Math 284 Final Project

Spring 2016 Name: **Surtej Sarin**; April 9th, 2016

This worksheet contains a list of MATLAB commands that you may want/need to use for the linear algebra exam PROJECT II. Note: I have included the commands, but not their meanings! Many will do exactly what you (hopefully!) expect, but some others may not be so obvious – enter each of these, and pay careful attention to the outputs to and see exactly what the commands are doing. You may want to test your conjectures by making up different matrices and applying the commands to those, and/or doing some computations by hand or with your calculator.

**Due date and time: MAY 11, 2016 at11:59 PM on Blackboard.**

**PROJECT II consists of parts A, B and C.**

**Please Note: Answers are written in blue**

**PART A**

Start by entering the matrix A = [1 3 -6; -1 5 -6; -2 2 -2] and the vectors B=[6;-2;-6] and C=[1;2;1]

Then enter each of the following commands, and determine what the effect of the command is.

[A, B] Effect: [1 3 -6 6; -1 5 -6 -2; -2 2 -2 -6]

The Augmented Matrix

A(1,:) *and* A(:,2) Effects: [1 3 -6]

First Row of Matrix A

*and* [3; 5; 2]

Second Row of Matrix A

rref[A, B] Effect: [1 0 0 3; 0 1 0 -1; 0 0 1 -1]

Row reduced echelon form of the Augmented Matrix

A \* A Effect: [10 6 -12; 6 10 -12; 0 0 4]

Multiplies Matrix A by A

dot(B,C) Effect: -4

The scalar dot product of B and C

transpose(A) Effect: [1 -1 -2; 3 5 2; -6 -6 -2]

The transpose of A

eig(A) Effect: [-2.0000; 4.0000; 2.0000]

The column vector containing the eigenvalues of Matrix A

[V , D] = eig(A) Effect: [-2.0000 0 0; 0 4.0000 0; 0 0 2.0000]

The columns of V represent eigenvectors of A.

The diagonal matrix D contains eigenvalues.

If the resulting V has the same size as A, the matrix A has a full set of linearly independent eigenvectors that satisfy A\*V = V\*D.

sortrows(A) Effect: [-2 2 -2; -1 5 -6; 1 3 -6]

Sorts the rows of A in ascending order of the first column

[E,index]=sortrows(A) Effect: E = [-2 2 -2; -1 5 -6; 1 3 -6]

index = [3; 2 1]

Returns an index vector using any of the previous syntaxes. The index vector satisfies E = A(index,:)

Some other useful variations on the sortrows command:

-sortrows(-A) Effect: [1 3 -6; -1 5 -6; -2 2 -2]

Sorts the rows in descending order

sortrows(A,2) Effect: [-2 2 -2; 1 3 -6; -1 5 -6]

Switch rows 2 and 3

A .\* A *(****note the period*** *preceding the multiplication symbol)*

Effect: [1 9 36; 1 25 36; 4 4 4]

Multiplying each element by itself

*What is the significance, in terms of the mathematics we have been doing in class, of the values of* *the elements in* sum(A .\* A) ?

Effect: [6 38 76]

Sum of each column in the matrix (A .\* A)

The sum method returns a row vector containing the sum of each column.

For example sum of first indexs of each column is 1 + 1 + 4 = 6

R = rand(4,6) Effect: [0.8147 0.6324 0.9575 0.9572 0.4218 0.6557;

0.9058 0.0975 0.9649 0.4854 0.9157 0.0357;

0.1270 0.2785 0.1576 0.8003 0.7922 0.8491;

0.9134 0.5469 0.9706 0.1419 0.9595 0.9340]

Generate a 4-by-6 matrix of uniformly distributed random numbers between 0 and 1.

Try to predict exactly what the following routine does. Then enter it (either line by line, or as an M-file) and test your prediction:

*Answer*: This routine creates a 4x6 matrix populated with random decimal numbers from 0 to 1. Each element is squared, then each column is added together, and then the square root of the sum of the columns. Then it sums the values in the matrix returning the row vector containing the sum of each column. Then it square roots the sum and stores it in variable S. then it stores the first column and divides it by S(i) to get the normalized columns. My prediction was correct.

R=rand(4,6)

S=sqrt(sum(R.\*R))

for i=1:6

R(:,i)=R(:,1);

R(:,i)=R(:,i)/S(i);

end

fprintf('normalized columns are')

for i=1:6

R(:,i)

end

Write an M-file which does the following:

1. Creates a matrix with 10 columns, each of which is a random vector in
2. For each column vector, determines the cosine of the angle between it and the vector (1, 2, 3)
3. Ranks the vectors from 1 (the largest cosine) to 10 (the smallest cosine)

R=rand(3,10)

v = [1; 2; 3]

A = zeros(1,10)

for i=1:10

cosAngle = dot(R(:,i),v)/(norm(R(:,i))\*norm(v));

Degrees = acosd(cosAngle)

Degrees

A(:,i)= Degrees

end

sort(A)

**MATH 284 – MATLAB, part B**

***(Note: this first part is based on the material in your book in section 7.1, Theorem 3d, page 399)*** : if A is symmetric, then A can be factored as A=PDPT , and this is equivalent to writing

A =  where the λ's are the eigenvalues and the u's are the corresponding unit eigenvectors (i.e., the columns of P).

We can therefore store A in a computer in several alternative ways, including as a list of n scalars (the eigenvalues) along with a list of n vectors (the corresponding unit eigenvectors)

Furthermore, if some λi=0, then λiuiuiT = 0, and we can take advantage of this to store A more efficiently.

Problems 1-5 help illustrate:

1. Find the orthogonal diagonalization of 

you may do the calculations by hand if you really want, but I suggest you use your calculator or MATLAB using the command [V , D] = eig(A)

A = [11 11 9 9;11 11 9 9;9 9 11 11;9 9 11 11];

V =

[0.000000000000000 0.707106781186548 -0.500000000000000 0.500000000000000

0.000000000000000 -0.707106781186547 -0.500000000000000 0.500000000000000

0.707106781186548 -0.000000000000000 0.500000000000000 0.500000000000000

-0.707106781186548 0 0.500000000000000 0.500000000000000]

D =

[ -0.000000000000009 0 0 0

0 -0.000000000000000 0 0

0 0 3.999999999999999 0

0 0 0 40.000000000000007]

2. Express A in the form . Arrange the λ's in decreasing order.

>> D(4,4)\*V(:,4)\*(V(:,4)')+D(3,3)\*V(:,3)\*(V(:,3)')+D(2,2)\*V(:,2)\*(V(:,2)')+D(1,1)\*V(:,1)\*(V(:,1)')

D(1,1)= -0.000000000000009

D(2,2)= -0.000000000000000

D(3,3)= 3.999999999999999

D(4,4)= 40.000000000000007

V(:,1)= [0.000000000000000;0.000000000000000;0.707106781186548;-0.707106781186548]

V(:,2)= [0.707106781186548;-0.707106781186547;-0.000000000000000;0]

V(:,3)= [-0.500000000000000;-0.500000000000000;0.500000000000000;0.500000000000000]

V(:,4)= [0.500000000000000;0.500000000000000;0.500000000000000;0.500000000000000]

ans =

[10.999999999999998 10.999999999999998 8.999999999999998 8.999999999999996

10.999999999999998 10.999999999999998 8.999999999999998 8.999999999999996

8.999999999999998 8.999999999999998 10.999999999999995 11.000000000000000

8.999999999999996 8.999999999999996 11.000000000000000 10.999999999999991]

3. Now re-write the sum from (2), but leave off the irrelevant 0 term(s).

>> D(4,4)\*V(:,4)\*(V(:,4)')+D(3,3)\*V(:,3)\*(V(:,3)')

ans =

[10.999999999999998 10.999999999999998 8.999999999999998 8.999999999999996

10.999999999999998 10.999999999999998 8.999999999999998 8.999999999999996

8.999999999999998 8.999999999999998 10.999999999999998 10.999999999999996

8.999999999999996 8.999999999999996 10.999999999999996 10.999999999999995]

4. Let S be the matrix whose columns are the orthonormal vectors that were used in (3), and let D be the diagonal matrix whose entries are the corresponding (nonzero) eigenvalues of A. Evaluate SDST .

>> S = [-.5 .5;-.5 .5; .5 .5;.5 .5]

S =

-0.500000000000000 0.500000000000000

-0.500000000000000 0.500000000000000

0.500000000000000 0.500000000000000

0.500000000000000 0.500000000000000

>> D = [4 0; 0 40]

D =

4 0

1. 40

>> transpose(S)

ans =

-0.500000000000000 -0.500000000000000 0.500000000000000 0.500000000000000

0.500000000000000 0.500000000000000 0.500000000000000 0.500000000000000

>> S\*D\*transpose(S)

ans =

11 11 9 9

11 11 9 9

9 9 11 11

9 9 11 11

Notice that if A is a very large matrix, and it has the eigenvalue 0 with a large multiplicity, this factorization can provide a way of storing A that requires much less memory than A itself would require.

Some follow up questions to this technique for data compression:

* What if A does not have the eigenvalue 0 or, if it does, if it does not have a large multiplicity?

Then this won't be an effective way of compressing data. However, if A does have some eigenvalues that are much smaller than the others, then we ignore the corresponding terms in the expansion and thereby compress data while only introducing small errors – which is often a worthwhile exchange.

* What if A is not symmetric – or not even square?

The short answer is that in these cases, this technique does not work.

HOWEVER there is an analogous factorization that works for *any* matrix: This is described below:

Any matrix A can be written in the form A = PDQT , where P and Q are orthogonal matrices, and D is zero off the main diagonal. The main difference between this and an orthogonal diagonalization is that if A is not symmetric, then P and Q are no longer the same as each other.

Note: If A is m x n, then P is m x m, and Q is n x n. D is then necessarily m x n, i.e. the same shape as A.

The entries on the diagonal of D are called the singular values of A, and this factorization is called the ***Singular Value Decomposition* of A**.

If we had more time in the semester we’d investigate how to find the singular value decomposition of a matrix and the many reasons why you’d want to do so. Instead, we’ll let MATLAB do the computations for us, and we’ll concentrate on one important application.

The command you’ll use is [P, D, Q] = svd(A) , where A is the matrix you want to factor, and the 3 variables in the brackets (where I used P, D, and Q but you are obviously free to pick other names) represent the 3 matrices you get in the factorization.

Find the singular value decomposition of the matrix A from the other side of this worksheet.

A = [11 11 9 9;11 11 9 9;9 9 11 11;9 9 11 11];

[P,D,Q] = svd(A)

D=

39.999999999999993 0 0 0

0 4.000000000000004 0 0

0 0 0.000000000000002 0

0 0 0 0.000000000000000

As explained in the assigned reading, a truncated SVD is found by replacing D with a square matrix, effectively trimming off any “excess” rows (or columns). P or Q is then trimmed as appropriate to make the matrix products still possible. This process does not result in the loss of any information!

Let B = [43 45 36 36; 45 43 36 36; 36 36 44 44; 36 36 44 44]. Then let B=[B, 2\*B, -2\*B]

What does this do?

B = [B, 2\*B, -2\*B]

B=

43 45 36 36 86 90 72 72 -86 -90 -72 -72

45 43 36 36 90 86 72 72 -90 -86 -72 -72

36 36 44 44 72 72 88 88 -72 -72 -88 -88

36 36 44 44 72 72 88 88 -72 -72 -88 -88

Matrix B is combine with two times itself and also negative 2 times itself.

Type [P, D, Q] = svd(B) to see the SVD of B

Then type [P, D, Q] = svds(B) to see the truncated SVD of B. Carefully note both the similarities and differences between the truncated and non-truncated results.

Now, notice that even the truncated SVD can be trimmed further, by removing the 0 singular value, and trimming P and Q appropriately.   
MATLAB will do this for you automatically if you replace svds(B) with svds(B,3).

P =

-0.500000000000000 0.500000000000000 -0.707106781186547

-0.500000000000000 0.500000000000000 0.707106781186547

-0.500000000000000 -0.500000000000000 0.000000000000000

-0.500000000000000 -0.500000000000000 -0.000000000000000

D =

1.0e+02 \*

1.600000000000000 0 0

0 0.160000000000000 0

0 0 0.020000000000000

Q =

-0.500000000000000 0.499999999999999 0.707106781186548

-0.500000000000000 0.500000000000000 -0.707106781186548

-0.500000000000000 -0.500000000000000 0.000000000000001

-0.500000000000000 -0.500000000000000 -0.000000000000001

Construct P1, D1, and Q1 in this way, and confirm that P1\*D1\*transpose(Q1) really does equal B.

S = D(1,1)\*P(:,1)\*Q(:,1)'+D(2,2)\*P(:,2)\*Q(:,2)'+D(3,3)\*P(:,3)\*Q(:,3)'

S=

43 45 36 36

45 43 36 36

36 36 44 44

36 36 44 44

This is exactly the same matrix as B.

Now trim P, D, and Q to keep only the 2 largest singular values via the command svds(B,2).

This time, there *will* be a loss of accuracy, but a gain in data compression. Construct P2, D2 and Q2 and compare P2\*D2\*transpose(Q2) to B.

svds(B,2)

D = [160 0; 0 16]

S = D(1,1)\*P(:,1)\*Q(:,1)'+D(2,2)\*P(:,2)\*Q(:,2)'

S =

44 44 36 36

44 44 36 36

36 36 44 44

36 36 44 44

There is a loss of accuracy.

Suggestion: In The Journal of Online Mathematics and Its Applications (JOMA), read the article: “The Linear Algebra Behind Search Engines”: [http://mathdl.maa.org/mathDL/4/?pa=content&sa=vi.e.wDocument&nodeId=636](http://mathdl.maa.org/mathDL/4/?pa=content&sa=viewDocument&nodeId=636)

**MATH 284 – Part C**

As is explained in the reading, each of the 3+billion pages on the world wide web can be modeled as a vector in R300000 (note: we’ll use the number 3 billion, even though a more current estimate is that there are over 10 times that many pages!) Given any search criterion, also represented as a vector in R300000 , you could look for the page or pages that most closely resembles your vector by finding the angles between each page and your vector, and finding the angle closest to 0 (or, equivalently, the cosine closest to 1).

This is not really practical, if you want an answer within your lifetime, so some data compression is needed. Fortunately, there is a lot of redundancy – both in terms of similar webpages and, more importantly, in the English language. We can assume that any search on one word is equivalent to a search on any synonym.

We are going to model this on a smaller scale:

In this project, we are modeling the web as containing 900 articles, each of which is relevant, to varying degrees, on 200 search terms (or words). However, due to relationships between the words, there are really only 50 “distinct” words.

***PLEASE suppress printing (use semicolons) for the huge matrices!!!!!!!!!!!***

Create a random 50 by 900 matrix (representing 50 words and 900 articles to search), and call it A.

Now replace A by vertcat(A,A,A,A)

A = rand (50, 900)

A = vertcat(A,A,A,A)

**(NOTE: these questions are meant to help guide you in your work on the final part of the project. You *should* answer them all, but *do not need to submit these*).**

What does this command do?

Vertically concatenates A

In the context of the application, why did we use this command (i.e., the original A was set up to model 50 words and 900 articles. What does the new matrix represent, and why did we set it up in this way?)

We are modeling it for 200 queries and 900 articles

If we want to simply store all of the elements of A in a computer, how many numbers must we keep track of?

180000

What is the rank of A? (Note: You should be able to determine what this value (probably) is without asking MATLAB! How?)

50

Let [p,d,q]=svds(A,30). How many numbers must be stored to track p?

6000

For q?

27000

For d, the computer “knows” it is diagonal, and so only needs to store the numbers on the diagonal.

In reality, we are concerned more with the number of flops – arithmetic operations – than the number of elements in the matrices, but these are going to be roughly proportional, so this is a reasonable measure of savings.

Let B = p\*d\*transpose(q) (AGAIN SUPPRESS PRINTING!)

What does B represent, in the context of the application?

B represents p times d times the transpose of q

The part of the assignment that will be graded is given below. It is a continuation of the work above (so, for example, the matrices A and B are the ones generated previously).

**For questions 1 and 2 below, submit your work electronically as an m-file**.

We want to do a search for the 1st article – column #1 in A. Obviously, we should hope to find that article #1 is an exact match, and then determine which of the other articles are most similar.

**1)** Let the vector x be the 1st column of A.

Find the cosine of the angle between x and each column in A. (column 1 better have a cosine of 1!)

Rank the columns by their cosines, and list the *column* *numbers* of the 8 highest ranked columns, together with their cosines.

For example, your output might be something like:

column 1 has cosine 1, column 234 has cosine .921, column 83 has cosine .904, etc. (You do not need to have all these words- an answer along the lines of 1,1; 234, .921; 83, .904; … is acceptable)

***Please don’t actually list the columns – remember, they have 200 elements each!***

**2)** Now, find the 8 articles that most closely resemble x (the same “x” from above) using the columns of

*B*.

Report your findings in the same way you did in #1.

Exported m-file for Part C - Questions 1 and 2:

A = rand(50,900);

A = vertcat(A,A,A,A);

[p,d,q] = svds(A,30);

B = p\*d\*transpose(q);

C = zeros(900, 2);

for i = 1:900

y = A(:,i);

x = A(:,1);

C(i,1) = dot(x,y)/(norm(x)\*norm(y));

C(i,2) = i;

end

D = sortrows(C);

D(893:900,1);

E = sortrows(C(:,2));

F = zeros(900, 2);

F = cat(2,D,E);

M = D(893:900,:);

M = D(893:900,:)

**3)** After completing the MATLAB analysis, write a report EXPLAINING the math you performed and what the results were. You should address the results you would expect to find in part (2), and why things worked (or didn’t work) well, as well as what tradeoffs are available in time vs accuracy.

Matlab is an important tool, especially in math, to calculate, analyze, and compute large sets of data. In Part C, I used Matlab as a search engine to model the web containing 900 articles. I began by performing some calculations: creating a random 50 (words) by 900 (articles) matrix, A = rand (50, 900). Then I

replaced A by vertcat(A,A,A,A) which vertically concatenates the matrix, A = vertcat(A,A,A,A). Even though the original A was set up to model 50 words and 900 articles, I used this command to model the application for 200 queries and 900 articles. In order to store the elements on the computer, we keep track of 180000 numbers. Moreover, I calculated the rank of my matrix to be 50. Then, using the command [p,d,q]=svds(A,30), I found how many numbers must be stored to track p, 6000 and q, 27000. Then I used the command B = p\*d\*transpose(q), B represents p times d times the transpose of q. For part two, I sorted the rows and the result for part 2 was the following: M = [0.8387 862.0000; 0.8393 741.0000; 0.8406 413.0000; 0.8431 458.0000; 0.8520 126.0000; 0.8587 858.0000; 0.8712 23.0000; 1.0000 1.0000].

I expected the search to show a result like this one because the rows are sorted. If you print column x and y, you will be able to see the 900 elements. One thing that worked was the dot product and rank of the columns, which were concatenated and sorted in order. The time complexity of the algorithm employed is log(n) as its worst case and log(1) as its best case, which is efficient in terms of sorting the numerous articles. One tradeoff in accuracy could be the limit of only four significant figures in terms of decimals of data. Also, concatenating the two columns D, and E is an extra step, which could increase the amount of time and space needed for the computer when doing a search.

**Due date: MAY 11, 2016, 11:59 PM**

**Submit your project electronically on Blackboard**

**No late submissions will be accepted**