Manifold Learning[*]

Ke Chen

To carry out this assessed coursework, you will need the Python notebook file, `manifold.ipynb`, which contain code to get you started with each of the assignments and the data files: `bars.npz` for the bar image dataset, `face_tenenbaum.npz` for the face image dataset, and `dist_18_cities.csv` for the airline distance data. Furthermore, you need to use the given initial embedded coordinates in files, `init_y_spairal.npz`, `init_val_airline_2d.npz` and `init_val_airline_3d.npz`, for initialisation in an MDS assignment. To help you work on ISOMAP assignments, you are also provided with our own implemented cMDS python extension, `mds.cp3x-win_amd64.pyd` for Microsoft Window, `mds.cpython-3x-x86_64-linux-gnu.so` for Linux or `mds.cpython-3x-darwin.so` for Mac OS, where 3x is the version of python (e.g. 37 for Python 3.7). All above are available in a zipped file on BlackBoard alongside this document.
Caveat: You may not display some resultant figures correctly with `%matplotlib notebook` which appears in the first line of the given Jupyter notebook file, especially on MS Windows. To deal with this issue, you need to use the Firefox browser instead of Chrome, Microsoft IE and Edge.

Manifold learning is one of the central themes in representation learning. In this coursework, you are asked to implement classical MDS (cMDS) algorithm in Python and apply your own cMDS and stress-based MDS implementations as well as the provided implementations of Sammon mapping, ISOMAP and LLE in Python to synthetic and real datasets for manifold learning.

Supportive Software

To do this coursework, you are provided with our own Python implementation of ISOMAP, LLE and visualisation tools required by this coursework. Also, you are provided with our cMDS implementation required in ISOMAP in case you cannot complete the cMDS implementation assignment.

Our implementation includes manifold learning algorithms and display functions in Python: optimization functions, `optimization.py`, isometric feature mapping, `isomap.py`, locally linear embedding, `lle.py`, display functions, `helpers.py`, dataset functions, `dataset.py`.

`optimization.py` enables you to carry out the conjugate gradient method and the gradient descent method required for the assignments regarding MDS.

To solve an optimization problem by using the conjugate gradient method, the signature of this function is as follows:

```
conjugate_gradient(D, x0, loss_f, grad_f, max_iter)
```

where `D` is a distance matrix between points in the original space and `x0` is an initial value we want to get. `loss_f` and `grad_f` are the loss function and its first derivative or gradient, respectively. `max_iter` is the maximum number of iteration.

[*] **Assessed Coursework**: the deadline and requirements can be found at the end of this document.

For example, given N points in the target space, the distance matrix is $D_{N\times N}$ and the initial values of the embedded coordinates is x0, e.g., a matrix of size $2 \times N$ for the projection to 2-D target space, this function can be set as follows:

```
conjugate_gradient(D, x0, loss_sammon, grad_sammon, 1000)
```

where Sammon mapping is used and max_iter is 1000. This function returns the optimal 2-D embedded coordinates of all the points in D.

Similarly, the signature of the gradient descent function is as follows:

```
gradient_descent(D, x0, loss_f, grad_f, lr, tol, max_iter)
```

where lr is a learning rate (step size) and tol is tolerance used to specify a criterion for early stopping and the remaining parameters are same as those of the conjugate descent function.

isomap.py allows you to apply ISOMAP to relevant tasks as described in the assignments, while lle.py provides you with the LLE function.

The signature of isomap function is as follows:

```
isomap(x, n_components, **kwargs)
```

where x refers to data points of the original space which are column vectors and n_components is the dimensionality of the target space and kwargs is additional parameters. kwargs is a set of parameters that can be either (dist_type, epsilon, dist_func, dist_func) for ϵ nearest neighbourhood or (dist_type, n_neighbors, dist_func, cmds_func) for K nearest neighbourhood.

Each parameter of kwargs is defined as follows:

- dist_type: distance type. It can be set either "nearest" or "radius".
- epsilon: the value of radius. This is only used when the dist_type is set as "radius".
- n_neighbors: the number of nearest neighbours. This is only used when the dist_type is set as "nearest".
- dist_func: the function for distance calculation. This can be set dist_nearest_neighbor in Code.isomap and fixed_radius_distance in **Assignment 4**.
- cmds_func: the function of classical multidimensional scaling (cMDS). This can be set cmds in **Assignment 1** or the mds module which is provided as python extension.

For example, when a K nearest neighbour (KNN) distance or ϵ nearest neighbour (ϵNN) distance s used, this function is used as follows:

```
from Code.isomap import isomap, dist_nearest_neighbor

points = .... # data point in column vector

n_components = 2
n_neighbors = 6
Y, dist, predecessors = isomap(points,
                               n_components,
                               dist_type='nearest',
                               n_neighbors=n_neighbors,
                               dist_func=dist_nearest_neighbor,
                               cmds_func=cmds)
```

or

```
from Code.isomap import isomap

points = .... # data point in column vector

n_components = 2
epsilon = 3.5
Y, dist, predecessors = isomap(points,
                               n_components,
                               dist_type='radius',
                               epsilon=epsilon,
                               dist_func=fixed_radius_distance,
                               cmds_func=cmds)
```

where `Y` is the embedded coordinates in column vectors, `dist` is the distance matrix and `predecessors` is the index of predecessors for the shortest path which is the result of `sparse.csgraph.shortest_path` from scipy package.

The signature of `lle` function is like follows:

```
lle(data, n_components=2, n_neighbors=None, epsilon=None, reg_func=None)
```

where `data` is data points in the original space, `n_components` is the dimensionality of target embedded space, `n_neighbors` is the number of neighbours for KNN, `epsilon` is the value of fixed radius for ϵ-distance, and `reg_func` is a regularization term to avoid the singular case.

Below, you can find two examples on how to use the `lle` function.

```
from Code.lle import lle

data = ... # data point in the original space

n_dim = 2
k = 7
Y = lle(data, n_components=n_dim, n_neighbors=k, reg_func=reg_func)
```

```
from Code.lle import lle

data = ... # data point in the original space

n_dim = 2
e = 5.3
Y = lle(data, n_components=n_dim, epsilon=e, reg_func=reg_func)
```

where KNN is used for the first case and ϵ-NN is used for the second case. In terms of `reg_func`, `None` will be fine except for **Assignment 7** where **reg_func(C, K)** is provided in `manifold.ipynb`.

`helpers.py` enables you to visualise the result of each assignment. This Python module contains `VIS`, `VIS_Shortest_path_2d`, `ImageViewer`, `VIS_Bars` classes.

`VIS` is used for displaying the embedded coordinates along with showing images when you click the points in the plot. Its constructor is as follows:

`VIS(data, proj, fig_vis, img_size=(28,28), cmap='gray')`

where `data` refer to data points in the source space, `proj` refer to the correponding embedded points in the target space, `fig_vis` is figure object created by Python built-in function `figure()` in the matplotlib package, `img_size` is the size of images which are located in the first parameter, `data`, and `cmap` is colour map which is used for displaying the images (e.g. "gray", "gray_r". For the details of cmap, please refer to the documentation of matplotlib package).

For example, this constructor can be used as follows:

```
import matplotlib.pyplot as plt

data = ... # data point in the original space
proj = lle(data, ...) # embedded points in the target space
fig_vis = plt.figure()
img_size = (28, 28)

VIS(data, proj, img_size, cmap='gray')
```

`VIS_Shortest_path_2d` is for displaying not only embedded points but also the shortest path between two specified data points. Its constructor is as follows:

`VIS_Shortest_path_2d(proj, dist, predecessors, fig_vis)` where `proj` refer to the embedded points in the target space, `dist` and `predecessors` are distance matrix and the index of precessors for shortest path which are obtained from `isomap` function. `fig_vis` is figure object created by the Python built-in function `figure()` in the matplotlib package. The usage of this class is as follows:

```
import matplotlib.pyplot as plt
from Code.isomap import isomap, dist_nearest_neighbor

points = .... # data point in column vector

n_components = 2
n_neighbors = 6
Y, dist, predecessors = isomap(points,
                               n_components,
                               dist_type='nearest',
                               n_neighbors=n_neighbors,
                               dist_func=dist_nearest_neighbor,
                               cmds_func=cmds)

fig = plt.figure()
VIS_Shortest_path_2d(Y, dist, predecessors, fig)
```

The embedded points will be plotted, so you can see the shortest path by selecting two points.

`ImageViewer` is for displaying images on the shortest path and its constructor is as follows:

`ImageViewer(data, index, image_size, fig_vis, max_row=5)`

where `data` refer to data points in the source space, `index` refer to the indices of points on the shortest path, `image_size` is the size of images which are in the first parameter, `data`, `fig_vis` is a figure object created by Python built-in function, `figure()`, in the matplotlib package, and `max_row` is the maximum number of images in a row. The usage of this class is as follows:

```
import matplotlib.pyplot as plt

data = ... # data points in the original space
image_size = [64,64]

start_idx = 1
end_idx = 100
path = get_shortest_path(predecessors, start_idx, end_idx)

fig = plt.figure()
img_viewer = ImageViewer(data, path, image_size, fig, 5)
img_viewer.show()
```

where `get_shortest_path` is provided in `manifold.ipynb` for **Assignment 5** and `predecessors` is returned from the `isomap` function.

`VIS_Bars` allows you to display the bar images in the dataset, bars.npz, and its result yielded by the `lle` function. Its constructor is as follows:

```
VIS_Bars(data, proj, fig_vis, color='r', image_size=(40,40), both=True)
```

where `data` are points in the source space, `proj` are the embedded points yielded by `lle`, `fig_vis` is a figure object created by `figure()` in the matplotlib package, `color` is the value of colour to represent the embedded points in this colour (e.g. 'r' for red, 'b' for blue), `image_size` is the size of images specified in `data`, and `both` is the flag to indicate whether both horizontal bars and vertical bars appear in `data` or not. If the last parameter, `both`, is set as True, the colour parameter, `color`, will be ignored. The usage of this class is as follows:

```
from Code.dataset import bars

data_bar, centers = bars()
data_bar = data_bar.T
centers = centers.T
image_size = [40,40]

n_neighbors = 5
n_components = 2
Y_bar = lle(data_bar, n_components=n_components,
            n_neighbors=n_neighbors, reg_func=reg_func)

fig_bar = plt.figure()
vis_both = VIS_Bars(data=data_bar, proj=Y_bar, fig_vis=fig_bar,
                    image_size=image_size, both=True)
```

or

```
from Code.dataset import bars

data_bar, centers = bars()
data_bar = data_bar.T
centers = centers.T
image_size = [40,40]

n_neighbors = 5
n_components = 2
Y_bar = lle(data_bar, n_components=n_components,
            n_neighbors=n_neighbors, reg_func=reg_func)

fig_bar = plt.figure()
vis_v = VIS_Bars(data=data_bar[:,:500], proj=Y_bar[:,:500],
                 fig_vis=fig_v_bar, color='r',
                 image_size=image_size, both=False)
```

The above two examples correspond to the case of `both=True` and the case of `both=False`.

The `dataset.py` module contains functions for loading datasets given in the `Data/` directory or for being used as the built-in functions for data generation.

`tetrahedron` generates the tetrahedron dataset. This dataset gives you a 4×4 distance matrix.

```python
from Code.dataset import tetrahedron

tetra_dist = tetrahedron()
```

`airline_dist` loads the airline dataset from `Data/dist_18_cities.csv`. This returns a lower triangle matrix which contains distance between cities and the name of the cities. The full distance matrix for this dataset can be achieved by adding this matrix and its transpose as follows:

```python
from Code.dataset import airline_dist

flying_dist, city = airline_dist()
flying_dist = flying_dist + flying_dist.T
```

`synthetic_spiral` generates 3-D spiral dataset. This is a 3×30 matrix, meaning that there are 30 points in 3-D space. This function can be used as follows:

```python
from Code.dataset import synthetic_spiral

X_spiral = synthetic_spiral()
```

`bars` loads the bar image dataset from `Data/bars.npz` and returns bar images in row vector and its center coordinates, and the size of images is 40×40. The bar dataset can be loaded and used as follows:

```python
from Code.dataset import bars

data_bar, centers = bars()
data_bar = data_bar.T
centers = centers.T
image_size = [40,40]
```

`face_tenenbaum` loads the face dataset from `Data/face_tenenbaum.npz`. The size of each image in this dataset is 64×64. The images are returned as column vectors, so this dataset can be used as follows:

```python
from Code.dataset import face_tenenbaum

data_face = face_tenenbaum()
image_size = [64,64]
```

1 Multidimensional Scaling

Multidimensional scaling (MDS) is one of manifold learning techniques that learns to preserve the distance between observations in a high-dimensional source space into a low-dimensional target space. It has been one of the most commonly used techniques for visualisation.

Given a distance matrix for N points in d-dimensional source space, $\Delta = (\delta_{ij})_{N\times N}$, MDS is going to find out low-dimensional representations of N points in a p-dimensional target space ($p < d$), $Z_{p\times N} = \{\boldsymbol{z}_1, \boldsymbol{z}_2, \cdots, \boldsymbol{z}_N\}$, such that $d_{ij} \approx f(\delta_{ij})$, where d_{ij} is the distance between points i and j in the target space and $f(\cdot)$ is a parametric monotonic function, e.g., $f(\delta_{ij}) = \alpha + \beta\delta_{ij}$. To solve the MDS problem, there are several different algorithms, e.g., cMDS and stress-based MDS[1].

1.1 Classical MDS (cMDS)

For a centralised data matrix, X, its inner-product (Gram) matrix, $G_{N\times N} = X^T X$, is expressible by its distance matrix, $\Delta^2 = (\delta_{ij}^2)_{N\times N}$:

$$G = X^T X = -\frac{1}{2} H\Delta^2 H; H_{N\times N} = I_{N\times N} - \frac{1}{N}\boldsymbol{e}\boldsymbol{e}^T,$$

where $I_{N\times N}$ is the identity matrix and $\boldsymbol{e} = [11\cdots 1]^T$

In cMDS, p-dimensional optimal coordinates in the target space can be obtained by conducting eigen-decomposition on $G = X^T X$. Let $\lambda_i, \boldsymbol{v}_i$ denote the i-th largest eigenvalue and its corresponding eigenvector, and the optimal p-dimensional coordinates are as follows:

$$\boldsymbol{z}_i^* = \sqrt{\lambda_i}\boldsymbol{v}_i, \;\; i = 1, 2, \cdots, p.$$

Assignment 1 [6 marks] In your answer notebook, complete the function `cmds`, which implements cMDS algorithm described above. In your implementation, you can use the built-in functions in Python libraries, e.g., `sklearn.metrics.pairwise.euclidean_distances`, to calculate Euclidean distance and `numpy.linalg.eigh` to conduct eigen decomposition. Apply your completed `cmds` code on two datasets, tetrahedron contained in the notebook file as well as airline distance dataset, `dist_18_cities.csv`. In your answer notebook, (a) report the 3-largest eigenvalues and their corresponding eigenvectors achieved on the tetrahedron dataset and the 2-largest ones on the airline distance dataset, and (b) visualise 3-D and 2-D embedded points of tetrahedron and airline distance data in their target spaces, respectively.

1.2 Stress-based MDS

In stress-based MDS, optimal embedded coordinates are obtained by minimising loss functions (a.k.a. stress). There are three commonly-used stress loss functions, absolute error ($\mathcal{L}_{ee}(Z;\Delta)$), relative error ($\mathcal{L}_{ff}(Z;\Delta)$), and Sammon mapping (($\mathcal{L}_{ef}(Z;\Delta)$)). The loss functions are as follows:

[1] For details of the cMDS and stress-based algorithms, see the "Multi-dimensional Scaling (MDS)" lecture note.

$$\mathcal{L}_{ee}(Z;\Delta) = \frac{\sum_{i<j}(d_{ij} - f(\delta_{ij}))^2}{\sum_{i<j}\delta_{ij}^2}$$
$$\mathcal{L}_{ff}(Z;\Delta) = \sum_{i<j}(\frac{d_{ij} - f(\delta_{ij})}{\delta_{ij}})^2$$
$$\mathcal{L}_{ef}(Z;\Delta) = \frac{1}{\sum_{i<j}\delta_{ij}}\sum_{i<j}\frac{(d_{ij} - f(\delta_{ij}))^2}{\delta_{ij}}$$

and their gradient with respect to $\boldsymbol{z}_k \in Z$ are as follows:

$$\nabla\mathcal{L}_{ee}(\boldsymbol{z}_k) = \frac{2}{\sum_{i<j}\delta_{ij}^2}\sum_{j\neq k}(d_{kj} - f(\delta_{kj}))\frac{\boldsymbol{z}_k - \boldsymbol{z}_j}{d_{kj}}$$
$$\nabla\mathcal{L}_{ff}(\boldsymbol{z}_k) = 2\sum_{j\neq k}\frac{d_{kj} - f(\delta_{kj})}{\delta_{kj}^2}\frac{\boldsymbol{z}_k - \boldsymbol{z}_j}{d_{kj}}$$
$$\nabla\mathcal{L}_{ef}(\boldsymbol{z}_k) = \frac{2}{\sum_{i<j}\delta_{ij}}\sum_{j\neq k}\frac{d_{kj} - f(\delta_{kj})}{\delta_{kj}}\frac{\boldsymbol{z}_k - \boldsymbol{z}_j}{d_{kj}}$$

The optimal coordinates are obtained in an iterative way by update rules such as

$$\boldsymbol{z}_k^{(t+1)} \leftarrow \boldsymbol{z}_k^{(t)} - \eta\nabla\mathcal{L}(\boldsymbol{z}_k)|_{\boldsymbol{z}_k=\boldsymbol{z}_k^{(t)}}$$

Assignment 2 [2 marks] In your answer notebook, complete the functions `loss_abs`, `grad_abs`, `loss_rel` and `grad_rel` that calculate the stresses, $\mathcal{L}_{ee}(Z;\Delta)$), ($\mathcal{L}_{ff}(Z;\Delta)$) and their gradients, respectively. For demonstration, you have been provided with the stress and the gradient computation for Sammon mapping along with its application to the 3-D spiral data. Therefore, you can get hints from the provided Sammon mapping implementation to do this assignment. Again, You can use the built-in function in scikit-learn, `sklearn.metrics.pairwise.euclidean_distances`, to calculate distances between data points.

Assignment 3 [4 marks] Initial coordinates are set randomly in the current implementation of stress-based MDS, which often causes different results for different executions. To avoid this phenomenon, set the initial coordinates with the given values in the file, `init_y_spairal.npz`, for the 3-D spiral data, `init_val_airline_2d.npz` and `init_val_airline_3d.npz` for 2-D and 3-D projections of the airline distance dataset, `dist_18_cities.csv`, respectively. To use the given initial coordinates, you need to modify the initialisation part of the function, `stress_based_mds`. For learning, you need to use the provided gradient descent optimisation function, `gradient_descent` contained in the notebook by setting `optim='gd'` in `stress_based_mds()`. In your experiment, set the number of maximum iterations `max_iter=50,000`. To yield the correct results, you need to find out proper hyperparameters, `lr` (learning rate) and `tol` (tolerance - stop criterion). Apply the completed code in **Assignment 2** with the given initial coordinates and proper hyperparameters to the 3-D spiral data and the airline distance dataset, respectively, and show the 2-D embedding results of two datasets in your answer notebook.

2 Isometric Feature Mapping (ISOMAP)

In the ISOMAP[2], geodesic distance is used to overcome the weakness of MDS where the Euclidean distance metric does not respect the geometry of a non-linear manifold.

The ISOMAP algorithm consists of two steps:

1. Find out **approximate geodesic distances** between any points in high-dimensional space.
2. Apply **cMDS** to the geodesic distance matrix for low-dimensional embedding.

Assignment 4 [4 marks] To gain a geodesic distance matrix, local distances need to be calculated via one of two ways: i) K-nearest neighbourhood (KNN) or ii) ϵ-radius neighbourhood. In your answer notebook, implement the function `fixed_radius_distance(X, epsilon)` for choosing neighbours based on a fixed radius, where `X` and `epsilon` refer to the dataset and the fixed radius, respectively. By setting a proper ϵ value, run the provided ISOMAP code on Swiss roll dataset, `sklearn.datasets.make_swiss_roll`, which has been contained in your answer notebook. Display the 2-D embedded coordinates of the Swiss roll data in your answer notebook.

Assignment 5 [2 marks] Run the provided ISOMAP code on the face dataset, `face_tenenbaum.npz`. The routine used for displaying the connectivity between the neighbouring embedded points in low-dimensional space is provided for this dataset in the notebook file. In the figure showing the connectivity, randomly pick two pairs of start and end points to specify two shortest paths (e.g. from top to bottom, from left to right, etc). In your answer notebook, (a) display the images on the two specific paths you choose, and (b) report the indices of points you choose as well as all the points on the path and explain what you observe on those points in two paths in terms of manifold learning. To do this assignment, you need to use the provided functions/classes, `VIS_Shortest_path_2d` for selecting a pair of start and end points to generate a shortest path, and `ImageViewer` for displaying all images on the path.

3 Locally Linear Embedding (LLE)

The LLE[3] is yet another method to solve the same problem as ISOMAP encounters. For a given dataset, $X_{p\times N}$, a point $\boldsymbol{x}_i$ is reconstructed by linear combination of its neighbours $\boldsymbol{x}_j$ and optimal weights, W^*, can be learned by minimising the following loss function:

$$\mathcal{L}(W;X) = \sum_{i=1}^{N} \|\boldsymbol{x}_j - \sum_{j=1}^{N} W_{ij}\boldsymbol{x}_j\|^2$$
$$s.t. \sum_{j=1}^{N} W_{ij} = 1.$$

[2] For details of the ISOMAP, see the “Isometric Feature Mapping (ISOMAP)” lecture note.
[3] For details of the LLE, see the “Locally Linear Embedding (LLE)” lecture note.

Once the weights are obtained, the points ,$\boldsymbol{x}_i$, can be embedded into target space as p-dimensional coordinates ($p < d$), $Z_{p\times N} = \{\boldsymbol{z}_1^*, \boldsymbol{z}_2^*, \cdots, \boldsymbol{z}_N^*\}$, by minimising the loss function with respect to the embedded coordinates in Z:

$$\mathcal{L}(Z; W^*) = \sum_{i=1}^{N} \|\boldsymbol{z}_i - \sum_{j=1}^{N} \boldsymbol{z}_j\|^2$$

$$s.t. \sum_{i=1}^{N} \boldsymbol{z}_i = 0, \frac{1}{N-1} \sum_{i=1}^{N} \boldsymbol{z}_i \boldsymbol{z}_i^T = I_{N\times N}$$

The p-dimensional optimal embedded coordinates can be obtained by conducting eigen-analysis on $D = (I - W^*)^T(I - W^*)$ where W^* is the optimal weight matrix for data reconstruction in high dimensional space.

Assignment 6 [4 marks] To do this assignment, you are provided with our LLE implementation in Python, `lle(data, n_components=2, n_neighbors=None, epsilon=None, reg_func=None)` in the notebook file. You are asked to apply this function to the S-Curve dataset provided in the `sci-kit learn` library, `sklearn.datasets.make_s_curve`, which has been included in the notebook file. In your experiment, you need to use both KNN and the fixed ϵ radius methods to decide local neighbourhood of a point. Thus, you need to find proper values for `n_neighbors` and `epsilon`, respectively. To find out the optimal K and ϵ value, you need to use correlation coefficient between original manifold coordinates and embedded coordinates. To do so, you can use a built-in function, `scipy.stats.pearsonr`, in the `scipy` library to calculate the correlation coefficient of two vectors. For example, you may calculate two correlation coefficients: one between 1st axis of original manifold (angle) and 1st embedded coordinates and the other between 2nd axis of original manifold (height) and 2nd embedded coordinates. The best parameters should have the highest values in both. In your experiment, look into K taken in the range from 5 to 50 with the increment by 1 for KNN neighbourhood and ϵ in the range from 0.1 to 0.8 with the increment by 0.1. In your answer notebook, (a) plot the correlation coefficients when using KNN and ϵNN, respectively, in two diagrams, and (b) display 2-D embedded coordinates of the S-Curve data with the optimal K and ϵ value you found in a), respectively.

Assignment 7 [3 marks] Unlike linear models such as PCA, LLE can preserve the topological locations of data residing in a nonlinear manifold. In this assignment, you are asked to apply the provided LLE function with KNN neighbourhood, `lle(data, n_components=2, n_neighbors=None, epsilon=None, reg_func=None)`, to the bar dataset, `bars.npz`, where there are 1,000 images; 500 vertical and 500 horizontal bars. To display 2-D embedding coordinates, you can use the provided `VIS_Bars` class. In your answer book, (a) report the range you search for K and the optimal K value you find, and (b) display 2-D embedding coordinates of 1,000 bar images resulting from the optimal k value.

Requirement: Before starting working on this assessed coursework, you need to

1. download all the files required by this coursework from Blackboard as specified at the beginning of this document;

2. unzip the file then you should be see a Jupyter notebook file named `manifold.ipynb` and two sub-directories named `Data` and `Code`, respectively (you must keep this directory/sub-directory structure and their names unchanged when you work on this coursework);
3. rename `manifold.ipynb` in the directory as `yourfullname_manifold.ipynb`. For instance, if your name is "John Smith", your filename should be `john_smith_manifold.ipynb`. This file will be your answer notebook to be submitted for marking, so you must include everything required by the coursework in this Jupyter notebook.

Deliverable: Only your answer notebook, `yourfullname_manifold.ipynb`, which should include all your code, output, answers and your interpretation/justification. In this Jupyter notebook, all assignments have been separated with the clear delimiters. You must put your stuff regarding an assignment in those cells related to this assignment and, if necessary, create new cells within the delimiters of this assignment.

Your answer notebook, `yourfullname_manifold.ipynb`, must be submitted via the Blackboard.

Marking: Marking will be on the basis of (1) correctness of results and quality of comments on your code; (2) rigorous experimentation; (3) how informative and clear your results and answers are presented in your answer book; and (4) your knowledge exhibited, interpretation and justification.

Late Submission Policy: The default departmental late submission policy is applied to this coursework.

Deadline: 23:30, 12th January 2021 (Tuesday)