Eigenvalues and Eigenvectors

MAT 67L, Laboratory III

Contents

Instructions

(1) Read this document.
(2) The questions labeled "Experiments" are not graded, and should not be turned in. They are designed for you to get more practice with `MATLAB` before you start working on the programming problems, and they reinforce mathematical ideas.
(3) A subset of the questions labeled "Problems" are graded. You need to turn in `MATLAB` M-files for each problem via Smartsite. You must read the "Getting started guide" to learn what file names you must use. Incorrect naming of your files will result in zero credit. Every problem should be placed is its own M-file.
(4) Don't forget to have fun!

Eigenvalues

One of the best ways to study a linear transformation

$$f : V \longrightarrow V$$

is to find its eigenvalues and eigenvectors or in other words solve the equation

$$f(v) = \lambda v\,, \qquad v \neq 0\,.$$

In this MATLAB exercise we will lead you through some of the neat things you can to with eigenvalues and eigenvectors. First however you need to teach MATLAB to compute eigenvectors and eigenvalues. Lets briefly recall the steps you would have to perform by hand: As an example lets take the matrix of a linear transformation $f : \mathbb{R}^3 \to \mathbb{R}^3$ to be (in the canonical basis)

$$M := \begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{pmatrix}\,.$$

The steps to compute eigenvalues and eigenvectors are

(1) Calculate the characteristic polynomial

$$P(\lambda) = \det(M - \lambda I)\,.$$

(2) Compute the roots λ_i of $P(\lambda)$. These are the eigenvalues.
(3) For each of the eigenvalues λ_i calculate

$$\ker(M - \lambda_i I)\,.$$

The vectors of any basis for for $\ker(M - \lambda_i I)$ are the eigenvectors corresponding to λ_i.

Computing Eigenvalues and Eigenvectors with MATLAB. As a warning, there are two very different ways to compute the characteristic polynomial of a matrix in `MATLAB`.

```
>> M =[1 2 3;2 4 5; 3 5 6]

M =

     1     2     3
     2     4     5
     3     5     6

>> %compute the characteristic polynomial
>> poly(M)

ans =

    1.0000   -11.0000    -4.0000     1.0000

>> %convert M into a symbolic matrix using the symbolic math toolkit
>> sym B;
>> B= sym(M)

B =

[ 1, 2, 3]
[ 2, 4, 5]
[ 3, 5, 6]

>> %notice how matlab printed out the symbolic matrix differently than M

>> %compute the characteristic polynomial again
>> poly(B)
ans =

x^3 - 11*x^2 - 4*x + 1
```

Next, let us compute the eigenvalues by finding the roots of the characteristic polynomial and the eigenvectors of M.

```
>> eigenValues = roots(poly(M))

eigenValues =

   11.3448
   -0.5157
    0.1709

>> for i = 1:length(eigenValues)
>>  shiftedM = M - eigenValues(i)*eye(size(M));
>>  rref(shiftedM)
>> end
```

```
ans =

    1.0000         0   -0.4450
         0    1.0000   -0.8019
         0         0         0


ans =

    1.0000         0    1.2470
         0    1.0000    0.5550
         0         0         0


ans =

    1.0000         0   -1.8019
         0    1.0000    2.2470
         0         0         0
```

It is now easy to see that the kernel of these three matrices are $(0.4450, 0.8019, 1)^T$, $(-1.2470, -0.5550, 1)^T$, and $(1.8019, -2.2470, 1)^T$

To double check this, run the following in MATLAB and you should get vectors very close to zero.

```
v1= [0.4450, 0.8019, 1]';
v2= [-1.2470, -0.5550, 1]';
v3= [1.8019, -2.2470, 1]';

%these should be close to zero.
M*v1 - eigenValues(1)*v1
M*v2 - eigenValues(2)*v2
M*v3 - eigenValues(3)*v3
```

Symmetric Matrices. The matrix M in the above example is symmetric, *i.e.* $M = M^T$ (remember that the transpose is the mirror reflection about the diagonal). It turns out (we will learn why from Chapter 11 of the book) that symmetric matrices can always be diagonalized. Recall that to diagonalize a matrix M you need to find a basis of eigenvectors and arrange these (or better said their components) as the columns of a change of basis matrix P. Then it follows that

$$MP = PD$$

where D is a diagonal matrix of eigenvalues. Thus $D = P^{-1}MP$.

Diagonalization with MATLAB. Above, we computed the eigenvalues and vectors the long and hard way, but MATLAB has a function that will make your life easy:

```
>> [P, D] = eig(M)
```

```
P =

    0.7370    0.5910    0.3280
    0.3280   -0.7370    0.5910
   -0.5910    0.3280    0.7370


D =

   -0.5157         0         0
         0    0.1709         0
         0         0   11.3448
```

The ith column of P is an eigenvector corresponding to the eigenvalue in the ith column of D. Notice how MATLAB changed the order the eigenvectors from the previous way I wrote them down. Also, MATLAB normalized each eigenvector, and changed the sign of v_2. This is ok because eigenvectors that differ by a non-zero scaler are considered equivalent.

```
>> v1/norm(v1)

ans =

    0.3280
    0.5910
    0.7370

>> v2/norm(v2)

ans =

   -0.7370
   -0.3280
    0.5910

>> v3/norm(v3)

ans =

    0.5910
   -0.7370
    0.3280
```

For fun, let us check that $MP = PD$:

```
>> M*P

ans =

   -0.3801    0.1010    3.7209
   -0.1692   -0.1260    6.7049
    0.3048    0.0561    8.3609
```

```
>> P*D

ans =

   -0.3801    0.1010    3.7209
   -0.1692   -0.1260    6.7049
    0.3048    0.0561    8.3609
```

Orthogonal Matrices. Symmetric matrices have another very nice property. Their eigenvectors (for different eigenvalues) are orthogonal. Hence we can rescale them so their length is unity to form an orthonormal basis (for any eigenspaces of dimension higher than one, we can use the Gram-Schmidt procedure to produce an orthonormal basis). Then when we form the change of basis matrix O of orthonormal basis vectors we find

$$O^T O = I\,,$$

so that O is an "orthogonal matrix". The diagonalization formula is now

$$D = O^T M O\,.$$

Dot Products and Transposes with MATLAB. Taking the standard dot product over real or complex vectors in MATLAB is easy as it can be done by multiplying two vectors together:

```
>> %remove the variable i from memory so matlab will know i is the complex number
    .
>> clear i
>> a=[1; 3+i; 5+i]

a =

   1.0000
   3.0000 + 1.0000i
   5.0000 + 1.0000i

>> b = [1; 2 ; 3]

b =

     1
     2
     3
>> % a' will give the Hermitian transpose
>> a'

ans =

   1.0000    3.0000 - 1.0000i    5.0000 - 1.0000i

>> a'*b
```

```
ans =

  22.0000 - 5.0000i
```

Because `MATLAB` already gave us normalized eigenvectors, in the notation of the last sections we have $O = P$. Let us check $O^T O = I$ and $D = O^T M O$.

```
>> P

P =

    0.7370    0.5910    0.3280
    0.3280   -0.7370    0.5910
   -0.5910    0.3280    0.7370

>> P* P'

ans =

    1.0000    0.0000         0
    0.0000    1.0000    0.0000
         0    0.0000    1.0000

>> P' * M * P

ans =

   -0.5157    0.0000    0.0000
    0.0000    0.1709   -0.0000
    0.0000   -0.0000   11.3448

>> D

D =

   -0.5157         0         0
         0    0.1709         0
         0         0   11.3448
```

Functions of matrices. If you can diagonalize a matrix then it is possible to compute functions of matrices $f(M)$ by writing $f(x) = 1 + x + \frac{1}{2!}x^2 + \cdots$ (a Taylor series) and use the fact that $M^k = (P^{-1}DP)^k = P^{-1}D^kP$ so that

$$f(M) = P^{-1}\big(I + D + \frac{1}{2!}D^2 + \cdots\big)P = P^{-1}f(D)P\,.$$

(so long as the series converges on the eigenvalues). Also $f(D)$ is the matrix with $f(\lambda_i)$ along the diagonal.

Functions of matrices with MATLAB. `MATLAB` can compute some functions on matrices like e^M and $\sqrt{M}$, but care must be taken. For example, to compute e^M, the `MATLAB`

function "expm(M)" must be used instead of "exp(M)" as the latter will simply return there matrix where each entry is replaced with $e^{m_{ij}}$, which is not the same thing as taking the matrix exponential. Also, for some functions (like the square root of a negative or complex eigenvalue) such things like branch cuts must be considered.

```
>> expm(M) - P*expm(D)*P'

ans =

   1.0e-09 *

    0.0327    0.0546    0.0655
    0.0546    0.1055    0.1237
    0.0655    0.1164    0.1310

>> sqrtm(M) - P*sqrtm(D)*P'

ans =

     0     0     0
     0     0     0
     0     0     0
```

Exercises.

Problem 1.

Given a diagonalizable matrix A, let the list $(\lambda_1, \ldots, \lambda_n)$ be the eigenvalues. Write a function that will compute the sum $\sum_i \lambda_i$ and the product $\prod_i \lambda_i$.

The function signature is

```
%Input
%   A: n x n diagonalizable matrix
%Output
%   theSum: sum of eigenvalues
%   theProd: product of eigenvalues
function [theSum, theProd] = ev_sumAndProduct(A)
    code here
end
```

You may NOT use `MATLAB`'s "eig" function. If you do, you will get zero points.

Hint: you may want to research what properties the determinant and trace function have.

Problem 2.

Given a diagonalizable matrix A, write a function that return true if A is positive definite and false otherwise.

The function signature is

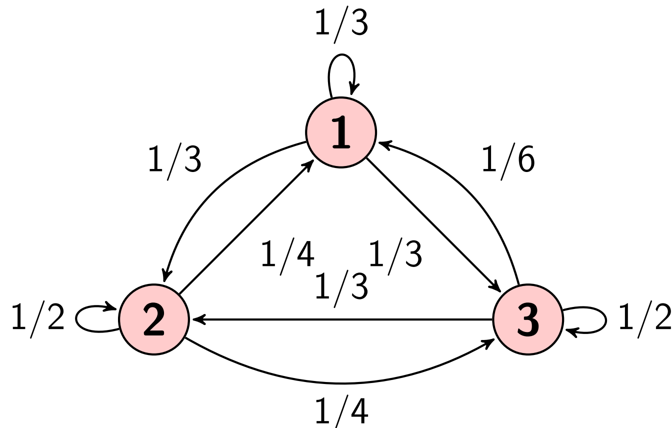
```
%Input
%   A: n x n diagonalizable matrix
%Output
%   isPosDef: true if positive definite, false otherwise
function [isPosDef] = ev_isPositiveDefinite(A)
    code here
end
```

A matrix A is called positive definite if $\forall x \neq 0, x^T Ax > 0$. Hint, use the fact that A is diagonalizable and the eigenvector matrix in invertible.

Another hint, the value "true" in `MATLAB` is typed like this: true.

Problem 3.

Consider the following Markov Chain on the graph below. The nodes are called states, and the edge weights represent a probability of moving from a state to another state.



We can encode the probability of moving from one state to another in a matrix

$$P = \begin{pmatrix} 1/3 & 1/3 & 1/3 \\ 1/4 & 1/2 & 1/4 \\ 1/6 & 1/3 & 1/2 \end{pmatrix}$$

where P_{ij} is the probability of moving from state i to state j.

If you start at state 1, then you move to a new state randomly according to the probabilities on node 1. Using matrix notation, $e_1 P$ is a probability vector of which state you will be in after one iteration, where $e_1 = (1, 0, 0)$ encodes you are starting from state 1. It can be shown that $e_1 P^2$ is the probability vector of where you will be if you start at state 1 and then randomly moved twice. This generalizes to $e_1 P^n$. Now lets say you do not know which state you are going to start with; instead, you know you will start at state 1 with probability 1/2 or state 2 with probability 1/2. Then after 1 step, your probability distribution of where you will be is $(1/2, 1/2, 0)P = (7/24, 5/12, 7/24)$ (so there is a 7/24 chance you will be in state 1). After n steps, your probability distribution is $(1/2, 1/2, 0)P^n$.

An interesting question is that does there exist an initial probability distribution row vector π such that your probability distribution as you walk does not change: $\pi P = \pi$? Write a function to find such a starting probability distribution.

The function signature is

```
%Input
%   P: n x n diagonalizable Markov matrix
%Output
%   pd: row vector such that pd * P = pd and the sum of the elements in pd is 1.
%Example
%   For the P defined above, pd=[6/25, 2/5, 9/25]
function [pd] = ev_MarkovSteadyState(P)
    code here
end
```

Hint, use “eig” in a special matrix.

Problem 4.

Let us breed rabbits, starting with one pair of rabbits. Every month, each pair of rabbits produces one pair of offspring. After one month, the offspring is an adult, and will start reproduction. We neglect all kinds of effects like death and we always consider pairs of rabbits.

To model this dynamical system, we use the space of rabbit vectors $r = (j, a)^T \in \mathbb{R}^2$. The first component of the rabbit vector gives the number of a juvenile pairs, the second component the number of adult pairs of rabbits. Translating the standard Fibonacci recurrence into matrix notation, we get

$$r_{n+1} = \begin{pmatrix} j_{n+1} \\ a_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} j_n \\ a_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} r_n =: Ar_n$$

The standard Fibonacci sequence uses $r_0 = (1, 0)^T$, but we could start $r_0 = (a, b)^T$ at any point in $\mathbb{R}^2$.

Let M be a 3-row k-column matrix. Write a function that takes M and returns a k-vector f where f_i is the $m_{3,i} - th$ Fibonacci sequence number where $r_0 = (m_{1,i}, m_{2,i})$. Compute the f vector using the above matrix recursion (in a smart way).

For example, if

$$M = \begin{pmatrix} 1 & 1 & 3 \\ 0 & 0 & 5 \\ 1 & 2 & 10 \end{pmatrix},$$

then the f vector is $(1, 2, 987)$, because $1 = sum(A(1, 0)^T)$, $2 = sum(A^2(1, 0)^T)$, and $987 = sum(A^{10}(3, 5)^T)$

```
%Input
%   M: description is above.
%Output
%   f:  description is above.
function [f] = ev_FastFibonacci(M)
    code here
end
```

Remark: M will contain on the order of 100000 columns. The integers in the 3rd row of M will contain many digits. The submatrix $M(1:2,:)$ may not be integer.

Remark: Your function will be timed and the amount of memory it uses will be recorded. If it takes too long or uses to much memory, I will assume your function is simply computing A^n and you will get zero credit. You must think of a way to find A^n quickly.