HPSC - Assignment 1

NGUYEN T. Hoang - SID: 15M54097 Spring 2016, W831 Mon-Thu. Period 1-2

Due date: 2016/05/10

Problem

Measure the convergence rate of FDM `step09.py`, given the error function:

$$\text{error} = \sqrt{\sum_{i,j=1}^{nx,ny} \frac{(p_{exact} - p_{approx})^2}{p_{exact}^2}}$$

The exact solution is available from BEM `step02.py`:

$$p_{exact} = \frac{x}{4} - 4 \sum_{n=odd}^{\infty} \frac{1}{(nx)^2 \sinh 2n\pi} \sinh n\pi x \cos n\pi y$$

The source code and jupyter notebook for this assignment can be found at:
`https://github.com/gear/HPSC/tree/master/hw`

Answer

Using the given code for FDM and exact solution in the lectures, I extract the boundary points from the solution of FDM and compare with the exact solution.

Extracting boundary points In this assignment, I rewrite `step09.py` of FDM as a function named `fdm` (file: `assign1.py`). The parameters of this function is:

- `nx`: x-axis resolution.
- `ny`: y-axis resolution.
- `nit`: number of time step.
- `draw`: (boolean) plot the data.

`fmd`'s output is a nx-by-ny numpy array with the final values of the solution of 2D Laplace's equation for the given number of time step `nit`. Function `get_border` is used to generate a 1-D array border from the 2D output. To match it with the exact result output of the function `exact`, the extracting order is given as follow:

Listing 1: Get border solution from 2D FDM

```python
# Extracted from assign1.py
...
def get_border(a):
  size = a.shape
  length = size[0]
  size = 2*(size[0] + size[1])
  ret = np.zeros(size)
  ret[0:length] = a[:,0]
  ret[length:2*length] = a[length-1,:]
  temp = a[:,length-1]
  ret[2*length:3*length] = temp[::-1]
  temp = a[0,:]
  ret[3*length:] = temp[::-1]
  return ret[::-1]
...
```

Calculating error The first problem I have with the given error function is the fact that it might contain zero division when p_{exact} is zero. Besides, the second problem is this error term isn't normalized and hence can be difficult to plot. In here, I propose two solutions for these problems:

- Zero division: Introduce a tolerance parameter of small value. If p_{exact} is smaller than this value, we use this tolerance value instead of real value of p_{exact}. The function named `error` in `assign1.py` implements this solution.
- Normalization: Each term inside the square root of the given error function is a square of the relative error of a data point. It is sufficient to divide each of these terms to the total number of data point in the sense that each error contributes a small portion to the overall error. In addition to the division, we can also introduces a different way to compute relative error. Instead of dividing to p_{exact}, we can divide the difference to $(p_{approx} + p_{exact})$. The function named `error_rel` implements the new relative error and the function named `error_rat` implements the normalization by dividing to number of data point.

The vector result of Listing ?? is shown as follow:

$$\begin{pmatrix} 0.437 & 0.437 & 0.437 & 0.437 & 0.464 & 0.136 & 0.044 & 0.044 \end{pmatrix}$$

Figure 1 shows the result in the graph. Eigenvector centrality of each vertex is shown by a blue decimal number next to it. As we can see, vertex number 5 has the highest eigenvector centrality means that vertex number 5 is the most *central* vertex. By looking at the graph, we can intuitively understand this fact.

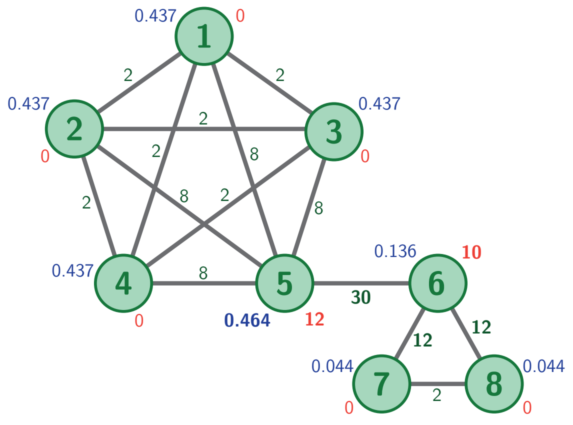


Figure 1: *The given network with eigenvector centrality (blue), and betweenness centrality of vertices (red) and edges (green)*

Betweenness centrality is another metric to measure how important a vertex is within the network. Different from other metric, betweenness centrality measure how important a vertex is in the information flow between other vertices. In [?], the author defines betweenness centrality x_i of vertex i as follow:

$$x_i = \sum_{st} n^i_{st}, \text{ or } x_i = \sum_{st} \frac{n^i_{st}}{g_{st}}$$

where n^i_{st} is the number of geodesic paths between vertex s and vertex t that go through i, and g_{st} is the total number of geodesic paths between s and t. Besides the normal betweenness centrality, in [?] the author also mentioned 2 other types of betweenness: *flow betweenness* and *random walk betweenness*. However, in this assignment, I will only compute the standard betweenness for the given network.

Listing 2: Betweenness centrality computation with SNAP.PY

```
# Extracted from UnweightedUndirectedGraph class - File:
   cn_a2_p1.py
...
import snap as sn
self._graph = sn.LoadEdgeList(sn.PUNGraph, edge_list, 0, 1, '␣')
...
# Compute betweenness centrality of vertices and edges and store
    to a hash table.
def BetweennessCentrality(self, isNode = true):
    # Create 2 hash maps: Int -> Float
    NodeScore = sn.TIntFltH()
    EdgeScore = sn.TIntFltH()
    sn.GetBetweennessCentr(self._graph, NodeScore, EdgeScore,
       1.0)
    if (isNode):
        return NodeScore
    else:
        return EdgeScore
```

The vector result of Listing 2 is shown as follow:

$$\text{NodeScore} = \begin{pmatrix} 0 & 0 & 0 & 0 & 12 & 10 & 0 & 0 \end{pmatrix}$$

$$\text{EdgeScore} = \begin{pmatrix} 2 & 2 & 2 & 8 & 2 & 2 & 8 & 2 & 8 & 8 & 30 & 12 & 12 & 2 \end{pmatrix}$$

Figure 1 shows the result in the graph. Betweenness centrality of each vertex is shown by a red integer next to it. As we can see, vertex number 5 has the highest betweenness centrality since it is the connection between two cliques.

Question 2: Construct Laplacian matrix and Modularity matrix for the given network. Laplacian matrix and Modularity matrix of the graph is computed with *numpy* as follow:

Listing 3: Laplacian matrix and Modularity matrix computation with Numpy

```python
# Extracted from UnweightedUndirectedGraph class - File:
   cn_a2_p1.py
...
import numpy as np
...
self._adj_matrix = edge_list_to_np(edge_list)
...
# Compute and return Laplacian matrix
def LaplacianMatrix(self):
    D = np.diag(np.sum(self._adj_matrix, 0))
    return D - self._adj_matrix
...
# Compute and return Modularity matrix
def ModularityMatrix(self):
    B = np.zeros(self._adj_matrix.shape)
    d = np.sum(self._adj_matrix, 0)
    m = sum(d)
    for i in range(B.shape[0]):
        for j in range(B.shape[1]):
            B[i,j] = A[i,j] - (d[i] * d[j]) / float(m)
    return B
```

The result of Listing 3 is:

$$\mathcal{L} = \begin{pmatrix} 4 & -1 & -1 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 & 0 & 0 \\ -1 & -1 & -4 & -1 & -1 & 0 & 0 & 0 \\ -1 & -1 & -1 & -4 & -1 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & 5 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 3 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

$$\mathcal{B} = \begin{pmatrix} -0.571 & 0.428 & 0.428 & 0.428 & 0.281 & -0.443 & -0.429 & -0.289 \\ 0.428 & -0.571 & 0.428 & 0.428 & 0.285 & -0.423 & -0.229 & -0.229 \\ 0.428 & 0.428 & -0.571 & 0.428 & 0.285 & -0.423 & -0.229 & -0.229 \\ 0.428 & 0.428 & 0.428 & -0.571 & 0.285 & -0.423 & -0.229 & -0.229 \\ 0.285 & 0.285 & 0.285 & 0.285 & -0.892 & 0.464 & -0.356 & -0.356 \\ 0.428 & -0.428 & -0.428 & -0.428 & 0.464 & -0.327 & 0.729 & 0.729 \\ 0.285 & -0.285 & -0.285 & -0.285 & -0.357 & 0.785 & -0.144 & 0.856 \\ 0.285 & -0.285 & -0.285 & -0.285 & -0.357 & 0.785 & 0.856 & -0.144 \end{pmatrix}$$

(a) Compute the eigenvalue of the Laplacian Matrix, deduce the second smallest eigenvalue, from there perform a spectral bisection on the graph into 2 equal parts.

Listing 4: Perform Spectral Bisection on the graph

```python
# Extracted from UnweightedUndirectedGraph class - File:
   cn_a2_p1.py
...
import numpy as np
...
self._adj_matrix = edge_list_to_np(edge_list)
...
# Compute spectral bisection
def SpectralBisection(self):
    L = self.LaplacianMatrix()
    # Get eigenvectors of \L
    u , v = np.linalg.eig(L)
    # Get the second smallest eigenvalue and its eigenvector
    i = np.argsort(u)[1]
    eigv = v[:,i]
    partition = np.ones(eigv.shape[0])
    index_sorted_eigv = np.argsort(eigv)
    for i in range(partition.shape[0] / 2, partition.shape[0]):
        partition[index_sorted_eigv[i]] = -1
    return partition
```

The vector result for this bisection is demonstrated as follow:

$$\text{partition} = \begin{pmatrix} 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

(b) Compute the eigenvalue of the Modularity Matrix, deduce the largest eigenvector, from there divide the network into 2 communities. In spectral modularity, we decide groups by possitiveness of each element in the eigenvector. Maximum modularity is archieved when there is a densense connection within a community and sparse between communites. The computation is performed as follow:

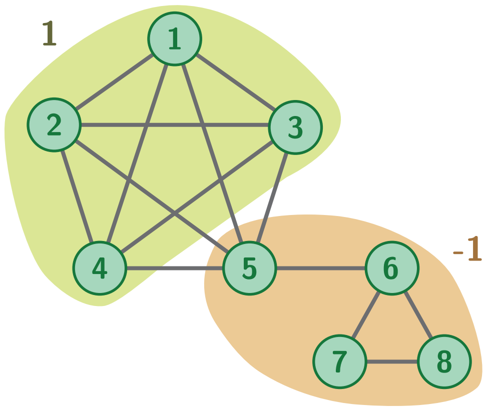


Figure 2: *Illustration of spectral bisection.*

Listing 5: Perform Spectral Modularity on the graph

```python
# Extracted from UnweightedUndirectedGraph class - File:
   cn_a2_p1.py
...
import numpy as np
...
self._adj_matrix = edge_list_to_np(edge_list)
...
# Compute spectral modularity
def SpectralBisection(self):
    B = self.LaplacianMatrix()
    # Get eigenvectors of \L
    u , v = np.linalg.eig(L)
    # Get the second smallest eigenvalue and its eigenvector
    i = np.argsort(u)[1]
    eigv = v[:,i]
    partition = np.ones(eigv.shape[0])
    index_sorted_eigv = np.argsort(eigv)
    for i in range(partition.shape[0] / 2, partition.shape[0]):
        partition[index_sorted_eigv[i]] = -1
    return partition
```

The vector result for this bisection is demonstrated as follow:

$$\text{partition} = \begin{pmatrix} -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 \end{pmatrix}$$

Comparing result between Figure 2 and Figure 3, we can see that the spectral modularity gives us a more *appropriate* result since it divided the given network into two *clique*. Both technique I used here minimizes the number of connection between communities. However, by forcing spectral bisection to divide network into 2 *equal* parts, we obtained an unoptimized result. In conclusion, it is clear that modularity method gives us a better clustering in both computational value and common sense.

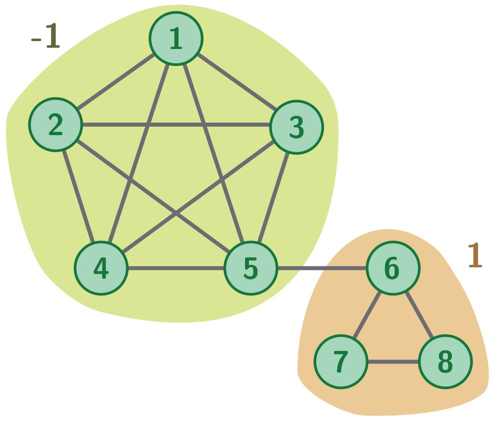


Figure 3: *Illustration of spectral modularity.*

Question 3: Explain quantitatively why "*your friends have more friends than you do*" in the configuration model. *Proof:* Suppose we are given a vector $k = \{k_0, k_1, ..., k_{n-1}\}$ contains the degree of all vertices in the configuration model with n vertices and m edges.

Without changing the generality of the statement, we assume that $k_i > 0 \ \forall \ k_i \ \in k$ and $n > 0$. In this context, the word "friend" of a vertex i implies the adjacency vertices of vertex i. Also, k_i is the number of friend that vertex i has. Denote F_i as the number of friends that some vertex i has, and FF_i as the number of friends that friends of vertex i has. Quantitatively, the average number of friend that an arbitary vertex i has is:

$$\mathbb{E}(F_i) = \frac{1}{n}\sum_{i=0}^{n-1} k_i = \frac{2m}{n}$$

Therefore, in this configuration model, the average friend that "you" have is $2m/n$. On the other hand, the probability that there is a connection between vertex i and vertex j (i and j are friends) is:

$$p(i \leftrightarrow j) = \frac{k_i k_j}{2m-1} \approx \frac{k_i k_j}{2m}$$

The average number of friends that a friend of vertex i has is:

$$\begin{aligned}\mathbb{E}(FF_i) = \frac{1}{k_i}\sum_{j=0}^{n-1} p(i \leftrightarrow j) \times k_j \ &= \ \frac{1}{k_i} \times k_i \sum_{j=0}^{n-1} \frac{k_j}{2m} \times k_j \\ = \frac{1}{2m}\sum_{j=0}^{n-1} k_j^2\end{aligned}$$

Consider the different $\mathcal{D}$ between $\mathbb{E}(F_i)$ and $\mathbb{E}(FF_i)$. Note that $2m = \sum_j k_j$:

$$\mathcal{D} = \mathbb{E}(FF_i) - \mathbb{E}(F_i) \ = \ \frac{\sum_{j=0}^{n-1} k_j^2}{2m} - \frac{2m}{n} \ = \ \frac{n\sum_{j=0}^{n-1} k_j^2 - 2\left(\sum_{j=0}^{n-1} k_j\right)^2}{2mn}$$

Expand the square term $2(...)^2$, we have the following result:

$$\mathcal{D} = \frac{(n-1)\sum_{j=0}^{n-1} k_j^2 - 2\sum_{i<j} k_i k_j}{2mn}$$

Using the Cauchy-Schwarz inequality, we have:

$$k_i^2 + k_j^2 \geq 2k_i k_j$$

Therefore, using Cauchy-Schwarz for $n(n-1)/2$ pairs of k_i, k_j we have:

$$(n-1)\sum_{j=0}^{n-1} k_j^2 \geq 2\sum_{i<j} k_i k_j$$

Hence,

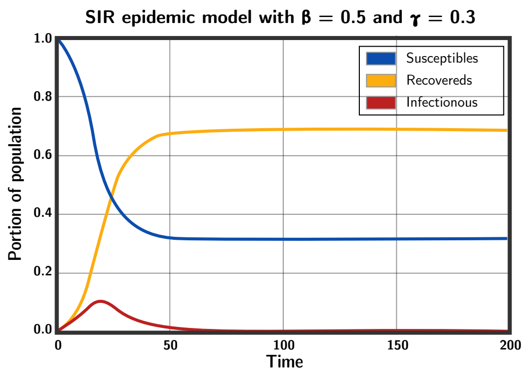
$$\mathcal{D} = \frac{(n-1)\sum_{j=0}^{n-1} k_j^2 - 2\sum_{i<j} k_i k_j}{2mn} \geq 0$$

In conclusion, the different between average number of a friend's friend and your friend is $\mathcal{D} \geq 0$, therefore the statement "*(on average) your friends have more friends than you do*" holds true. The equal sign happens when everyone has exactly 1 friend. `QED`.

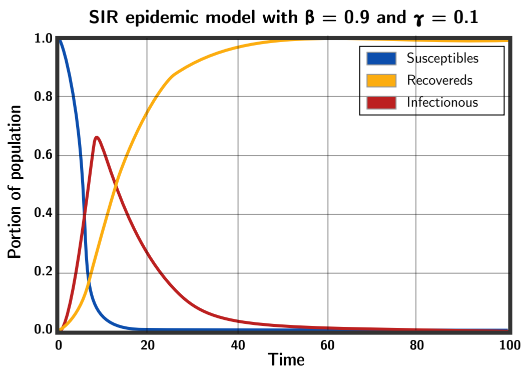
Question 4: Example of parameter β and γ of SIR model. In the SIR epidemic model, β is the infection rate and γ is recover (or death) rate. The model is represented as a system of diffirential equation:

$$\frac{ds}{dt} = -\beta sx; \quad \frac{dx}{dt} = \beta sx - \gamma x; \quad \frac{dr}{dt} = \gamma x$$

where (s, x, r) are fraction of population that are *subceptible*, *recovered*, and *infectionous* respectively. Since the solution of these diffirential equation is not analytical, numerical method is applied to analize this SIR model. In real-life application, the parameter β and γ is determined empirically by trying to fit known data with some trial value of β and γ. The result for each value (s, x, r) in this exercise is stored in an array, which is indexed my time step. According to [**?**], the epidemic happens when $\beta > \gamma$, which means the disease spreads faster than human recovery. I plotted the time graph for 4 cases:

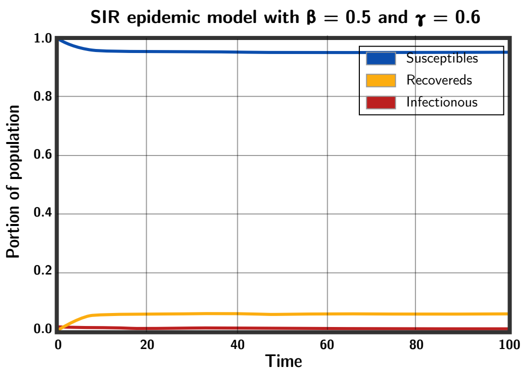


(a) SIR model, case $\beta = 0.5$ and $\gamma = 0.3$

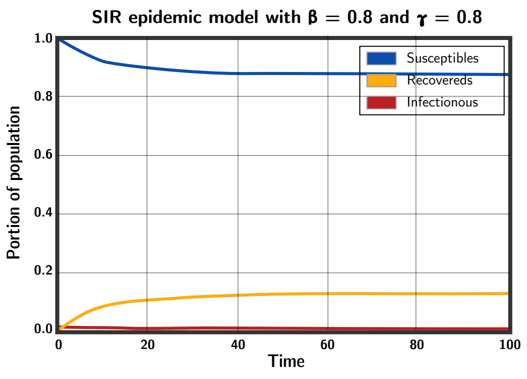


(b) SIR model, case $\beta = 0.9$ and $\gamma = 0.1$

Figure 4: SIR model in case of epidemic happens.



(a) SIR model, case $\beta = 0.5$ and $\gamma = 0.6$



(b) SIR model, case $\beta = 0.8$ and $\gamma = 0.8$

Figure 5: SIR model in case of no epidemic happens.