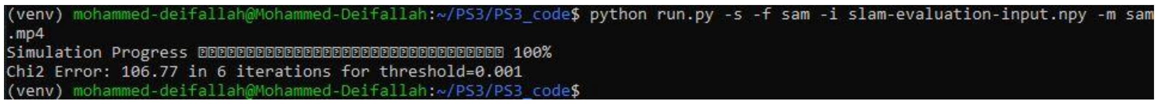
This briefly shows that the optimizer helps the algorithm to converge in only 4 iterations with χ^2 error ≈ 106.77.

Now, there’s an extra step, encouraged by Prof. Gonzalo and TA, to compare LM performance with the default GN solver. The following code snippet shows the python code used to conduct this experiment:



```python
# --> Task 2.E (GN solver)
slam_filter.solve()
error = slam_filter.chi2()
counter = 1
threshold = 1e-3
while True:
    slam_filter.solve()
    error2 = slam_filter.chi2()
    if error2 < error:
        error = error2
        counter = counter + 1
    else:
        break

print('Chi2 Error: {:.2f} in {} iterations for threshold={} '.format(error, counter, threshold))
```

The algorithm is assumed to converge when it can’t achieve less error in the current iteration. Thus, the following screenshot shows the output of the last print statement:



```
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$ python run.py -s -f sam -i slam-evaluation-input.npy -m sam
.mp4
Simulation Progress □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ 100%
Chi2 Error: 106.77 in 6 iterations for threshold=0.001
(venv) mohammed-deifallah@Mohammed-Deifallah:~/PS3/PS3_code$
```

This explicitly shows that the algorithm converges approximately with the same error which is achieved by LM solver. However, GN converges in 6 iterations while LM does so in only 4 iterations, which proves LM outperformance in our case.

P.S. The last subtask (GN Investigation) is optional as Prof. Gonzalo stated, so I hope it will be graded as bonus :)