AEM-ADV08 Computational Linear Algebra Coursework 2

Name: Wenze Liu Chen, **CID:** 01526073

3. Write a code using the same development environment as used in 2 which can solve a general/standard NxN eigenvalue problem using the following techniques:

a) Rayleigh Quotient Iteration along with Householder deflation – write subroutines RQI and HHD for this purpose

Rayleigh Quotient Iteration, RQI:

```
function [lambda, evec] = RQI(A)
%RQI performs the rayleigh quotient iteration
%   A - the input matrix
%   lambda - eigenvalues of A
%   evec - each column corresponds to an eigenvector of A

%transform the input matrix into sparse matrix format
A = sparse(A);
%save the input matrix A
inp_A = A;
%get the length of input matrix
m = length(A);

for i = 1:m
%get the lenght of deflated matrix
n = length(A);
if n == 1
    %when the matrix is deflated to a number, the eigenvalue is the
numberitself
    lambda(i,i) = A;
    evec = 1;
else
%create a random initial x-vector
evec = ones(n,1);
%create the identity matrix
I = eye(n);
%compute the rayleigh quotient
lambda(i,i) = (evec'*A*evec)/(evec'*evec);
%initilisation of error value
err = 1;
%set convergence criteria
conv_crit = 1e-8;
%initilisation of iteration number
n_iter = 0;
%perform rayleigh quotient iteration
while err > conv_crit
    %compute number of iteration
    n_iter = n_iter + 1;
    %compute the shifted matrix
    A_shifted = A - lambda(i,i)*I;
    %compute the next x-vector using function that solves a linear system
    [evec] = SLSYS_LU(A_shifted,evec);
    %save the lambda of last iteration
    lambda_old = lambda(i,i);
    %compute the rayleigh quotient
    lambda(i,i) = (evec'*A*evec)/(evec'*evec);
    %compute the error between each iteration
    err = abs(lambda(i,i) - lambda_old);
```

```
end
%comput the unit vector of eigenvector
evec = evec/norm(evec);
%apply householder deflation
 A = HHD(A,evec);
end
end

%use shifted iverse iteration to calculate eigenvectors
%initilisation of eigenvector matrix
evec  = ones(m);
%initilisation of error value
err = 1;
while err > conv_crit
    %save the eigenvector of last iteration
    evec_old = evec;
    for i = 1:m
        %perform shifted inverse iteration
        evec(:,i) = SLSYS_LU(inp_A - eye(m)*(lambda(i,i)-0.1*lambda(i,i)),
evec(:,i));
        %normalise each eigenvector
        evec(:,i) = evec(:,i)/norm(evec(:,i));
    end
    %compute error
    err = max(max(abs(evec - evec_old)));
end
%end of RQI function
end
```

Householder deflation, HHD:

```
function [A_defl] = HHD(A,x)
%this function performs householder deflation of the input A matrix
%   A - input matrix
%   x - eigenvector of input matrix A
%   A_defl - deflated A matrix

%transform the input matrix into sparse matrix format
A = sparse(A);
n = length(A);
if n==1
    A_defl = A;
else
%householder deflation
alpha = norm(x,2);
v(1,1) = sqrt(0.5*(1-x(1)/alpha));
for i = 2:n
    v(i,1) = x(i)/(-2*alpha*v(1,1));
end
H = eye(n) - 2*(v*v');
G = H*A*H';
%extract the deflated matrix
A_defl = G(2:n,2:n);
%end of HHD function
end
```

Function that solves linear system by LU decomposition, SLSYS_LU:

```
function [b] = SLSYS_LU(A,b)
%SLSYS_LU solves a linear system Ax = b using LU decomposition with pivoting
%   A - input matrix
%   b - input, right hand side of the linear system
%   b - output, solution of the system

%get col and row number of the rhs
[m,n] = size(b);
%initialisation of unknown x vector/matrix
```

```
x = zeros(m,n);
%perform LU decomposition
[L, U, P] = lu(A);
%multiply the rhs by permutation matrix
b = P*b;
for i = 1:n
% forward substitution
b(1,i) = b(1,i)/L(1,1);%compute the first entry of the solution of lower
triangular matrix
for j = 2:m %perform the forward substitution from second entry
   b(j,i) = (b(j,i)-L(j,1:j-1)*b(1:j-1,i))/L(j,j);
end
%backward substitution
b(m,i) = b(m,i)/U(m,m); %compute the last entry of the solution of upper
triangular matrix
for k = m-1:-1:1 %perform the backward substitution from n-1 entry
   b(k,i) = (b(k,i)-U(k,k+1:m)*b(k+1:m,i))/U(k,k);
end
end
%end of SLSYS_LU function
end
```

b) QR iteration technique – write QRITER for this purpose

QR iteration, QRITER:

```
function [eig_val, eig_vec] = QRITER(A)
%QRITER performs QR iteration to obtain the eigenvalues and eigenvectors
%   A - input matrix A
%   eig_val - eigenvalue matrix, each diagonal entry corresponds to an
eigenvalue
%   eig_vec - eigenvector matrix, each column corresponds to an eigenvector

%transform the input matrix into sparse matrix format
A = sparse(A);
%initialisation of error value
err = 1;
%set convergence criteria
conv_crit = 1e-10;
%initialisation of iteration number
n_iter = 0;
%get the size of matrix A
n = length(A);
U = A;
%perform QR iteration
while err > conv_crit
    %conut the iteration number
    n_iter = n_iter + 1;
    %store the U from last iteration
    U_old = U;
    % compute the first orthonormal unit vector
    Q(:,1) = U(:,1)/norm(U(:,1));
    %construct the orthonormal unit vector
    for i = 2:n
        j = 2;
        while (j<=i)
        U(:,i) = U(:,i) - U_old(:,i)'*Q(:,j-1)*Q(:,j-1);
        j = j+1;
        end
        %if the vectors coincide then set to 0
        if norm(U(:,i))<conv_crit
            Q(:,i) = zeros(n,1);
        else
            Q(:,i) = U(:,i)/norm(U(:,i));
        end
    end
    %compute the upper triangular matrix R
    R = Q'*U_old;
    %compute the eigenvalue matrix
    U = R*Q;
    %compute the error between each iteration
    err = abs(max(max(diag(U_old - U))));
end

%extract the eigenvalues form the U matrix
for i = 1:n
eig_val(i,i) = U(i,i);
end

%use shifted iverse iteration to calculate eigenvectors
%initialisation of eigenvector matrix
eig_vec  = ones(n);
%initialisation of error value
err = 1;
while err > conv_crit
    %save the eigenvector of last iteration
    evec_old = eig_vec;
    for i = 1:n
```

```
        %perform shifted inverse iteration
        eig_vec(:,i) = SLSYS_LU(A - eye(n)*(eig_val(i,i)-0.1*eig_val(i,i)),
eig_vec(:,i));
        %normalise each eigenvector
        eig_vec(:,i) = eig_vec(:,i)/norm(eig_vec(:,i));
    end
    %compute error
    err = max(max(abs(eig_vec - evec_old)));
end
%end of function QRITER
end
```

Function that solves linear system by LU decomposition, SLSYS_LU:

```
function [b] = SLSYS_LU(A,b)
%SLSYS_LU solves a linear system Ax = b using LU decomposition with pivoting
%   A - input matrix
%   b - input, right hand side of the linear system
%   b - output, solution of the system

%get col and row number of the rhs
[m,n] = size(b);
%initialisation of unknown x vector/matrix
x = zeros(m,n);
%perform LU decomposition
[L, U, P] = lu(A);
%multiply the rhs by permutation matrix
b = P*b;
for i = 1:n
% forward substitution
b(1,i) = b(1,i)/L(1,1);%compute the first entry of the solution of lower
triangular matrix
for j = 2:m %perform the forward substitution from second entry
   b(j,i) = (b(j,i)-L(j,1:j-1)*b(1:j-1,i))/L(j,j);
end
%backward substitution
b(m,i) = b(m,i)/U(m,m); %compute the last entry of the solution of upper
triangular matrix
for k = m-1:-1:1 %perform the backward substitution from n-1 entry
   b(k,i) = (b(k,i)-U(k,k+1:m)*b(k+1:m,i))/U(k,k);
end
end
%end of SLSYS_LU function
end
```

c) Subspace Iteration- write subroutines SSI and RITZ for this purpose

<u>Subspace Iteration, SSI:</u>

```
function [val, X] = SSI(K,M,n)
%This function performs subspace iteration
%   K - the input k stiffness matrix
%   M - the input m mass matrix
%   n - number of eigenvalues/vectors to calculate
%   X - output eigenvector matrix
%   val - output eigenvalue matrix

%transform the input matrices into sparse matrices format
K = sparse(K);
M = sparse(M);
%create a random initial matrix X0 for iteration
X = rand([length(K),n]);
%initialisation of error
err = 1;
%convergence criteria
conv_crit = 1e-8;
%calculate K^-1*M
inv_KM = SLSYS_LU(K,M);

%subspace iteration
while err > conv_crit
    X_old = X;
    X = inv_KM * X;
    %solve the Ritz Problem
    [val,X] = RITZ(K,M,X);
    %normalised the vectors
    for j = 1:n
    X(:,j) = X(:,j)/norm(X(:,j));
    end
    %calculate the error between two iterations
    err = max(max(abs(X - X_old)));
end
%end of subspace iteration
end
```

<u>Ritz Eigenvalue Problem, RITZ:</u>

```
function [eig_val, eig_vec] = RITZ(K,M,x)
%This function solves the Ritz eigenvalue problem
%   K - the input k stiffness matrix
%   M - the input m mass matrix
%   x - the input x matrix for iteration which will eventually converge to a
%   matrix with each column corresponding to an eigenvector
%   eig_val - output eigenvalue
%   eig_vec - output eigenvector

    %define Ritz EVP
    kbar = x'*K*x;
    mbar = x'*M*x;
    %calculate the left hand side matrix of EVP -> mbar^-1*kbar
    A = SLSYS_LU(mbar, kbar);
    %use qr iteration to solve the Ritz EVP
    [eig_val, evc] = QRITER(A);
    %compute the eigenvector/x for next subspace iteration
    eig_vec = x*evc;
    %end of RITZ function
end
```

d) In the development of codes, advantage must be taken of the banded structures of matrices involved. Application of methods such as Skyline storage method is encouraged.

To take advantage of banded matrices, the MATLAB built-in function *sparse* is used to convert the banded matrices into sparse matrix format. This function is used on the first few lines of function RQI, HHD, QRITER and SSI, as can be seen from the codes above. If the input matrix is a sparse or banded matrix, the use of *sparse* can reduce the cost of computational resource, since only the non-zero values will be saved and taken into the computation processes.

e) Comment on the sensitivity of each method to numerical errors in A matrix

Consider the following symmetric matrix A (symmetric matrix guarantees real eigenvalues and eigenvectors):

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 6 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

which gives the following eigenvalues and eigenvectors:

$$\lambda = \begin{bmatrix} 7.7417 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0.2583 \end{bmatrix}, v_1 = \begin{bmatrix} 0.3148 \\ 0.8489 \\ 0.4245 \end{bmatrix}, v_2 = \begin{bmatrix} 0 \\ -0.4472 \\ 0.8944 \end{bmatrix}, v_3 = \begin{bmatrix} 0.9492 \\ -0.2816 \\ -0.1408 \end{bmatrix}$$

Now incorporate randomly small perturbation to the A matrix:

$$A' = \begin{bmatrix} 1.003 & 2.0002 & 1.0008 \\ 2.002 & 6.001 & 2.002 \\ 1.001 & 2.0004 & 3.002 \end{bmatrix}$$

Results from RQI with HHD:

$$\lambda = \begin{bmatrix} 7.7447 & 0 & 0 \\ 0 & 2.0008 & 0 \\ 0 & 0 & 0.2604 \end{bmatrix}, v_1 = \begin{bmatrix} 0.3149 \\ 0.8489 \\ 0.4245 \end{bmatrix}, v_2 = \begin{bmatrix} 0.000116 \\ -0.4475 \\ 0.8943 \end{bmatrix}, v_3 = \begin{bmatrix} 0.9491 \\ -0.2819 \\ -0.1409 \end{bmatrix}$$

Results from QRITER:

$$\lambda = \begin{bmatrix} 7.7447 & 0 & 0 \\ 0 & 2.0008 & 0 \\ 0 & 0 & 0.2604 \end{bmatrix}, v_1 = \begin{bmatrix} 0.3149 \\ 0.8489 \\ 0.4245 \end{bmatrix}, v_2 = \begin{bmatrix} 0.000116 \\ -0.4475 \\ 0.8943 \end{bmatrix}, v_3 = \begin{bmatrix} 0.9491 \\ -0.2819 \\ -0.1409 \end{bmatrix}$$

Results from SSI with RITZ:

$$\lambda = \begin{bmatrix} 7.7447 & 0 & 0 \\ 0 & 2.0008 & 0 \\ 0 & 0 & 0.2604 \end{bmatrix}, v_1 = \begin{bmatrix} 0.3149 \\ 0.8489 \\ 0.4245 \end{bmatrix}, v_2 = \begin{bmatrix} 0.000116 \\ -0.4475 \\ 0.8943 \end{bmatrix}, v_3 = \begin{bmatrix} 0.9491 \\ -0.2819 \\ -0.1409 \end{bmatrix}$$

Looking at the results from different techniques, their behaviour to the perturbation incorporated is exactly the same. This implies that all of these techniques are having the same or very similar sensitivity to the numerical errors. The perturbation added to the matrix A ranges from 0.3% to 0.01% for different entries. The largest change in eigenvalues is about 0.21%, which is not a very big percentage, thus the eigenvalues is not sensitive to the numerical errors. The first eigenvector seems to be insensitive to the perturbation added, it remains the same with incorporation of perturbation, the largest change in the second eigenvector is about 0.067% and the largest change in the third eigenvector is about 0.071%. From these percentages, it can be concluded that the sensitivity of all three techniques to the numerical errors is quite low, small perturbation will not introduce unacceptable levels of error to the eigenvalues and eigenvectors.

4. Using the codes developed in part 3 and for N=4, calculate the eigenvalues and eigenvectors of the bar. Then, compare the accuracy and the cost of the computations for the various techniques. The cost is comprised of computation time and storage capacity required for a particular code.

For $N = 4$, the stiffness matrix and mass matrix are given by:

$$K = \begin{bmatrix} 9.4955 & -4.0024 & 0 & 0 \\ -4.0024 & 6.9048 & -2.9024 & 0 \\ 0 & 2.9024 & 4.9956 & -2.0932 \\ 0 & 0 & -2.0932 & 2.0932 \end{bmatrix},$$

$$M = \begin{bmatrix} 3.2525 & 0.6486 & 0 & 0 \\ 0.6486 & 2.0868 & 0.4151 & 0 \\ 0 & 0.4151 & 1.332 & 0.2642 \\ 0 & 0 & 0.2642 & 0.4709 \end{bmatrix}$$

These matrices are obtained from the document upload by Prof. Ali Nobari on Blackboard. Input the above matrices into MATLAB and use the codes developed in part 3:

```
clear all; close all;

M = [3.2525  0.6486  0       0      ;
     0.6486  2.0868  0.4151  0      ;
     0       0.4151  1.332   0.2642 ;
     0       0       0.2642  0.4709];

K = [9.4955   -4.0024  0         0       ;
     -4.0024  6.9048   -2.9024   0       ;
     0        -2.9024  4.9956    -2.0932 ;
     0        0        -2.0932   2.0932 ];

%transform the general EVP to standard EVP, K*x = M*eval*x -> M^-1*K*x =
%eval*x -> A*x = eval*x
A = SLSYS_LU(M, K);

%exact eigenvectors and eigenvalues
[eigvec, eigval] = eig(K, M);
%normalise eigenvectors
for i = 1:length(K)
   eigvec(:,i) = eigvec(:,i)/norm(eigvec(:,i));
end

% RH iteration
[RH_eval, RH_evec] = RQI(A);

% QR iteration
[QR_eval, QR_evec] = QRITER(A);

% Subspace iteration
[SR_eval, SR_evec] = SSI(K,M,3);
```

The results of eigenvalues and eigenvectors using different methods are:

Rayleigh Quotient Iteration along with householder deflation

$$eigenvalue = \begin{bmatrix} 0.3005 & 0 & 0 & 0 \\ 0 & 1.8426 & 0 & 0 \\ 0 & 0 & 12.8218 & 0 \\ 0 & 0 & 0 & 5.7273 \end{bmatrix}$$

$$eigenvectors = \begin{bmatrix} 0.2087 & 0.4293 & 0.1149 & 0.3951 \\ 0.4236 & 0.2893 & -0.3005 & -0.4676 \\ 0.5891 & -0.3671 & 0.5531 & -0.1306 \\ 0.6557 & -0.7728 & -0.7685 & 0.7799 \end{bmatrix}$$

QR iteration

$$eigenvalue = \begin{bmatrix} 12.8218 & 0 & 0 & 0 \\ 0 & 5.7273 & 0 & 0 \\ 0 & 0 & 1.8426 & 0 \\ 0 & 0 & 0 & 0.3005 \end{bmatrix}$$

$$eigenvectors = \begin{bmatrix} 0.1149 & 0.3951 & 0.4293 & 0.2087 \\ -0.3005 & -0.4676 & 0.2893 & 0.4236 \\ 0.5531 & -0.1306 & -0.3671 & 0.5891 \\ -0.7685 & 0.7799 & -0.7728 & 0.6557 \end{bmatrix}$$

Subspace iteration

$$eigenvalue = \begin{bmatrix} 12.8218 & 0 & 0 & 0 \\ 0 & 5.7273 & 0 & 0 \\ 0 & 0 & 1.8426 & 0 \\ 0 & 0 & 0 & 0.3005 \end{bmatrix}$$

$$eigenvectors = \begin{bmatrix} -0.1149 & 0.3951 & 0.4293 & 0.2087 \\ 0.3005 & -0.4676 & 0.2893 & 0.4236 \\ -0.5531 & -0.1306 & -0.3671 & 0.5891 \\ 0.7685 & 0.7799 & -0.7728 & 0.6557 \end{bmatrix}$$

Using MATLAB built-in function *eig( )*:

$$eigenvalue = \begin{bmatrix} 0.3005 & 0 & 0 & 0 \\ 0 & 1.8426 & 0 & 0 \\ 0 & 0 & 5.7273 & 0 \\ 0 & 0 & 0 & 12.8218 \end{bmatrix}$$

$$eigenvectors = \begin{bmatrix} -0.2087 & -0.4293 & 0.3951 & -0.1149 \\ -0.4236 & -0.2893 & -0.4676 & 0.3005 \\ -0.5891 & 0.3671 & -0.1306 & -0.5531 \\ -0.6557 & 0.7728 & 0.7799 & 0.7685 \end{bmatrix}$$

By comparing the results from different methods with those from MATLAB built-in function, it can be seen that the eigenvalues are exactly the same, the eigenvectors are also the same but some eigenvectors with different sign. Thus, all three techniques are able to give results with high accuracy.

After the comparison of the accuracy between different techniques, the computation cost will be discussed in the following. The computation cost can be measured by looking at the time and the storage capacity required to run the function. The built-in function *timeit()* is used to get the approximate computation time of each technique:

```
% Computation time
t_RH = timeit(@()RQI(A),2);
t_QR = timeit(@()QRITER(A),2);
t_SI = timeit(@()SSI(K,M,4),2);
```

For the case of $N = 4$, the time to run each technique is approximately $0.0021s$ for Rayleigh Quotient Iteration (RQI), $0.0104s$ for QR iteration and $0.0137s$ for Subspace iteration. The running time of RQI is the shortest one, meaning that it is the most efficient one among these three techniques. Since there does not exist any built-in function to measure the capacity, the required storage capacity is calculated approximately by looking at the sum of sizes of the intermediate variables created when running the function, the size of variables can be obtained through the function *whos*:

Table 1: Storage capacity required for each technique for N = 4

Techniques	**Approximated required storage (Bytes)**
RQI with HHD	1944
QRI	2144
SSI with RITZ	1960

Form *Table 1*, it can be observed that the RQI with Householder deflation is the technique which requires the least storage capacity; the subspace iteration requires a few bytes more than the RQI; and the Subspace iteration requires the most storage capacity, it is about 10.2% then the RQI with householder deflation. Thus, considering the cost of time and storage together, the RQI with householder technique is the one that uses less computational resources.

5. Write a subroutine SENS which can calculate the sensitivity of eigenvalues of the bar with respect to the changes in the elements' parameters ρ_{0i} , E_{0i}, where i is the element index. Comment on your results.

The sensitivity function SENS is given by the following code:

```
function [ch_eig, S] = SENS(eval,evec,ch_dp)
%SENS calculates the sensitivity of eigenvalues of the bar with respect to
%the changes in material properties rho and E
%   eval - input eigenvalue matrix
%   evec - input eigenvector matrix
%   ch_dp - change of design parameter
%   ch_eig - output change in eigenvalues
%   S - sensitivity matrix

%define cell matrix that contain the elemental stiffness matrices
k{1} = [5.4931  0 0 0;
        0       0 0 0;
        0       0 0 0;
        0       0 0 0];

k{2} = [4.0024   -4.0024  0 0;
        -4.0024  4.0024   0 0;
        0        0        0 0;
        0        0        0 0];

k{3} = [0  0        0        0;
        0  2.9024   -2.9024  0;
        0  -2.9024  2.9024   0;
        0  0        0        0];

k{4} = [0  0  0          0      ;
        0  0  0          0      ;
        0  0  2.0932     -2.0932;
        0  0  -2.0932    2.0932];
%define cell matrix that contain the elemental mass matrices
m{1} = [1.8052  0 0 0;
        0       0 0 0;
        0       0 0 0;
        0       0 0 0];

m{2} = [1.4473   0.6486  0 0;
        0.6486  1.1594   0 0;
        0        0        0 0;
        0        0        0 0];

m{3} = [0  0        0        0;
        0  0.9273   0.4151   0;
        0  0.4151   0.741    0;
        0  0        0        0];

m{4} = [0  0  0          0      ;
        0  0  0          0      ;
        0  0  0.591      0.2642 ;
        0  0  0.2642     0.4709];

%define rho0
rho_0 = 3;
%define E0
E_0 = 4;
%get the number of eigenvalues
n = length(eval);
for i = 1:n
%compute the derivative of dk/dE
dkdE{i} = k{i}./E_0;
%compute the derivative of dM/drho
dMdrho{i} = m{i}./rho_0;
end
%construct the 1st column of sensitivity matrix, which consist of
```

```
%derivatives of eigenvalues with respect to Ei
for i = 1:n
    s1(i,1) = evec(:,i)'*dkdE{i}*evec(:,i);
end
%construct the 2nd column of sensitivity matrix, which consist of
%derivatives of eigenvalues with respect to rhoi
for i = 1:n
    s2(i,1) = evec(:,i)'*(-eval(i,i)*dMdrho{i})*evec(:,i);
end
%combine the columns together to get the sensitivity matrix
S(:,1) = s1;
S(:,2) = s2;
%compute the change in eigenvalues
ch_eig = S*ch_dp;

%end of SENS function
end
```

The SENS function computes the sensitivity of the eigenvalues with respect to the element property E and ρ. The elemental mass and stiffness matrices are obtained from the document upload by Prof. Ali Nobari on Blackboard. Note that SENS requires eigenvalues and eigenvectors as the input. Assume a random change of element properties E and ρ:

$$a = \begin{bmatrix} \Delta \mathrm{E} \\ \Delta \mathrm{\rho} \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.4 \end{bmatrix}$$

The following code is an example of using the function SENS to compute the sensitivity matrix and the new eigenvalues after changing design parameters by amount specified above:

```
clear all; close all;

M = [3.2525  0.6486  0       0      ;
     0.6486  2.0868  0.4151  0      ;
     0       0.4151  1.332   0.2642 ;
     0       0       0.2642  0.4709];

K = [9.4955   -4.0024  0         0       ;
     -4.0024  6.9048   -2.9024   0       ;
     0        -2.9024  4.9956    -2.0932 ;
     0        0        -2.0932   2.0932 ];

% A = M^-1*K
A = SLSYS_LU(M, K);
% RH iteration
[RH_eval, RH_evec] = RQI(A);
%matrix of change of parameters
ch_dp = [0.1; 0.4];

%compute the sensitivity
[deig S] = SENS(RH_eval,RH_evec,ch_dp);

%put the eigenvalues in vector of size n*1
for i = 1:4
    eigval(i,1) = RH_eval(i,i);
end
%calculated the changed eigenvalues
eigval = eigval + deig;
```

The SENS function produces the following sensitivity matrix:

$$S = \begin{bmatrix} 0.0598 & -0.0079 \\ 0.0196 & -0.3224 \\ 0.5287 & -0.7370 \\ 0.4338 & -0.4633 \end{bmatrix}$$

This gives the following change of eigenvalues:

$$\Delta\lambda = \begin{bmatrix} 0.0028 \\ -0.1270 \\ -0.2419 \\ -0.1419 \end{bmatrix}$$

Thus, the eigenvalues will become:

$$\lambda + \Delta\lambda = \begin{bmatrix} 0.3005 \\ 1.8426 \\ 12.8218 \\ 5.7273 \end{bmatrix} + \begin{bmatrix} 0.0028 \\ -0.1270 \\ -0.2419 \\ -0.1419 \end{bmatrix} = \begin{bmatrix} 0.3033 \\ 1.7156 \\ 12.5799 \\ 5.5853 \end{bmatrix}$$

As can be seen from the sensitivity matrix, the entries on the first column are all positive and those on second column are all negative. The first column represents the derivative of eigenvalues with respect to the Young's modulus E and the second column represents the derivative with respect to the density ρ. Thus, increment in E tends to make eigenvalues larger and increment in ρ tends to make the eigenvalues smaller.