Math 260, Fall 2020
Last updated: 10/9/20

HOMEWORK 7

DUE FRI. OCT. 16

Important: You are required to have your code in a github repository. I suggest maintaining it there as well (as opposed to just copy-pasting your submission into it at the end) as you edit and revise.

To submit via Gradescope, you first need to authorize Gradescope to access your GitHub (this should only be necessary once); then you can submit from your repository.

Note that (I think) you can have just one repository for this course (like my course github), so you don't need to have one for each homework.

Exercises (not submitted). Suggested for practice/context; not required.

E1 (an important note). Let P be a transition matrix in a 'closed' Markov chain (so the probabilities in each row sum to 1). Show that $\lambda = 1$ is always an eigenvalue of P and find an eigenvector. *Hint: what vector v makes Pv involve the sum of the elements in each row?.*

Remark: The eigenvector for $\lambda = 1$ of P is not the one we want for the stationary distribution! You run into this in practice if you forget to transpose P in calculation.

E2 (sparse matrices). Create an $n \times n$ sparse matrix with zeros everywhere except 1's on the main diagonal and the 'backwards' diagonal from the upper right to lower left (i.e. an X shape). Check that it is correct by printing the matrix (converted to an array).

Computational problems. *Note:* make use of the example code in `power.py` (power method) and `stationary_dist.py` (Markov chains). In some cases they only need to be modified slightly. It will hopefully save time.

Q1 (power method, convergence?). Here is a test matrix and its eigenvalue/vectors:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 6 & -11 & 6 \end{bmatrix}$$

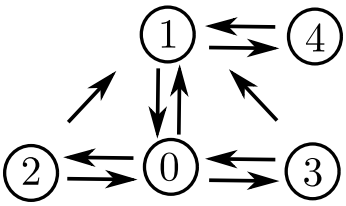
$$\lambda_1 = 3,\ \mathbf{v}_1 = \begin{bmatrix} 1 \\ 3 \\ 9 \end{bmatrix}, \quad \lambda_2 = 2,\ \mathbf{v}_2 = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}, \quad \lambda_3 = 1,\ \mathbf{v}_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix},$$

(note: recall that the eigenvectors can be scaled by a constant).

a) Modify the power method code so that it calculates the error in the eigenvector at each step (given the solution eigenvector as an input - you don't have to worry about efficiency).

b) Now make a convergence plot of the error vs. step of the appropriate type, and figure out how the error behaves. How does it relate to the eigenvalues?

Q2 (PageRank, small example). Consider the following network of sites and links:



a) Modify the stationary distribution code (`stationary` in the example code) to use the size of $\mathbf{x}_{k+1} - \mathbf{x}_k$ as a stopping condition (less than some tolerance), plus a **maximum** number of steps. Use it in (b), and pick reasonable tolerances, max number of steps etc. (so you don't have to input the number of steps manually).

b) Calculate the ranking of theses sites via PageRank and $\alpha = 0.95$. (Note: you certainly don't have to use the method given in the example to build the matrix M!!). How many iterations are required to get a reasonable solution?

c) Now try decreasing α to see how the rank changes. How small must α be to change the highest ranked page (if such a value exists)? (Put in a comment)

Note: the larger, sparse version of this will be on the next homework.

Q3 (File I/O, to be used next week). You may want to review the File I/O lecture and example from much earlier in the course.

Suppose you have a text file defining graphs in the following way:

- An **edge** from i to node j is listed as:

```
e i j
```

- A vertex indexed i with a certain name is listed as:

```
n i name
```

Write a function `read_graph_data(fname)` that takes in the name of a text file and returns the adjacency list for this graph *and* a list of the names for each vertex.

You can use a python list or a **dictionary** for the adjacency list and names (which is easier depends on exactly how you write the code).

For instance, given the file `graph.txt` with text

```
n 0 A
n 1 B
n 2 C
n 3 D
e 0 1
e 0 3
e 1 2
e 2 1
e 2 0
```

the function `read_graph` return the following:

```
# dictionary version
adj = {{0: [1, 3]}, {1: [2]}, {2: [0, 1]}, {3: []}}
names = {{0: 'A'},{1: 'B'},{2: 'C'},{3: 'D'}}

# list version
adj = {[1, 3], [2], [0, 1], []}
names = {'A', 'B', 'C', 'D'}
```

Note: You may assume that the data file is formatted correctly, i.e. that whoever made the file was responsible and followed the formatting requirements.