Wireless Networks
Lab: Range-Free Localization

Nanjing University, Spring, 2023

I. Introduction

In this assignment, you will use range-free localization algorithms introduced in our lecture to calculate locations of sensor nodes. Range-free localization algorithms use the neighboring relationship to localize sensors. Given that two sensor nodes can communicate with each other, the distance between these two sensors must be smaller than the communication range, *e.g.*, $R = 1$. Using this fact, we can calculate the relative positions for nearby sensors using range-free localization algorithms. In this assignment, you will try to implement two of these localization algorithms, Graph-Laplacian based algorithm and Multi-Dimensional Scaling (MDS) based algorithm.

You will use MATLAB (or Octave)/Python to run simulations on randomly generated data sets in this assignment. If you are not familiar with MATLAB/Python, please use Google to find "Matlab tutorials" or "Python tutorials" on the web. You can either install MATLAB on your computer or use MATLAB online to complete this lab. The simulation code frameworks will be given to you, and you need to fill in your algorithms in certain functions to finish the assignment.

Tip 1: You are encouraged to use Google to find out terms, concepts or even partial solutions for this assignment.

MATLAB source files are included in the matlabsource directory of this assignment:

- `Localization_main.m`: the main script for this assignment. The code is given to you.
- `generate_random_network.m`: generate a random sensor network. The code is given to you.
- `drawconnection.m`: draw the communication graph of a sensor network. The code is given to you.
- `getedges.m`: get the edge list and adjacency matrix of a sensor network. The code is given to you.
- `getanchor.m`: select the anchor node for device-free localization. The code is given to you.
- `balancenet.m`: calculate the balanced locations using the graph laplacian method. *You need to write your own code for this function.*
- `adjustweight.m`: adjust the weight used in the graph laplacian method. *You need to write your own code for this function.*
- `compareresults.m`: draw the difference between estimated location and the true location. The code is given to you.
- `mds.m`: calculate the location using the MDS method. *You need to write your own code for this function.*
- `gradientdescent.m`: linear regression with gradient descent used for in rotating and scaling of the MDS method. The code is given to you.

Python source files are included in the pythonsource directory of this assignment:

- `localization_main.py`: the main script for this assignment. The code is given to you.
- `util_loc.py`: the utility functions used in this assignment, including generating a random sensor network, drawing results, and linear regression. The code is given to you.
- `lab2step2.py`: solve the problem using the graph laplacian method. *You need to write your own code for this function.*
- `lab2step3,py`: solve the problem using the MDS method. *You need to write your own code for this function.*

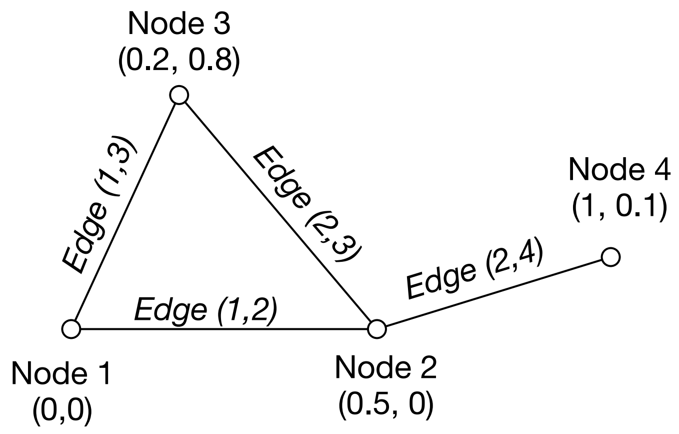


Fig. 1. Example for UDG of four nodes

II. Preliminary and Data Structure

Our assignment has a main script called `Localization_main.m` or `localization_main.py`, which will guide you through the assignment.

A. Generate a Random Network

The first step for our simulation is to generate a random sensor network, where the locations of sensors are randomly selected in a rectangular area. We provide sample codes to get random (x, y) coordinates for n sensor nodes. For example, use

```
true_loc = generate_random_network(500,10,10);
```

to generate $n = 500$ sensors randomly distributed in a 10×10 square region. The output `true_loc` will be a 500×2 matrix for the true locations of the sensor nodes, with the first column to be the x coordinates and the second column to be the y coordinates for the 500 nodes. There is a similar function in Python.

B. Graph Representations

The second step is calculating the communication graph for the generated network. Given the true locations of sensors, we build a Unit Disk Graph (UDG) to represent the communication network between sensor node. Every sensor node corresponds to a vertex in the UDG. If two sensor nodes are within the communication range, which is $R = 1$ in our case, then we add an undirected edge on the UDG to connect the corresponding vertices in the graph. We don't connect the node to itself. Figure 1 shows the UDG of four nodes. The number below the nodes are the (x, y) coordinates for sensors. There are also four edges that connects sensor nodes that have a distance smaller than 1 to each other.

There are many ways to represent a graph. In this assignment, we use two types of graph representation. The first one is an edge list, which lists all the edges (i, j) in the UDG, where i is the tail vertex and j is the head vertex of the edge. For the graph in Figure 1, we have the edge list as: $(1, 2)$, $(1, 3)$, $(2, 3)$ and $(2, 4)$. For an undiected graph, we include edges for both directions in the list, *e.g.*, include both $(1, 2)$ and $(2, 1)$ in the list.

The second graph representation method used in our assignment is a node-node adjacency matrix. The adjacency matrix $\mathbf{A}$ is a $n \times n$ matrix, with element $\mathbf{A}_{ij}$ to be the weight of edge (i, j). If there is no edge between node i and j, we normally set the weight to 0. For example, the adjacency matrix for the graph in Figure 1 is:

$$\begin{bmatrix} 0 & 0.500 & 0.825 & 0 \\ 0.500 & 0 & 0.854 & 0.510 \\ 0.824 & 0.854 & 0 & 0 \\ 0 & 0.510 & 0 & 0 \end{bmatrix} \tag{1}$$

In MATLAB or Python, we usually use sparse matrix to represent an adjacency matrix, where edges that do not appear in the graph are omitted in the sparse matrix. You are given an example code `getedges(true_loc,range)` that can return the edge list and adjacency matrix given the node locations and communication range.

C. Find Anchor Nodes

To determine sensor locations, range-free localization algorithms need to know the locations of some sensor nodes, which are called *anchor nodes*. With the locations of these anchor nodes, other nodes can use their relative positions to the anchors to determine their absolute location. For example, we may deploy some sensor nodes that are equipped with GPS, so that other nodes can use these nodes as anchors. For the algorithms in our assignment, it is better to put anchor nodes on the edge of the network. We give you an example `getanchor(true_loc,anchor_num);`, which can pick $4 \times anchor\_num$ anchors nodes on the edge of the network, with $anchor\_num$ anchors on each side of the network.

After all the preparations have been done, the main script will draw a connection graph as shown in Figure 2. The green points are non-anchor sensors and the red points are the anchors, and gray lines are edges.

The goal of our assignment is to use the position of anchor nodes and the edge list to find out the positions of these green non-anchor sensor nodes.

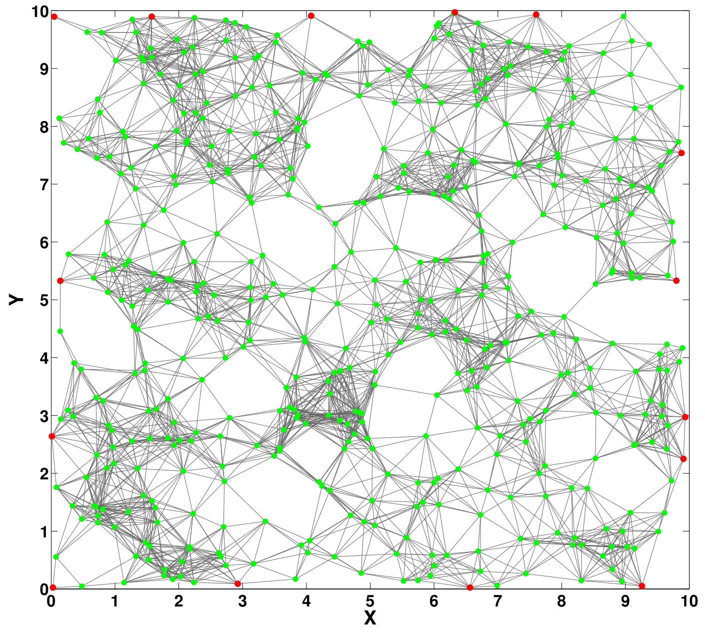


Fig. 2. Connection graph for a 500 nodes network.

III. Graph Laplacian based Algorithm

A. *Intuition*

The first approach that we used for device-free localization is graph laplacian based. The basic intuition behind this algorithm is to assume that every edge in Figure 2 is a spring that connects two sensors. These springs will try to drag sensors that are with in their communication range closer to each other. If we nail the anchor nodes at their fixed position, the forces of these springs will finally balance and each node will stay at a balanced location. These balanced locations will be the estimated location for sensor nodes. As sensors within communication range are dragged closer by springs, these estimated locations will be close to the true locations. For example, suppose we treat node 1, 3 and 4 in Figure 1 as anchor nodes, and replace edges with springs. The balanced location for node 2 is shown in Figure 3. We can see that this location is quite close to the true position of node 2. Given that there are more connections in dense networks, it is possible to use the connection information to determine the position of nodes.

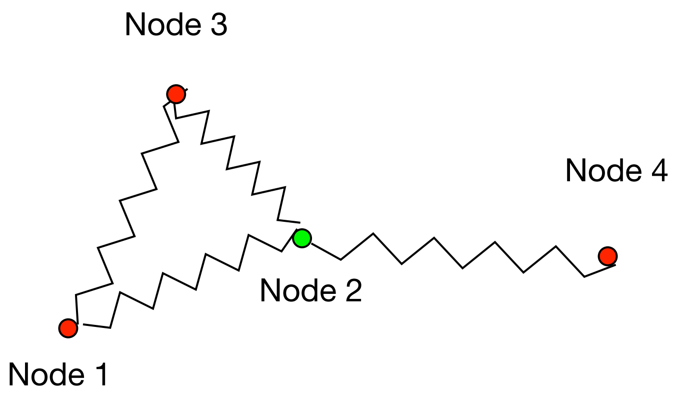


Fig. 3. Node 2 on the balanced location determined by the spring

We next consider how to calculate the balanced state of the spring network. Recall your high school physics, by Hooke's Law, if we have a spring with stiffness of k, the force on the spring is $F = -kd$ when the spring is extended by a distance of d. The potential energy stored in this spring is given by $E = \frac{1}{2}kd^2$. A balanced spring network will have the lowest potential energy. Therefore, we need to find the positions for sensor nodes that give the smallest sumation of the potential energy. For one spring between node i and j, we have its potential energy as:

$$E_{ij} = \frac{1}{2}k_{ij}\left((x_i - x_j)^2 + (y_i - y_j)^2\right) \tag{2}$$

where k_{ij} is the stiffness of the edge between node i and j and (x_i, y_i) are the coordinates for node i. Therefore our goal is to find a set of (x_i, y_i) for all non-anchor nodes that can minimize the overall potential energy:

$$\sum_{(i,j)\in \, edge\, list} E_{i,j} = \sum_{(i,j)\in \, edge\, list} \frac{1}{2}k_{ij}\left((x_i - x_j)^2 + (y_i - y_j)^2\right) \tag{3}$$

This problem is a convex optimization problem that can be solved easily. Recall your undergraduate calculus, if you want to minimize a function, you need to set the partial derivations to 0.

B. Example

Take the network in Figure 3 for example, we have:

$$\begin{aligned}\sum_{(i,j)} E_{i,j} &= \frac{1}{2}k_{12}\left((x_1-x_2)^2+(y_1-y_2)^2\right)+\frac{1}{2}k_{13}\left((x_1-x_3)^2+(y_1-y_3)^2\right)\\ &+\frac{1}{2}k_{23}\left((x_2-x_3)^2+(y_2-y_3)^2\right)+\frac{1}{2}k_{24}\left((x_2-x_4)^2+(y_2-y_4)^2\right) \qquad (4)\end{aligned}$$

If we want to find the balanced point (x_2, y_2) for node 2, we take partial derivation as:

$$\begin{aligned}\frac{\partial\sum_{(i,j)}E_{i,j}}{\partial x_2} &= \frac{1}{2}k_{12}\left(-2x_1+2x_2\right)+0\\ &+\frac{1}{2}k_{23}\left(-2x_3+2x_2\right)+\frac{1}{2}k_{24}\left(-2x_4+2x_2\right) \qquad (5)\\ \frac{\partial\sum_{(i,j)}E_{i,j}}{\partial y_2} &= \frac{1}{2}k_{12}\left(-2y_1+2y_2\right)+0\\ &+\frac{1}{2}k_{23}\left(-2y_3+2y_2\right)+\frac{1}{2}k_{24}\left(-2y_4+2y_2\right) \qquad (6)\end{aligned}$$

Setting $\frac{\partial\sum_{(i,j)}E_{i,j}}{\partial x_2}=0$ and $\frac{\partial\sum_{(i,j)}E_{i,j}}{\partial y_2}=0$, we have two linear equations:

$$\frac{1}{2}k_{12}\left(-2x_1+2x_2\right)+\frac{1}{2}k_{23}\left(-2x_3+2x_2\right)+\frac{1}{2}k_{24}\left(-2x_4+2x_2\right)=0 \qquad (7)$$

$$\frac{1}{2}k_{12}\left(-2y_1+2y_2\right)+\frac{1}{2}k_{23}\left(-2y_3+2y_2\right)+\frac{1}{2}k_{24}\left(-2y_4+2y_2\right)=0 \qquad (8)$$

Now let's rearrange Equations (7) and (8) by changing k_{12} to k_{21} and collecting terms:

$$(-k_{21})x_1+(k_{21}+k_{23}+k_{24})x_2+(-k_{23})x_3+(-k_{24})x_4=0 \qquad (9)$$

$$(-k_{21})y_1+(k_{21}+k_{23}+k_{24})y_2+(-k_{23})y_3+(-k_{24})y_4=0 \qquad (10)$$

Solving these two equations with known (x_1, y_1), (x_3, y_3) and (x_4, y_4) of anchor nodes, we can get the location (x_2, y_2) of node 2.

C. General Case

If you take partial derivations on Equation (3) and set them to zero, you will get a set of equations which can be written in the matrix form as:

$$\begin{bmatrix}\sum_j k_{1j} & -k_{12} & \dots & -k_{1n}\\ -k_{21} & \sum_j k_{2j} & \dots & -k_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ -k_{n1} & -k_{n2} & \dots & \sum_j k_{nj}\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ \vdots\\ x_n\end{bmatrix}=\begin{bmatrix}0\\ 0\\ \vdots\\ 0\end{bmatrix} \qquad (11)$$

$$\begin{bmatrix}\sum_j k_{1j} & -k_{12} & \dots & -k_{1n}\\ -k_{21} & \sum_j k_{2j} & \dots & -k_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ -k_{n1} & -k_{n2} & \dots & \sum_j k_{nj}\end{bmatrix}\begin{bmatrix}y_1\\ y_2\\ \vdots\\ y_n\end{bmatrix}=\begin{bmatrix}0\\ 0\\ \vdots\\ 0\end{bmatrix} \qquad (12)$$

when we treat $k_{ij} = k_{ji}$ and missing edges has k equal to 0. You can verify that the example in Equations (9) and (10) confirm with the above results.

The matrix on the left side of Equations (11) and (12) are the same, and it is called Graph Laplacian Matrix. The Laplacian Matrix can be calculated by the following equation:

$$\mathbf{L}=\mathbf{D}-\mathbf{A} \qquad (13)$$

where $\mathbf{A}$ is the adjacency matrix and $\mathbf{D}$ is a diagonal matrix with diagonal elements equal to the sum of each row in $\mathbf{A}$.

We need to make some special treatments on the anchor nodes so that their locations are not changed. This can be done by simply replace the equation corresponds to the anchor to be $x_a = x_a^*$, where x_a^* is the true x coordinate of anchor a. In the matrix form, we can simply set the row corresponds to the anchor a to be all zero, except change the element (a, a) equal to 1; and then place x_a^* or y_a^* on the correspond row in the right side vector. For example, we can rewrite Equation (9) in the matrix form as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -k_{21} & \sum_j k_{2j} & -k_{23} & -k_{24} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_1^* \\ 0 \\ x_3^* \\ x_4^* \end{bmatrix} \tag{14}$$

The three rows corresponds to the anchors are replaced to fix the anchor's location.

The coordinates for the non-anchor nodes can be calculated by solve the above equations, which can be done simply through matrix inversion in MATLAB or Python.

D. *Stiffness Adjustment*

In the initial state, we can choose to set the stiffness of all edges k_{ij} equal to 1. However, the estimated results contains large errors, as shown in Figure 4(a). This is due to the springs that connects to the anchors are stretched too much. Therefore, we need to adjust the stiffness of these springs.

The basic idea for adjusting the stiffness of springs is that if the spring is stretched too long, we need to increase its stiffness so that it can shrink its length. On the other hand, if the spring is too short, we can reduce its stiffness so that it will be stretched. Therefore, we propose to adjust the stiffness k_{ij} for all edges using the distance of the edge based on the estimated locations. For example, suppose that an edge (i, j) has weight of k_{ij}^w in the wth iteration. We can adjust the weight as follows:

$$k_{ij}^{w+1} = k_{ij}^w \frac{\sqrt{(x_i^w - x_j^w)^2 + (y_i^w - y_j^w)^2}}{R} \tag{15}$$

where (x_i^w, x_j^w) is the estimated location at wth iteration. Therefore, $\sqrt{(x_i^w - x_j^w)^2 + (y_i^w - y_j^w)^2}$ is the length of the edge (i, j) at wth iteration. In this way, the lengths of edges will iteratively converge to the communication range of R.

The iterative stiffness adjustment method works quite well. We can see from Figure 4(b), the node distribution becomes uniform after only five iterations. Figure 4(c) shows the location error, by comparing the estimated position (in black) to the true location (in green). We can see that the error for nodes at the center are quite small and most errors occurs at the edge where no anchors are around to "pull" the sensors.

E. *Tasks for laplacian method*

You are required to write your own code for the Graph Laplacian calculation, stiffness adjustments in this assignment. Namely, you should complete two functions:

- `balancenet`
- `adjustweight`

Tip 2: The algorithm may be hard to debug. Try to output intermittent results of your algorithm. You may also start from small networks, *e.g.*, network with 50 nodes in 3×3 area.

Furthermore, you need to write your own code to quantitatively evaluate the localization performance. You should at least evaluate the following three aspects for you algorithm.

- *Average localization error*: You should calculate the distance between the estimated location and the true location of each non-anchor node. Then, give the average of the error over all non-anchor nodes.
- *Cumulative Distribution Function (CDF) for localization error*: Draw a CDF plot of localization error.
- *Running time:* Your should calculate the running time of your algorithm and record it.

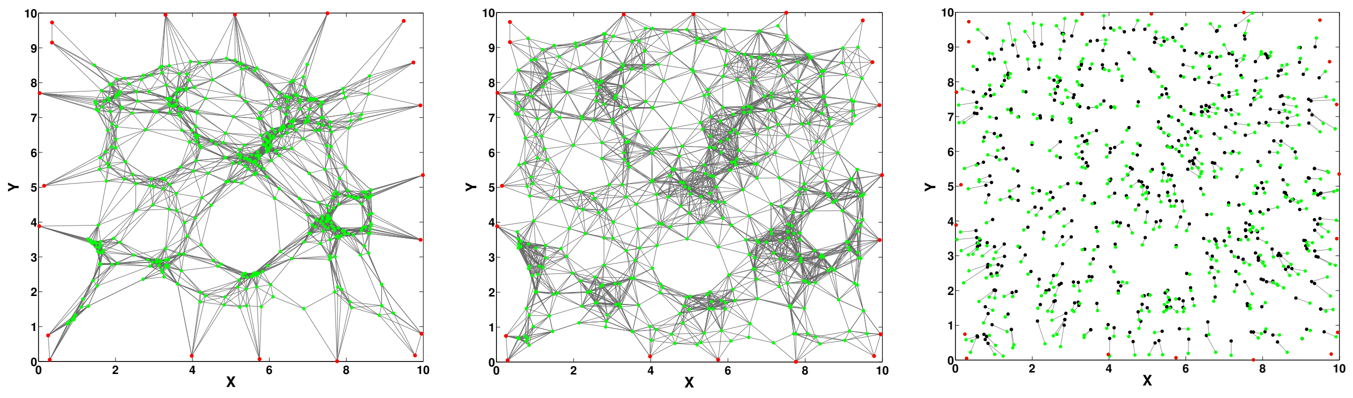
All the metrics should be evaluated by averaging over at least 100 randomly generated networks. You may need to write your own evaluation code for this task. For example, you may need to remove the `pause` in the code to let it run without human intervention. You may also modify the code to run in batches so that you can quickly get the statistics of your results.

Tip 3: MATLAB has functions to draw CDF plots.

Tip 4: Loops are not efficient in MATLAB or Python. Normally our algorithms finish within less than one second for a network with 500 nodes. If your algorithm is too slow, try reimplement it using matrix operations instead of loops.

Tip 5: When the network is not connected, the sample code may not work. Find a solution to detect and handle the cases when the randomly generated network is not connected.

Tip 6: The gradient decent algorithm in the sample code is just for demonstration, you can use other methods to solve the optimization problem.



(a) First iteration result (b) After five rounds of stiffness adjustment (c) Finial error graph

Fig. 4. Stiffness adjustment results

IV. MDS BASED ALGORITHM

The second range-free localization algorithm is the Multi-Dimensional Scaling (MDS) based algorithm [1], [2]. The basic idea for MDS is to find the relative location of the sensor nodes given their distances to each other. The distance between sensors can be approximated by their communication graph. In other words, we assume sensors within communication range have distance 1 to each other and 2-hop neighbors have distance 2 to each other, *etc.*

A. Distance matrix calculation

The first step for MDS based algorithm is to calculate the distance between every pair of sensors. We use the shortest path distance as the distance between sensors. We treat nodes connected by edges have distance of 1 to each other. The result of shortest path distance should be all integer, with distance to the node itself as 0. If the graph is not connected, some pair of nodes will have infinity distance to each other. Our algorithm cannot work for this case.

Given the adjacency matrix of the UDG, it is easy to use MATLAB or Python to calculate the shortest path distance between all pairs of sensors. Suppose that you store the shortest path length in a $n \times n$ matrix $\mathbf{D}$, with $\mathbf{D}_{ij}$ to be the length of shortest path between node i and j.

Tip 5: You can use the MATLAB function provided by the graph toolbox to calculate the shortest paths for the network. For Python, the SciPy library also has csgraph tools to handle shortest path problems.

B. MDS process

Given the shortest path distance matrix $\mathbf{D}$, MDS outputs a $n \times 2$ matrix $\mathbf{X}$ of node coordinates, such that $\mathbf{X}_{i1} = x_i$ and $\mathbf{X}_{i2} = y_i$ are the estimated coordinates of sensor node i. The output coordinates satisfies the distance relationship given by the shortest path distance matrix $\mathbf{D}$, *i.e.*, $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} = \mathbf{D}_{ij}$. Furthermore, MDS assumes that $\sum_i x_i = 0$ and $\sum_i y_i = 0$, so that the centroid of the output coordinates is at $(0, 0)$. We define a new $n \times n$ matrix $\mathbf{Y}$ as:

$$\mathbf{Y} \;=\; \mathbf{X}\mathbf{X}^T \tag{16}$$

where $\mathbf{X}^T$ is the transpose of $\mathbf{X}$.

For example, for a three node network, the matrix $\mathbf{X}$ and $\mathbf{Y}$ will be:

$$\mathbf{X} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} x_1^2 + y_1^2 & x_1x_2 + y_1y_2 & x_1x_3 + y_1y_3 \\ x_2x_1 + y_2y_1 & x_2^2 + y_2^2 & x_2x_3 + y_2y_3 \\ x_3x_1 + y_3y_1 & x_3x_2 + y_3y_2 & x_3^2 + y_3^2 \end{bmatrix} \tag{17}$$

We have:

$$\mathbf{D}_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 = \mathbf{Y}_{ii} + \mathbf{Y}_{jj} - 2\mathbf{Y}_{ij}. \tag{18}$$

From Equation (18), we have:

$$\mathbf{Y}_{ij} = -\frac{1}{2}\left(\mathbf{D}_{ij}^2 - \mathbf{Y}_{ii} - \mathbf{Y}_{jj}\right) \tag{19}$$

Note that with the assumption of $\sum_i x_i = 0$ and $\sum_i y_i = 0$, we have $\sum_i \mathbf{Y}_{ij} = 0$ and $\sum_j \mathbf{Y}_{ij} = 0$, *i.e.*, the sum of rows and columns of matrix $\mathbf{Y}$ is zero. Therefore we have:

$$\begin{aligned}
& \frac{1}{n}\sum_i \mathbf{D}_{ij}^2 + \frac{1}{n}\sum_j \mathbf{D}_{ij}^2 - \frac{1}{n^2}\sum_i\sum_j \mathbf{D}_{ij}^2 \\
= \; & \frac{1}{n}\sum_i (\mathbf{Y}_{ii} + \mathbf{Y}_{jj} - 2\mathbf{Y}_{ij}) + \frac{1}{n}\sum_j (\mathbf{Y}_{ii} + \mathbf{Y}_{jj} - 2\mathbf{Y}_{ij}) - \frac{1}{n^2}\sum_i\sum_j (\mathbf{Y}_{ii} + \mathbf{Y}_{jj} - 2\mathbf{Y}_{ij}) \\
= \; & \frac{1}{n}\left(\sum_i \mathbf{Y}_{ii} + n\mathbf{Y}_{jj} - 2\sum_i \mathbf{Y}_{ij}\right) + \frac{1}{n}\left(n\mathbf{Y}_{ii} + \sum_j \mathbf{Y}_{jj} - 2\sum_j \mathbf{Y}_{ij}\right) \\
& -\frac{1}{n^2}\left(n\sum_i \mathbf{Y}_{ii} + n\sum_j \mathbf{Y}_{jj} - 2\sum_i\sum_j \mathbf{Y}_{ij}\right) \\
= \; & \frac{1}{n}\left(\sum_i \mathbf{Y}_{ii} + n\mathbf{Y}_{jj} - 0\right) + \frac{1}{n}\left(n\mathbf{Y}_{ii} + \sum_j \mathbf{Y}_{jj} - 0\right) - \frac{1}{n^2}\left(n\sum_i \mathbf{Y}_{ii} + n\sum_j \mathbf{Y}_{jj} - 0\right) \\
= \; & \mathbf{Y}_{ii} + \mathbf{Y}_{jj}
\end{aligned} \tag{20}$$

Therefore, the elements of $\mathbf{Y}$ can be represented by values of $\mathbf{D}$ as:

$$\mathbf{Y}_{ij} = -\frac{1}{2}\left(\mathbf{D}_{ij}^2 - \frac{1}{n}\sum_i \mathbf{D}_{ij}^2 - \frac{1}{n}\sum_j \mathbf{D}_{ij}^2 + \frac{1}{n^2}\sum_i\sum_j \mathbf{D}_{ij}^2\right) \tag{21}$$

The matrix $\mathbf{Y}$ is a symmetric positive semidefinite matrix. Therefore, you can either use Eigendecomposition or Singular value decomposition (SVD) to find out the value for $\mathbf{X}$. MATLAB or SciPy (in linalg) provides both decompositon functions. For example, if we use SVD, we have:

$$\mathbf{Y} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{22}$$

where both $\mathbf{U}$ and $\mathbf{V}$ are $n \times n$ square matrixes and $\mathbf{\Sigma}$ is a $n \times n$ diagonal matrix. It can be shown that we can reconstruct $\mathbf{X}$ using:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}^{\frac{1}{2}} \tag{23}$$

where $\mathbf{\Sigma}^{\frac{1}{2}}$ take the square root of each element in $\mathbf{\Sigma}$. As we only need two columns of $\mathbf{X}$, we just need to include first two columns of $\mathbf{\Sigma}$ in the above operation.

In summary, the process for MDS is as follows:

1) Calculate the distance matrix $\mathbf{D}$ using the all-pair shortest path algorithm.
2) Use the distance matrix $\mathbf{D}$ to calculate matrix $\mathbf{Y}$.
3) Use SVD decomposition to decompose $\mathbf{Y} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$.
4) Calculate $\mathbf{X}$ using $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}^{\frac{1}{2}}$ by taking the first two columns of $\mathbf{\Sigma}$.

C. Rotating and Scaling

The result in $\mathbf{X}$ are only relative locations of sensors, with centroid at $(0, 0)$. If you plot the result of $\mathbf{X}$, possibly you will get a graph similar to Figure 5(a). To get the absolute location of these sensors, we need the helps of anchor nodes.

The basic idea is to rotate, scale and move the locations of these sensors. As we know, 2-D Cartesian coordinates can be rotated and moved through the following equations:

$$\hat{x}_i = \theta_{11}x_i + \theta_{21}y_i + \theta_{31} \tag{24}$$

$$\hat{y}_i = \theta_{12}x_i + \theta_{22}y_i + \theta_{32} \tag{25}$$

where $(\hat{x}_i, \hat{y}_i)$ are the coordinates after the transform. We can define:

$$\mathbf{\Theta} = \begin{bmatrix} \theta_{11} & \theta_{12} \\ \theta_{21} & \theta_{22} \\ \theta_{31} & \theta_{32} \end{bmatrix} \tag{26}$$

as the transform matrix. Thus, we can rewrite the transformation in the matrix form:

$$\hat{\mathbf{X}} = \begin{bmatrix} & 1 \\ \mathbf{X} & \vdots \\ & 1 \end{bmatrix} \begin{bmatrix} \theta_{11} & \theta_{12} \\ \theta_{21} & \theta_{22} \\ \theta_{31} & \theta_{32} \end{bmatrix} \tag{27}$$

where we add one column of all 1s in $\mathbf{X}$ before the transformation. To estimate the values of the six θ parameters, we can use the known locations of the anchors and find the set of θ values that can best transform the MDS result to the true locations of anchors.

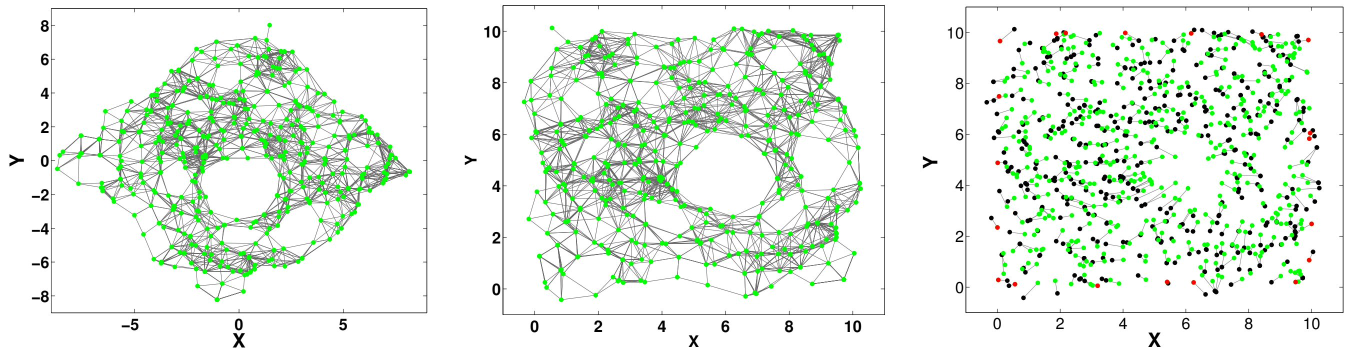
If you have taken the machine learning course, you will find the problem of finding the best θ values is exactly a Linear Regression problem that can be easily solved. We can use the known locations of the anchor as the training data to find $\mathbf{\Theta}$ and use the estimated $\mathbf{\Theta}$ to transform the MDS result. In the assignment, we give you the gradient descent code which can solve the linear regression problem for you. Figure 5(b) and 5(c) show the network after rotating and scaling, as well as the final error for the MDS method. We see that the MDS method has similar localization error as the Laplacian based method.

D. Tasks for MDS method

You are required to write your own code for the MDS calculation. Namely, you should complete the function:

- `mds.m`

Furthermore, you need to write your own code to quantitatively evaluate the localization performance. You should at least evaluate the following three aspects for you algorithm.



(a) Before rotating and scaling (b) After rotating and scaling (c) Finial error graph

Fig. 5. MDS based results

- *Average localization error*: You should calculate the distance between the estimated location and the true location of each non-anchor node. Then, give the average of the error over all non-anchor nodes.
- *Cumulative Distribution Function (CDF) for localization error*: Draw a CDF plot of localization error.
- *Running time:* Your should calculate the running time of your algorithm and record it.

All the metrics should be evaluated by averaging over at least 100 randomly generated networks. You may need to write your own evaluation code for this task.

V. Further experiments

After you have grasped the basics of the above two algorithms, you can try to explore different properties of these two algorithms. These further experimental results will account for 30% of your score of this assignment. We provide some basic ideas for further experiments:

- How would the running time change for different numbers of sensors node in the network? Does the running time for the two algorithms change in the same way?
- How would the algorithms perform when there is a "hole" in the network? Does these two algorithms perform in the same way? (*To make a hole in the network, you can intentionally remove all the nodes within a certain shape,* e.g.*, a circle with radius of 4 in the network.*)
- How would the algorithms perform when the density, *i.e.*, the number of nodes per unit area, changes?
- How would the algorithms perform when the number of anchor nodes changes?
- How would the algorithms perform when we randomly choose anchor nodes instead of put them on the edge?

The evaluation should be based on experimental results obtained through simulation via your codes. You should also explain why your results for different algorithms behave in that particular way. We recommend you to download and carefully read a few localization papers and focus on the experiments and evaluation section. Your goal is to write a evaluation section that is similar to those in research papers.

VI. Submission Requirements

Please submit both your code and your assignment report to the course web site before the deadline. Any late submission will be rejected. There is also a template for your assignment report in the handouts for this assignment.

VII. Acknowledgement

We would like to thank Mr. Zeng Shang and Mr. Hu Jiayu's efforts in developing this assignment.

References

[1] Y. Shang, W. Ruml, Y. Zhang, and M. P. Fromherz, “Localization from mere connectivity,” in *Proceedings of ACM Mobihoc*, 2003, pp. 201–212.

[2] Y. Shang and W. Ruml, “Improved MDS-based localization,” in *Proceedings of IEEE INFOCOM*, vol. 4, 2004, pp. 2640–2651.