Math 151B HW 6 Project Report

Justin JR Wang

5/27/14

User Guide:

eigfinder.m :

To find the dominant eigenvector and eigenvalue of a matrix, we define these MATLAB functions. eigfinder.m finds the dominant eigenvector and teh associated eigenvalue of a given matrix A in R^nxn.

The calling syntax is:

```
function [ lambda, v, iter ] = eigfinder( A, x0, paras )
```

Inputs:
A is the input matrix A. In the PageRank algorithm, the modified adjacency matrix M should be plugged into A.
x0 is the initial guess for the dominant eigenvector

paras is a structure array containing the following fields:
- **.tol** is the tolerance
- **.maxiter** is the maximal number of iterations allowed
- **.q** is the parameter q used in the inverse power method and is empty in other methods
- **.option** specifies the method to be used, eg., strings ‘power1’, ‘power2’, and ‘invpower’

Outputs:
lambda is the estimated dominant eigenvalue
v is the estimated dominant eigenvector
iter is the number of iterations that the algorithm actually used

power1.m : This function implements the power method with l-infinity-norm scaling (refer to algorithm 9.1)

Note that in algorithm 9.1, the Rayleigh quotient is replaced by the pth component of the vector A**x**^(k-1) where p is the smallest integer in [1,n] such that abs(x_p) = norm(**x**, Inf).
The drawback of this version is lim as k tends to Inf of lambda1^k=Inf if lambda1>1.

The calling syntax is:

```
function [ lambda, v, iter ] = power1( A, x0, tol, maxiter)
```

Inputs:
A is the input matrix A. In the PageRank algorithm, the modified adjacency matrix M should be plugged into A.
x0 is the initial guess for the dominant eigenvector
tol is the tolerance
maxiter is the maximal number of iterations allowed

Outputs:
lambda is the estimated dominant eigenvalue
v is the estimated dominant eigenvector
iter is the number of iterations that the algorithm actually used

power2.m :

This function implements the power method with l-2-norm scaling (refer to algorithm 9.2 or lecture notes)

The calling syntax is:

function [lambda, v, iter] = power2(A, x0, tol, maxiter)

Inputs:
A is the input matrix A. In the PageRank algorithm, the modified adjacency matrix M should be plugged into A.
x0 is the initial guess for the dominant eigenvector
tol is the tolerance
maxiter is the maximal number of iterations allowed

Outputs:
lambda is the estimated dominant eigenvalue
v is the estimated dominant eigenvector
iter is the number of iterations that the algorithm actually used

invpower.m :

This function implements the inverse power method with l-2-norm scaling (refer to lecture notes)

If the matrix A has eigenvalues lambda 1 to lambda n, and q is not an eigenvalue of A, then the matrix (A-q*I_n)^-1 has eigenvalues (lambda_i-q)^-1 with the same eigenvectors as A. In other words, if mu is the dominant eigenvalue of (A-q*I_n)^-1, then mu^-1+q is the eigenvalue of A closest to q. Notice that the power1 and power2 provide the approximate dominant eigenvalue of A.

The calling syntax is:

```
        function [ lambda, v, iter ] = invpower( A, x0, q, tol, maxiter)
```

Inputs:
A is the input matrix A. In the PageRank algorithm, the modified adjacency matrix M should be plugged into A.
x0 is the initial guess for the dominant eigenvector
q is a parameter such that lambda is the nearest eigenvalue to q
tol is the tolerance
maxiter is the maximal number of iterations allowed

Outputs:
lambda is the estimated dominant eigenvalue
v is the estimated dominant eigenvector
iter is the number of iterations that the algorithm actually used

Implementation:

The goal is to implement Google search engine's PageRank algorithm with the Power method and its variants, and apply them to a simplified graph simulating the World Wide Web. The ranking is based on finding the dominant eigenvector of a matrix which describes the connection of all web pages in one network.

From graph theory, we know that a directed graph is a set of vertices connected by edges where each edge has a direction. For example, we can compare the internet to a directed graph, where each web page represents a vertex, and a link represents an edge. The adjacency matrix of a directed graph is defined as an nxn matrix B whose entries are:

$$b_{ij} = \begin{cases} 1, & \text{if there is a link from the page } i \text{ to the page } j; \\ 0, & \text{otherwise.} \end{cases}$$

From B, we can construct a modified adjacency matrix M=(m_ij) which tells us the probability of visiting the webpage j from the web page i. Suppose that when visiting a web page, one surfer clicks on link in this web page with probability p and jumps to a completely random web page in the network with probability 1-p. The probability p is also known as the damping factor in the PageRank theory and is usually set around 0.85. Furthermore if there are n pages in the graph, we define the modified adjacency matrix M whose entries are:

$$m_{ij} = p\left(\frac{b_{ij}}{\sum_{k=1}^{n} b_{ik}}\right) + \frac{1-p}{n}$$

In fact, M is a stochastic matrix satisfying that each row sums to 1, and thereby 1 is the dominant eigenvalue of M with algebraic multiplicity 1 by the Perron's Theorem. The Markov theory states that if **v** with the sum of the absolute value of its entries equal to 1 is a left dominant eigenvector of M associated with the eigenvalue 1, i.e., **v**^T *M=**v**^T, then vi is the probability that one visits the page i at the stationary state independent of the starting page, which determines the rank of the ith web page.

I used the algorithms for the power method and the inverse power method presented in class. The power1 method uses the infinity norm, while the power2 method uses the l-2 norm, and the two are otherwise identical. We still use the infinity norm to calculate the tolerance. In these algorithms, and the inverse power method algorithm, we use the Rayleigh quotient, and the method differs than that which is presented in the textbook.

From the professor's presentation of the algorithm, it was not immediately obvious how to code the difference between xk and xk+1 to compute the tolerance telling when the iterations should stop, but you simply introduce a new variable x1 to store the value of the current x, normalized, and the other x will be the update you wish to compare it to. It'll be xk and xk+1, essentially.

Also, initially I was not sure how to know the actual values of the eigenvalues and eigenvectors that we need to compare to, in order to compute the error of my approximations, but then we just use MATLAB's eig function. That did cross my mind, but that method should be an approximation too, but I guess it's as good as it gets (and it's good enough). We are using these methods to find the dominant eigenvalue and dominant eigenvector of a matrix.

Solutions:

Use the three methods with at least two initial vectors randomly generated by randn(n,1) to solve the following problems. To ensure two generated vectors are different, you may consider setting the random number generator rng. Compare their performance in terms of the number of iterations and the error (use l-inf-norm) between the actual eigenvalue/eigenvectors and their respective estimations.

If the initial random vectors are changed, the iter and error will be changed accordingly in the results.
a) Find the rank of each webpage in the network shown in Figure 1 with 15 webpages. (Hint: Based on the graph, construct the corresponding adjacency matrix B and the modified adjacency matrix M, then execute eigfinder.)

When I computed the error using the actual dominant eigenvector, found using MATLAB's eig function, I made sure to make sure that the eigenvectors are compared fairly by making them have the same scale. (In other words, avoid the issue that eigenvectors can be scalar multiples of themselves by normalizing the eigenvectors found by my algorithms. I'm pretty sure MATLAB's eig function normalizes the eigenvectors.)

Also I make sure that the signs of the first entries of every eigenvector match (so they are all positive). The actual eigenvector of part B has a negative entry, so I just multiply each component of the eigenvector by -1.

The rank of each webpage in the network shown are the entries of the dominant eigenvector of the adjacency matrix.

I use error tolerance 10^-6, and max 100 iterations in all cases.

The eigenvalues are:

```
Eigenvalues =

  -0.2677 - 0.4637i
   1.0000 + 0.0000i
   1.0000 + 0.0000i
   0.6189 + 0.0000i
   1.0000 + 0.0000i
   1.0000 + 0.0000i
   0.6189 + 0.0000i
```

The first entry is the actual value of the eigenvector computed using eig function from matlab. We can see that powe1 and power2 resulted in the same eigenvalue.

```
Eigval_errors_a =

    1.3499
    1.3499
    1.0006
    1.3499
    1.3499
    1.0006
```

Here are the errors of the eigenvectors, followed by explanation:

```
error1 =

    0.6932
error2 =
        0.6932
error3 =

        1.2740
error4 =

        0.6932
error5 =

        0.6932

error6 =
        0.7984
```

We use two different randomly generated 15x1 vectors as the initial guess. So, 1 corresponds to the power1 method, 2 corresponds to the power2 method, and 3 corresponds to the power3 method. 4, 5, and 6 are the respective methods done using a different, randomly generated initial guess.

To compare their performance in terms of error, it looks like the power1 and power2 methods result in similar error, using both randomly generated initial guesses, while the error for the inverse power method is pretty substantially worse in the first initial guess, and noticeably worse in the second, compared to the first two methods.

Now let's see the number of iterations:

```
I =


        64
        46
        9
        65
        47
        13
```

It goes method: power1, power2, invpower, power1, power2, invpower, as before. So power1 takes the most number of iterations, at about 65, while power2 uses a little less iterations, at around 47, while the inverse power uses the least number of iterations by a significant amount - the number of iterations, averaging these two randomly generated initial guesses, comes to 11.

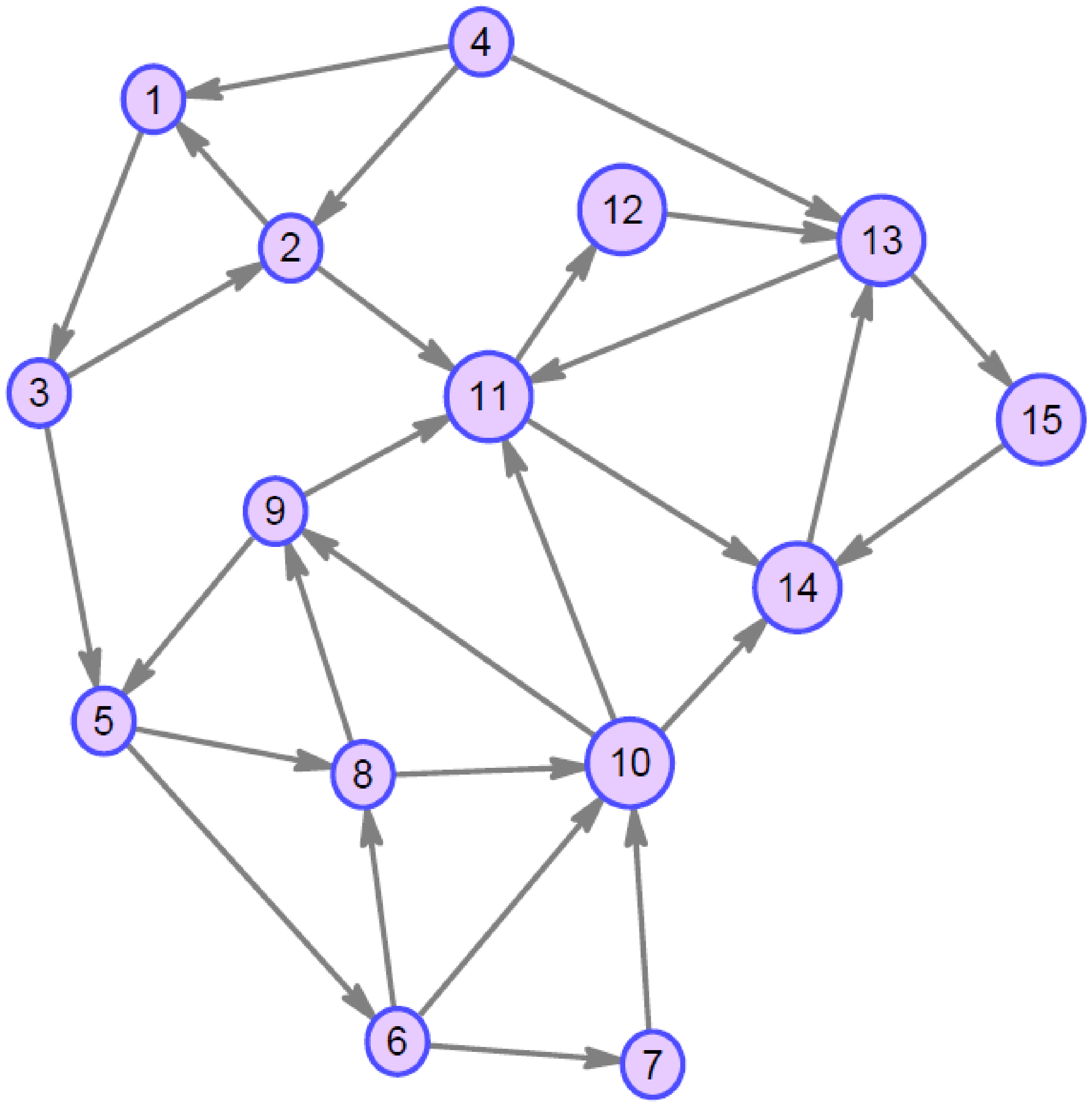

Figure 1: Simulated network with 15 nodes.

b) Find the dominant eigenvalue and the dominant eigenvector of the matrix:

$$A = \begin{bmatrix} 2.395798 & 0.234169 & 0.127074 & 0.146184 & 0.183889 \\ 0.113724 & 5.103374 & 0.243386 & 0.030779 & 0.241161 \\ 0.183743 & 0.199444 & 7.642053 & 0.199313 & 0.145211 \\ 0.085881 & 0.144653 & 0.104811 & 9.013056 & 0.024832 \\ 0.053909 & 0.180566 & 0.126246 & 0.249744 & 3.774798 \end{bmatrix}$$

This is pretty much part a), just using a different matrix.

Oh, since the actual eigenvector given by the eig function built into matlab returned a dominant eigenvector with the leading entry negative valued, to fairly calculate the error with my approximated eigenvalues, I multiply each entry of the actual eigenvalue by -1, since the approximated eigenvalues have a positive leading entry.

Here are the eigenvalues:

```
Eigenvalues =

    9.0360
    9.0360
    9.0360
    7.6534
    9.0360
    9.0360
    7.6534
```

These are pretty good, for my power1 and power2 methods.

```
Eigval_errors_b =

    0.0000
    0.0000
    1.3826
    0.0000
    0.0000
    1.3826
```

There was no error in eigenvalue to calculate for power1 and power2 methods, but the inverse power method was a little off though. Perhaps for this smaller 5x5 matrix, finding the dominant eigenvalue using power methods and eig function gave a similar result.

Here are the errors of the approximated eigenvectors:

```
error1b =
  1.8373e-08
error2b =
  1.8373e-08
error3b =
          1.0753
error4b =
  1.3961e-09
error5b =
  1.3961e-09
error6b =
          1.0753
```

So we see that the errors are super tiny for the power1 and power2 methods, and for the inverse power method, the error is worse in this case.

```
Iterations =

   100
   100
   13
   100
   100
   42
```

So to achieve the good accuracy, the power1 and power2 method(s), used the maximal number of iterations, while the inverse power method used much fewer, as little as 13 in one case.