Imperial College London

MSC ADVANCED COMPUTATIONAL METHODS

IMPERIAL COLLEGE LONDON

DEPARTMENT OF AERONAUTICS

Computational Linear Algebra Coursework

Author:
Ismael El Houas Ghouddana

Professor:
Dr. Ali Nobari

March 15, 2019

Contents

List of Figures

List of Tables

1 Rayleigh Quotient Iteration and Householder deflation method

Rayleigh Quotient Iteration

The Rayleigh Quotient Iteration algorithm is used to solve the following eigenvalue problem.

$$Ax = \lambda x \tag{1}$$

Multiplying Eq(1) by the transpose of x, the normal equation is obtained

$$x^T Ax = x^T \lambda x \tag{2}$$

The least squares solution gives the **Rayleigh quotient**

$$\lambda = \frac{x^T Ax}{x^T x} \tag{3}$$

We find the eigenvalue by iteration, using the previous formula:

$$\lambda_k = \frac{x^T Ax_k}{x_k^T x_k} \tag{4}$$

Then, shift to accelerate convergence using approximate eigenvalue as shift.

$$(A - \lambda_k * I)y_{k+1} = x_k \tag{5}$$

Finally, normalizing the obtained eigenvectors.

$$x_{k+1} = \frac{y_{k+1}}{||y_{k+1}||_\infty} \tag{6}$$

An error tolerance of 10^{-4} as criteria of convergence in the while loop has been used:

$$||(A - \lambda_k * I)x_{k+1}|| < 10^{-4} \tag{7}$$

The code for the Rayleigh Quotient Iteration is presented as follows, in the subroutine **RQI**:

```
function[x, lambda, RQI_iterations,timeElapsed] =  RQI(A,tol)
tic;
%Get size of Matrix A
[m,n] = size(A);
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

% Initialize vector where the eigenvectors are gonna be stored
x = ones(m,1);

%Initialize eigenvalues
lambda = 1;
k=1; %Counter

    while norm( (A-lambda * eye(m,n))*x )  > tol
        lambda = (x'*A*x)/(x'*x); %Compute the Rayleigh quotient
        y = (A-lambda * eye(m,n))\x; %Shifting
        x = y/norm(y); %Normalize the eigenvectors
        k=k+1;
    end
RQI_iterations = k;
timeElapsed = toc;
end
```

Householder Deflation

Rayleigh Quotient Iteration method computes the dominant eigenvalue and its eigenvector. In order to compute the other eigenvalues, the Householder Deflation Method is used, which removes the known eigenvalues at each iteration.

The hermitanian matrix H needs to be computed in order to solve the following eigenvalue problem:

$$Hx_1 = \alpha e_1 \tag{8}$$

This matrix is computed by:

$$H = I - 2(v * v^T) \tag{9}$$

Where v is the vector perpendicular to the hyperplane of symmetry with eigenvector x obtained from the Rayleigh Quotient Iteration method. In order to compute this vector, the following equation has been used:

$$v = sign(x_1)||x||_2 e_1 + x \tag{10}$$

$$v = \frac{v}{||x||_2}$$

In the equation the sign function of Matlab is used because it is important in terms of stability. Briefly, it avoids that the angle between the hyperplane and the vector would be small which could let to cancellation errors.

In order to compute this perpendicular function, a function called vector perpendicular has been implemented:

```
function[v] =  vector_perpendicular(x,m,i)
    %Get the sign of the first element of the eigenvector found by RQI.
    %This is important in terms of stability. Because the angle between the
    %hyperplane and the vector will be small which could let to
    %cancellation errors.
    p=sign(x(1));
    %Get the euclidian longitude of the vector, by using the parameter 2
    %in norm(x,2) and multiply it by the sign of the first element
    alpha=norm(x,2)*p;
    %Construct the perpendicular vector shrinking the size at each
    %iteration
    v=x+alpha*eye(m+1-i,1);
    %Normalize the perpendicular vector just computed
    v_norm=norm(v);
    %Divide the vector by its norm
    v=v/v_norm;
end
```

H matrix transforms A into:

$$HAH^{-1} = \begin{bmatrix} \lambda_1 & b^T \\ 0 & B \end{bmatrix} \tag{11}$$

B is used to compute its eigenvalue λ_2 and eigenvector y_2.

$$x_2 = H^{-1} \begin{bmatrix} \alpha \\ y_2 \end{bmatrix} \tag{12}$$

where

$$\alpha = \frac{b^T y_2}{\lambda_2 - \lambda_1}$$

Once the desired number of eigenvalues found by Rayleigh's quotient and Householder deflation are computed the corresponding eigenvectors must be found.

The most efficient method for finding of the eigenvectors afterwards is use shifted inverse iteration with the found eigenvalues as shift, which will make the method converge faster than computing the eigenvectors in House Holder. Then, the shift values will be equal to the relevant eigenvalue and an inverse of the shifted A matrix is required.

The code for the Householder Deflation Method with inverse iterations is presented as follows, in the subroutine **HHD**:

```
function[H_eig_values,H_eig_vector,H_timeElapsed] =  HHD(A,v,lambda,tol)
tic ;
%Get size of Matrix A
```

```
    [m,n] = size(A);
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

%Apply the Householder method in order to obtain the first
%Hermitanian Matrix, before to start the loop

        A_prev=A; %Store A value to use it later in the inverse iteration
        H = eye(n)-2*(v*v');
        A = H*A*H';
        B = A(2:m,2:n);

        HH = zeros(n,n,n); %3D array which stores the Hermitanian matrices
        HH(:,:,1) = H; %First H matrix computed and store it
        H_eig_values=zeros(n,1);
        H_eig_values(1)=lambda;
        A=B;

%Apply the Householder method from the second point until the size of
%the matrix

for i=2:m
    x0 = ones(m+1-i,1);

    %Apply the Rayleigh Quotient Iteration method to obtain eigenvalues
    %and eigenvectors
    [x,lambda,iterations] = RQI(A,tol);

    %Compute the perpendicular vector to the hyperplane with the new
    %eigenvectors x
    [v] =  vector_perpendicular(x,m,i);

    %Compute another time the Hermitanian matrix with the new perpendicular
    %vector
    H = eye(size(A))-2*(v*v');
    A = H*A*H';
    B = A(2:m+1-i,2:n+1-i);

    %Compute each H matrix and store them in the 3D array HH.
    %At each iteration, a smaller matrix is being store it, so a
    % mathematical ralation rule has been used in the indeces to be able
    % to store them properly

    HH(1:m+1-i,1:n+1-i,i) = H;

    %Store the eigenvectos and eigenvalues obtained
    %H_eig_vector(1:m,i)=x;
    H_eig_values(i)=lambda;

    %Reasign the D matrix to A to continue the loop computations
    A=B;
end

%Instead of computing the eigenvectors inside the Householder loop, an
%inverse iteration is performed afterwards once the eigenvalues are
%obtained in order to obtain the eigenvectors faster

%Initialize variables of vectors to compute later the eigenvectors
V=zeros(n,n);
H_eig_vector=ones(n,n);

for i = 1:n
    error=1;
    %While loop until convergence is achieved
    while error>10^-4
        %Compute the eigenvectors with shifting using the eigenvalues
        %previously computed
        V(:,i) = (A_prev-H_eig_values(i)* eye(n,n))\H_eig_vector(:,i);
        %Normalize the obtained eigenvectors
        V(:,i) = V(:,i)/norm(V(:,i));
        %Compute the error between the actual eigenvectors and the previous
        %ones to check if convergenced has been achieved
        error = max(abs(H_eig_vector(:,i))-abs(V(:,i)));
        H_eig_vector(:,i) = V(:,i);
    end
end

H_timeElapsed = toc;
end
```

2 QR iteration method

In this section, the QR iteration method is implemented. To do so, the QR decomposition has to be defined in order to obtain $A = Q * R$, where Q is an orthonormal matrix and R is an upper triangular. Note that for a full rank matrix A, QR decomposition is unique.

The code for the QR decomposition Method is presented as follows, in the subroutine **QR Decomposition**:

```
function[Q,R] =  QR_Decomposition(A)

%Get size of Matrix A
    [m,n] = size(A);
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

%Initialize matrices
     U = zeros(m,n);
     e = zeros(m,n);
     Q = zeros(m,n);

%Compute the first element for matrices U, e and Q
     U(:,1) = A(:,1);
     e(:,1) = U(:,1)/norm(U(:,1));
     Q(:,1) = e(:,1);

for k=2:n
    %Assign A values to matrix U
    U(:,k) = A(:,k);
    s=1;

    %In order to perform the right concatenate substraction
    %a while loop is need to substract
    %u_n = a_n-...-proj_e_(n-1)*a_n
    while (s~=k)
        t = k-s;
        %Perform concatenate substraction using own made projection
        %function
        U(:,k) = U(:,k)-projection(A(:,k),e(:,t));
        s=s+1;
    end

    %Normalize the e vectors that are each column of U
    e(:,k) = U(:,k)/norm(U(:,k));

    %Assign each e vector to a column of Q
    Q(:,k) = e(:,k);
end

%Compute R with the Q obtained and the initial matrix A
R = Q'*A;

end
```

In the script for QR Decompositioin the projection function is used in order to compute the inner product of the vectors and substract them. This function is presented in the following script:

```
function u = projection(a,e)

%This function defines the projection of vector "a" over "e",
%proj = (<a,e>/<e,e>) *e. It computes the inner product of <a,e> divided
%by the inner produt of <e,e> multiplied by the vector e.
```

```
u=((a'*e)/(e'*e))*e;
end
```

Thanks to the QR decomposition we get Q_v. Nonetheless, we still have to calculate R_v. In order to obtain it, a backward substitution is performed, in the case that the matrix is not symmetric, which allows to find all the eigenvectors. The equation used is:

$$R_{X^{k+1}} R_v = R_v * \Lambda \rightarrow V = Q_v R_v \tag{13}$$

With Λ having the eigenvalues on its diagonal and $R_{X^{k+1}}$ the matrix obtained after the QR decomposition and V the matrix containing the normalized eigenvectors.

The code for the QR Iteration method is presented as follows, in the subroutine **QRITER**:

```
function[QR_eig_vector,QR_eig_values,QR_timeElapsed] =  QRITER(A,tol)
tic;
%Get size of Matrix A
    [m,n] = size(A);
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

    %Initialize matrix Qv
    Qv = eye(m,n);
    a=0;%Break loop condition

    %Convergence criteria that checks if the norm of the subdiagonal of
    %the matrix A is less than the tolerance entered by the user.
    %As the elements of the subdiagonal are the largest ones, check
    %if they are under the tolerance is enough to know if the method has
    %converged

    %Perform QR decomposition before entering the loop
    [Q,R]=QR_Decomposition(A);

    while  (a==0)
        %Compute A with Q and R to perform another time QR_Decomposition
        A_new=R*Q;

        %Multiply Q by the matrix Qv
        Qv=Qv*Q;

         if (norm((diag(A_new,-1))) < tol)
            a=1;
         end

        %Perform QR_Decomposition
        [Q,R]=QR_Decomposition(A_new);

    end

    %In the case that A is not symmetric, the eigenvectos are not yet
    %found. So, a backwards substitution is made in order to find the right
    %eigenvectors

    if (issymmetric(A)==0)
        %Initialize  matrix Rv to solv eigenvalue problem
        Rv=zeros(n);
        %Create diagonal marix with the eigenvlues
        l = diag(R);
        %Start loop backwards the columns
        for j=n:-1:1
        %Start loop backwards the rows
            for i = n:-1:1
        %Condition which defines the diagonl elements to 1
                if (i==j)
                    Rv(i,j) = 1;
        %Condition which defines the lower diagonal elements to 0
                 elseif (j<i)
                     Rv(i,j) = 0;
                else
        %Condition which defines the upper diagonal elements to
```

```
                    Rv(i,j) = dot(R(i,i:n),Rv(i:n,j))/(l(j)-R(i,i));
                end
            end
        %Nomalize each column of the found matrix
            Rv(:,j)=Rv(:,j)/norm(Rv(:,j));
        end
        %Multiply by he Qv matrix previously found
        Qv=Qv*Rv;
    end

    %The eigenvectors are the columns of matrix
    QR_eig_vector = Qv;

    %The eigenvalues are on the diagonal of R
    QR_eig_values = diag(diag(R));
    QR_timeElapsed = toc;
end
```

3 Subspace iteration and Ritz eigenvalue problem

Considering our dynamical system specified with following differential equation:

$$Mq + K\dot{q} = 0 \tag{14}$$

Firstly, we define X_K as the initial set of vectors and calculate X_{k+1} from $KX_{k+1} = MX_K$

$$\bar{K}P = \bar{M}P\Lambda \tag{15}$$

With;

$$\bar{K} = X_{K+1}^T K X_{k+1} \tag{16}$$

$$\bar{M} = X_{K+1}^T M X_{k+1} \tag{17}$$

From the Ritz Eigenvalue problem we obtain P and Λ which contain the eigenvectors and eigenvalues of $M^{-1}K$.

In order to compute the inverse of M, a function to inverse a tridiagonal matrix has been used, using the following algorithm:

$$T = \begin{pmatrix} a_1 & b_1 & & & \\ c_1 & a_2 & b_2 & & \\ & c_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & c_{n-1} & a_n \end{pmatrix}$$

Figure 1: Tridiagonal matrix.

$$(T^{-1})_{ij} = \begin{cases} (-1)^{i+j} b_i \cdots b_{j-1} \theta_{i-1} \phi_{j+1} / \theta_n & \text{if } i < j \\ \theta_{i-1} \phi_{j+1} / \theta_n & \text{if } i = j \\ (-1)^{i+j} c_j \cdots c_{i-1} \theta_{j-1} \phi_{i+1} / \theta_n & \text{if } i > j \end{cases}$$

Figure 2: Inverse tridiagonal matrix algorithm.

where θ_i satisfy the recurrence relation

$$\theta_i = a_i\theta_{i-1} - b_{i-1}c_{i-1}\theta_{i-2} i = 2, 3...n$$

with initial conditions $\theta_0 = 1$, $\theta_1 = a_1$ and the ϕ_i satisfy

$$\phi_i = a_i\phi_{i+1} - b_ic_i\phi_{i+2} i = n-1, ...1$$

with initial conditions $\phi_{n+1} = 1$ and the $\phi_n = a_n$.

The code for the inverse tridiagonal function is presented as follows, in the subroutine **Inverse Tridiag**:

```
function[A_inv] =  Inverse_Tridiag(A)

[m,n] = size(A);
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

%This function computes the inverse of the matrix A using the principal
%minors

%Initialize an empty inverse matrix of the same size
A_inv=zeros(m,n);

a = diag(A);
b = diag(A,1);
c = diag(A,-1);

%Define vector of prinipal minors
theta = zeros(1,n+1);
%Define parameter phi to use in the recurrence formula
phi = zeros(1,n);

%Initialize initial conditions or theta and phi
theta(1) = 1;
theta(2) = a(1);
phi(n+1) = 1;
phi(n) = a(n);

%Loop to compute principal minors, where we need the first two initial
%values to compute the third one and so on so for
for i=3:n+1
    theta(i) = a(i-1)*theta(i-1)-b(i-2)*c(i-2)*theta(i-2);
end

%Backwards loop to compute parameter phi
for i=n-1:-1:1
    phi(i) = a(i)*phi(i+1)-b(i)*c(i)*phi(i+2);
end

%Main loop to construct the inverse matrix of A.
%During the loop the algoithm states to be aware of three possible
%solutions and compute them seprately.
for i=1:n
    for j=1:n
        if (i<j)
            A_inv(i,j) = (-1)^(i+j)*prod(b(i:j-1))*theta(i)*(phi(j+1)/theta(n+1));
        elseif (i==j)
            A_inv(i,j) = theta(i)*(phi(j+1)/theta(n+1));
        else
            A_inv(i,j) = (-1)^(i+j)*prod(c(j:i-1))*theta(j)*(phi(i+1)/theta(n+1));
        end
    end
end

end
```

The code for the Ritz Eigenvalue problem is presented as follows. As the matrices are tridiagonal, they can be stored in sparse format to take advantage of their shape and only compute with the nonzero entries. Storing the matrices in this format will speed up the computations. in the subroutine **RITZ**:

```
function [P,D,X] = RITZ(A,B,X0,tol)

 %Get size of Matrix A
    [m,n] = size(A);
```

```
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

%As the matrices are tridiagonal, they can be stored in sparse format to
%take adavantage of theiR shape and only compute with the nonzero entries.
%Storing the matrices in this format will speed up the computations.

A=sparse(A);
B=sparse(B);

%The matrices from the eigenvalue problem
%K_*P=M_*P*Delta are defined

[B_inv] =  Inverse_Tridiag(B);

X = B_inv*A*X0;
A_bar = X'*A*X;
B_bar = X'*B*X;

% The product matrix M^{-1}*K is computed

L=A_bar\B_bar;

%P and D are computed using QR iteration which contains
%the eigenvectors and eigenvalues respectively
[P,D,~] = QRITER(L,tol);


end
```

Then, the SSI script repeats process until achieve convergence. I created a function called **normc** to normlize each column of a matrix without doing a loop.

```
function [B] = normc(B)
%Function that normalize the column of each matrix
B = sqrt(B.^2 ./ sum(B.^2)) .* sign(B);
end
```

The code for the Subspace iteration method is presented as follows, in the subroutine **SSI**:

```
function [SSI_eig_vector,SSI_eig_values,SSI_timeElapsed] = SSI(A,B,X0,tol)
tic;
    %Get size of Matrix A
    [m,n] = size(A);
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

    a=0;%Variable used to break the loop in case of convergence
    Dprev=0;
    k=1; %Counter

    while (a==0)
        %Solve the Ritz eigenvalue problem and obtain
        %the matrix P of eigenvectors the D of eigenvalues and Xk+1
        [P,D,X] = RITZ(A,B,X0,tol);
         q = X*P;
         X0 = q;
         q = normc(q);
         if(norm(diag(D)-diag(Dprev)) < tol)
            a=1;
         end
         Dprev = D;
         k=k+1;
    end

    %The eigenvectors are stored in q and assigned to SSI_eig_vector
    SSI_eig_vector = q;

    %Normalize the eigenvectors obtained
    [SSI_eig_vector] = normc(SSI_eig_vector);

    %The eigenvalues are stored in D and assigned to SSI_eig_values
    SSI_eig_values = D;
    SSI_timeElapsed = toc;
end
```

4 Sensitivity of each method to numerical errors in A matrix

Once, all the methods are implemented, the sensitivity of each method to numerical errors in matrix A is studied. Then, a main program called **CW2** has been made to introduce the given matrices and implement the numerical methods. In addition, parameters as tolerance and numerical errors are entered by the user.

```
clear all;
close all;

%% Computational Linear Algebra CW2
%% Variables initialization to use the algebraic methods
% Given matrices by Structural Dynamics course for N=4

M=[3.2525 0.6486 0 0;
   0.6486 2.0868 0.4151 0
   0 0.4151 1.3320 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];

%As the matrices are tridiagonal, they can be stored in sparse format to
%take adavantage of theiR shape and only compute with the nonzero entries.
%Storing the matrices in this format will speed up the computations.

M=sparse(M);
K=sparse(K);

%Own function to inverse a tridiagonal matrix and solve  the system
%Ax=B
[M_inv] =  Inverse_Tridiag(M);

%Using the inverse computed function to solve the system W=M[U+FFFD]1K
W = M_inv*K;

%Get size of matrix M
[m,n]=size(W);

% Error tolerance to check convergence
tol=10^-4;

%Trial matrix used in SSI method
X0=rand(m);
%% Rayleigh Quotient Iteration and Householder deflation method

% Use the Rayleigh Equation to get the eigenvalues and eiegenvectors
[x,lambda,RQI_iterations,RQI_timeElapsed] = RQI(W,tol);

% Compute the vector perpendicular to the hyperplane used in the
% Householder method to compute the first Hermitanian matrix
[v] =  vector_perpendicular(x,m,1);

% Apply the Householder method and obtain the eigenvalues and
% eigenvectors which are stored in H_eig_values,H_eig_vector respectively

[H_eig_values,H_eig_vector,H_timeElapsed] =  HHD(W,v,lambda,tol);

% Apply Qriteration with the W matrix
[QR_eig_vector,QR_eig_values,QR_timeElapsed] =  QRITER(W,tol);

% Apply the SSI method using M, K and a trial random matrix X0
[SSI_eig_vector,SSI_eig_values,SSI_timeElapsed] =  SSI(M,K,X0,tol);
```

```
%% Sensitivity analysis for each iterative method

% Introduce some noise in the matrix W=M[U+FFFD]*K, to check how the
% eigenvalues and eigenvectors vary for each method. Instead of introducing
% the noise directly to the result matrix W, it is more realistic and
% meaningful to introduce noise in M and K where data from measures are
% stored and that error could be introduced in the process of comopute them

noise = 10^-1;
M_noise = M+noise;
K_noise = K+noise;

[M_inv_noise] =  Inverse_Tridiag(M_noise);

%Using the inverse computed function to solve the system W=M[U+FFFD]*K
W_noise = M_inv_noise*K_noise;

%Repeat each iterative method for the error matrix

% Use the Rayleigh Equation to get the eigenvalues and eiegenvectors
[x,lambda,RQI_iterations,~] = RQI(W_noise,tol);

% Compute the vector perpendicular to the hyperplane used in the
% Householder method to compute the first Hermitanian matrix
[v] =  vector_perpendicular(x,m,1);

% Apply the Householder method and obtain the eigenvalues and
% eigenvectors which are stored in H_eig_values,H_eig_vector respectively

[H_eig_values_noise,H_eig_vector_noise,~] =  HHD(W_noise,v,lambda,tol);

[QR_eig_vector_noise,QR_eig_values_noise,~] =  QRITER(W_noise,tol);

[SSI_eig_vector_noise,SSI_eig_values_noise,~] =  SSI(M_noise,K_noise,X0,tol);

Delta_HHD_eig_values=round(norm(H_eig_values-H_eig_values_noise),4);
Delta_HHD_eig_vector=round(norm(H_eig_vector-H_eig_vector_noise),4);

Delta_QRITER_eig_values=round(norm(QR_eig_values-QR_eig_values_noise),4);
Delta_QRITER_eig_vector=round(norm(QR_eig_vector-QR_eig_vector_noise),4);

Delta_SSI_eig_values=round(norm(SSI_eig_values-SSI_eig_values_noise),4);
Delta_SSI_eig_vector=round(norm(SSI_eig_vector-SSI_eig_vector_noise),4);

%% Computational cost and accuracy for each iterative method

% In this section the computational cost and time for each method is
% compared with the inbuild matlab function eig(A) which computes the
% eigenvalues and eigenvectors for a given matrix

% Compute eigenvectors and eigenvalues from Matlab function
[M_eig_vectors, M_eig_values] = eig(W);

Error_HHD_eig_values=abs(M_eig_values-H_eig_values_noise);
Error_HHD_eig_vector=abs(M_eig_vectors-H_eig_vector_noise);

Error_QRITER_eig_values=abs(M_eig_values-QR_eig_values_noise);
Error_QRITER_eig_vector=abs(M_eig_vectors-QR_eig_vector_noise);

Error_SSI_eig_values=abs(M_eig_values-SSI_eig_values_noise);
Error_SSI_eig_vector=abs(M_eig_vectors-SSI_eig_vector_noise);

%% Sensitivity matrix analysis

%Use the small matrices given to assemble a vector 3D that contains the
%derivatives of the global matri for each small matrix

[dK,dM] =  Assembly;

%Compute the sensitivity matrices and the increment in the eigenvalues
%due to an arbitrary error introduced in the Young Modulus or Density
[dL_K,dL_M,S_K,S_M] = SENS(A,dK,dM);
```

Applying a numerical error value of 10^{-1} to each matrix M and K, a matrix A with numerical errors is obtained and its eigenvalues and eigenvectors computed using each method. Then, they are compared with the eigenvalues and eigenvectors of matrix A without errors. The norm of the difference between them is showed in the parameters S_λ and S_X, respectively.

	House Holder	**QR iteration**	**Sub space iteration**
S_λ	0.8446	0.7766	0.2882
S_X	0.0777	0.0777	0.0721

Table 1: Sensitivity of each method to numerical errors in matrix A.

As it expected the methods implemented were robust enough that despite errors were introduced, they would not diverge and blow up during computations.

It has been observed that the Sub Space iteration method is the most robust to numerical errors introduced in the A matrix and House Holder with Rayleigh quotient is the most sensible one.

Then, in a practical engineering application it would be preferable to have a robust method that could obtain results very close to the theoretical ones despite that errors due to noise or measurements were introduced.

5 Accuracy and computational cost study

In this section, in order to perform an study of the accuracy and the computational cost of the three implemented methods, the eigenvectors and eigenvalues of the matrix representing our system ($A = M^{-1} * K$) are computed for each one and compared with the in-build Matlab function $eig(A)$.

Firstly, the eigenvalues and eigenvectors obtained for each method are presented in below as they are obtained directly from Matlab. Note that, the eigenvector matrix (X) is formed by 4 columns where each column is an eigenvector and the eigenvalue matrix (Λ) is a a diagonal matrix where the diagonal element are the eigenvalues. All the results are presented with 4 digit precision, in order to be able to see differences between the methods, take more digits does not have sense in engineering problems.

Matlab function eig()

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

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

Rayleigh Quotient Iteration and Householder deflation method

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

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

QR iteration method

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

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

Subspace iteration and Ritz eigenvalue problem

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

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

It is important to note that the eigenvectors obtained from our methods could be swapped or have different signs, because the eigenvectors are not unique. However, it does not affect the results which are consistent an accurate with the Matlab function as reference.

Secondly, the accuracy of the methods is computed and the results are compared with the Matlab function.

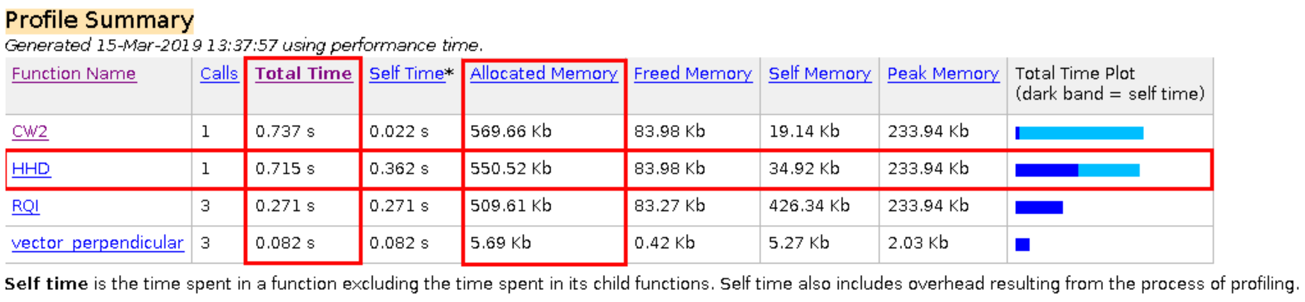
	House Holder and RQI	**QR iteration**	**Sub space iteration**
$\Delta\lambda$	0	0	0
ΔX	0	0	0

Table 2: Eigenvalues and eigenvectors error respect to Matlab function eig().

Note that, as the convergence criteria for every method is pretty small, $e = 10^{-4}$, the eigenvalues and eigenvectors already converge to the exact solution. Any difference could be find if greater number of decimals are displayed, which does not have sense in practical applications.

Finally, the computational time and storage capacity needed for every method is computed using the Matlab option **profile ('-memory','on')** and **profile viewer**, looking at the profile summary parameters, a performance analysis of the methods could be done. The perfomance summaries for every method are shown in Fig(3),(4) and (5).

Rayleigh Quotient Iteration and Householder deflation method



Profile Summary

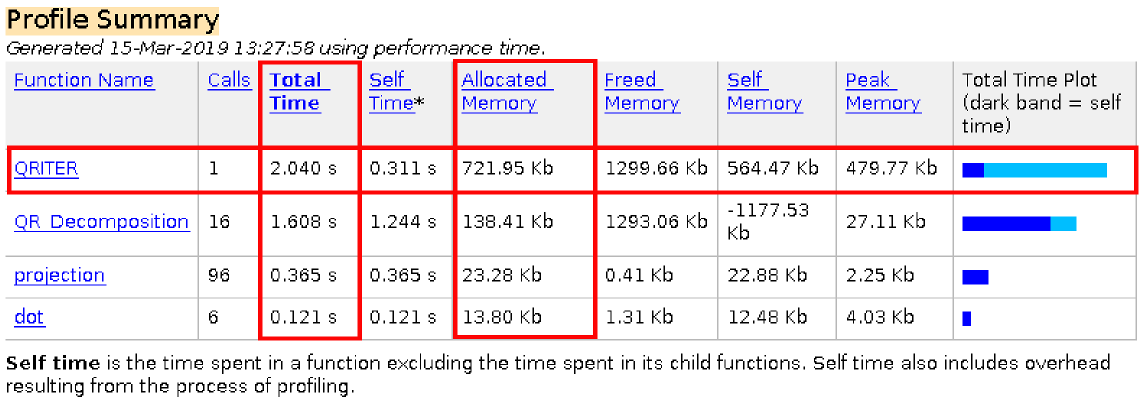
Generated 15-Mar-2019 13:37:57 using performance time.

Function Name	Calls	Total Time	Self Time*	Allocated Memory	Freed Memory	Self Memory	Peak Memory	Total Time Plot (dark band = self time)
CW2	1	0.737 s	0.022 s	569.66 Kb	83.98 Kb	19.14 Kb	233.94 Kb	▬
HHD	1	0.715 s	0.362 s	550.52 Kb	83.98 Kb	34.92 Kb	233.94 Kb	▬
RQI	3	0.271 s	0.271 s	509.61 Kb	83.27 Kb	426.34 Kb	233.94 Kb	▬
vector_perpendicular	3	0.082 s	0.082 s	5.69 Kb	0.42 Kb	5.27 Kb	2.03 Kb	▪

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 3: HHD performance summary.

QR iteration method



Profile Summary

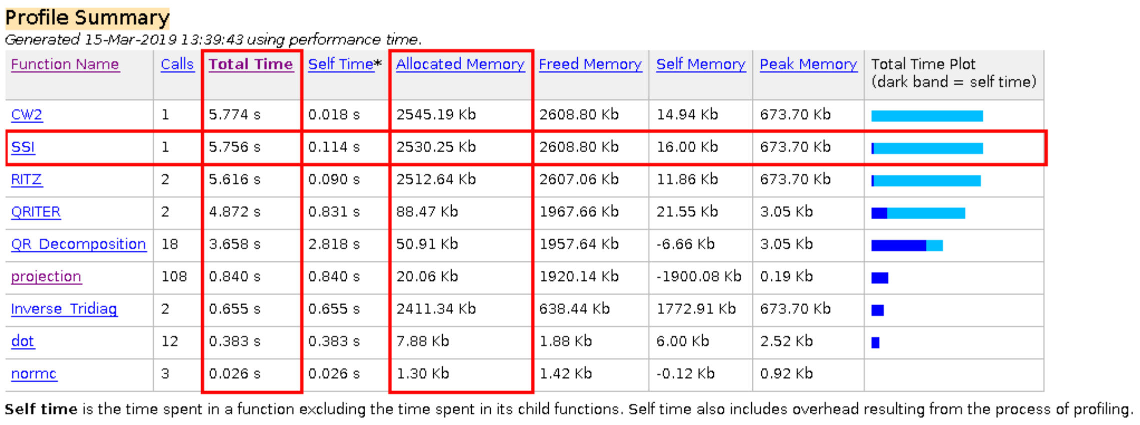
Generated 15-Mar-2019 13:27:58 using performance time.

Function Name	Calls	Total Time	Self Time*	Allocated Memory	Freed Memory	Self Memory	Peak Memory	Total Time Plot (dark band = self time)
QRITER	1	2.040 s	0.311 s	721.95 Kb	1299.66 Kb	564.47 Kb	479.77 Kb	▬
QR_Decomposition	16	1.608 s	1.244 s	138.41 Kb	1293.06 Kb	-1177.53 Kb	27.11 Kb	▬
projection	96	0.365 s	0.365 s	23.28 Kb	0.41 Kb	22.88 Kb	2.25 Kb	▬
dot	6	0.121 s	0.121 s	13.80 Kb	1.31 Kb	12.48 Kb	4.03 Kb	▬

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 4: QR performance summary.

Subspace iteration and Ritz eigenvalue problem



Profile Summary

Generated 15-Mar-2019 13:39:43 using performance time.

Function Name	Calls	Total Time	Self Time*	Allocated Memory	Freed Memory	Self Memory	Peak Memory	Total Time Plot (dark band = self time)
CW2	1	5.774 s	0.018 s	2545.19 Kb	2608.80 Kb	14.94 Kb	673.70 Kb	▬
SSI	1	5.756 s	0.114 s	2530.25 Kb	2608.80 Kb	16.00 Kb	673.70 Kb	▬
RITZ	2	5.616 s	0.090 s	2512.64 Kb	2607.06 Kb	11.86 Kb	673.70 Kb	▬
QRITER	2	4.872 s	0.831 s	88.47 Kb	1967.66 Kb	21.55 Kb	3.05 Kb	▬
QR_Decomposition	18	3.658 s	2.818 s	50.91 Kb	1957.64 Kb	-6.66 Kb	3.05 Kb	▬
projection	108	0.840 s	0.840 s	20.06 Kb	1920.14 Kb	-1900.08 Kb	0.19 Kb	▬
Inverse_Tridiag	2	0.655 s	0.655 s	2411.34 Kb	638.44 Kb	1772.91 Kb	673.70 Kb	▬
dot	12	0.383 s	0.383 s	7.88 Kb	1.88 Kb	6.00 Kb	2.52 Kb	▬
normc	3	0.026 s	0.026 s	1.30 Kb	1.42 Kb	-0.12 Kb	0.92 Kb	

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

Figure 5: SSI performance summary.

	House Holder and RQI	**QR iteration**	**Sub space iteration**
ΔT	$7.15 * 10^{-1}$s	2.040 s	5.756 s

Table 3: Computational time for every method.

As shown in Table(3), it could be observed that the fastest method is the House Holder iteration method and the slowest one is the Sub Space iteration method.

	House Holder and RQI	**QR iteration**	**Sub space iteration**
Storage capacity	550.52 Kb	721.95 Kb	2530.25 Kb

Table 4: Storage capacity for every method.

In addition, as shown in Table(4), SSI is the method with most use in storage capacity and HHD the least one.

SSI has a greater value of storage capacity and computational time, because it depends on another functions as RITZ and QRITER. Then, if the method for solving the Ritz eigenvalue problem were improved, it would be a faster method than QRITER and more robust than the other methods.

Finally, it can be concluded that the HHD method is the more advantageous in terms of time and memory. However, it is the less robust, so it is more sensible to measurement or noise errors. QR iteration is the most balanced method in terms of time and memory. Important to note that sub space iteration could be improved using a better method to solve the eigenvalue problem which would lead to a better method, so it is a better method to use in a engineering real problem.

6 Eigenvalues sensitivity analysis

In the case that some changes have to be implemented in the design process of a system, it is essential to study how the change of one or more eigenvalues will satisfy or not the restrictions.

In this section the sensitivity of the eigenvalues as a function of the Young Modulus E_{0i} and density ρ_{0i} with the ith element of the domain.

The problem considered here is:

$$KX_i = \lambda M X_i \tag{18}$$

The sensitivity matrix is based on the sensitivity of the eigenvalue λ_i, where a is E_{0i} or ρ_{0i} :

$$\frac{\delta \lambda_i}{\delta a_n} = X_i^T \left(\frac{\delta K}{\delta a} - \lambda_i \frac{\delta M}{\delta a} \right) X_i \tag{19}$$

As the stiffness matrix K only depends on E_0 and the mass matrix M on ρ, the sensitivity analysis is split in two problems separately.

$$\frac{\delta \lambda_i}{\delta E_n} = X_i^T \left(\frac{\delta K}{\delta E} \right) X_i$$

$$\frac{\delta \lambda_i}{\delta \rho_n} = X_i^T \left(-\lambda_i \frac{\delta M}{\delta \rho} \right) X_i$$

In the function Assembly, the script saves the derivatives for each local matrix K and M in a 3D array to use it later to compute the sensitivity matrix.

For example, in the stiffness K matrix, it constructed in the following way using the 4 matrices provided. For every value, there are two indices, the first index indicates to what local matrix is from and the second the position in its local matrix:

$$K = \begin{bmatrix} k_{14}+k_{21} & k_{22} & 0 & 0 \\ k_{23} & k_{24}+k_{31} & k_{32} & 0 \\ 0 & k_{33} & k_{34}+k_{41} & k_{42} \\ 0 & 0 & k_{43} & k_{44} \end{bmatrix}$$

The code for the Assembly function is presented as follows, in the subroutine **Assembly**:

```
function [dK,dM] =  Assembly

%Initialize the variables of Young Modulus and Density
Eo = 4;
rho = 3;

%Local stifness matrices to build K
k1=[5.4931 -5.4931;
    -5.4931 5.4931];
k1=k1/Eo;

k2=[4.0024 -4.0024;
    -4.0024 4.0024];
k2=k2/Eo;

k3=[2.9024 -2.9024;
    -2.9024 2.9024];
k3=k3/Eo;

k4=[2.0932 -2.0932;
    -2.0932 2.0932];
k4=k4/Eo;

%Local mass matrices to build M

m1=[2.2484 1.0087;
    1.0087 1.8052];
```

```
m1=m1/rho;

m2=[1.4473 0.6486;
    0.6486 1.1594];
m2=m2/rho;

m3=[0.9273 0.4151;
    0.4151 0.7410];
m3=m3/rho;

m4=[0.5910 0.2642;
     0.2642 0.4709];
m4=m4/rho;

%Initilalize empty matrices for K and M, and use them as tool to assembly
%the derivatives for each local matrix

K_global = zeros(4, 4);
M_global = zeros(4, 4);

%Get the size of the K global matirx
[m,n] = size(K_global);

%Create each derivative matrix of the local stifness matrices, assigning
%the right indices to the position specified in the report
dk1 = zeros(4, 4);
dk1(1,1)=K_global(1,1)+k1(2,2);

dk2 = zeros(4, 4);
dk2(1:2,1:2)=K_global(1:2, 1:2)+k2;

dk3 = zeros(4, 4);
dk3(2:3,2:3)=K_global(2:3,2:3)+k3;

dk4 = zeros(4, 4);
dk4(3:4,3:4)=K_global(3:4,3:4)+k4;

%Initialize 3D array to store each derivative matrix
dK = zeros(n,n,n);

%Assign each derivative to a position of the 3D array
dK(:,:,1)=dk1;
dK(:,:,2)=dk2;
dK(:,:,3)=dk3;
dK(:,:,4)=dk4;

%Create each derivative matrix of the local mass matrices, assigning
%the right indices to the position specified in the report
dm1 = zeros(4, 4);
dm1(1,1)=M_global(1,1)+m1(2,2);

dm2 = zeros(4, 4);
dm2(1:2,1:2)=M_global(1:2, 1:2)+m2;

dm3 = zeros(4, 4);
dm3(2:3,2:3)=M_global(2:3,2:3)+m3;

dm4 = zeros(4, 4);
dm4(3:4,3:4)=M_global(3:4,3:4)+m4;

%Initialize 3D array to store each derivative matrix
dM = zeros(n,n,n);

%Assign each derivative to a position of the 3D array
dM(:,:,1)=dm1;
dM(:,:,2)=dm2;
dM(:,:,3)=dm3;
dM(:,:,4)=dm4;

end
```

Once, the derivatives for all the matrices are obtained, the operation presented in Eq(19) is performed for the K and M matrix. Then, the values are stored in the sensitivity matrix S_K and S_M.

Afterwards, an arbitrary error in the Young Modulus (E) and density (ρ) which are ΔE and $\Delta\rho$, respectively. These deviations are introduced to analyze how the eigenvalues are affected by these deviations, following this equation:

$$\begin{bmatrix} \Delta\lambda_1 \\ . \\ . \\ \Delta\lambda_m \end{bmatrix} = \begin{bmatrix} \frac{\delta\lambda_1}{\delta a_1} & .... & \frac{\delta\lambda_1}{\delta a_n} \\ \frac{\delta\lambda_2}{\delta a_1} & .... & \frac{\delta\lambda_2}{\delta a_n} \\ \frac{\delta\lambda_3}{\delta a_1} & .... & \frac{\delta\lambda_3}{\delta a_n} \\ \frac{\delta\lambda_4}{\delta a_1} & .... & \frac{\delta\lambda_4}{\delta a_n} \end{bmatrix} * \begin{bmatrix} \Delta a_1 \\ . \\ . \\ \Delta N \end{bmatrix} \tag{20}$$

With Eq(19) the deviation for each eigenvalue could be computed and then be able to predict better how changes in the eigenvalues of our system will affect our final solution.

The code for the sensitivity analysis function is presented as follows, in the subroutine **SENS**:

```
function [dL_K,dL_M,S_K,S_M] = SENS(A,dK,dM)

[m,n] = size(A);
        if m~=n
            disp('A is not a square matrix') ;
            return;
        end

%Compute the eigenvectors and eigenvalues of our matrix to use them later
%in the sensitivity analysis

[V,D]=eig(A);

%Initialize the sensitivity matrix that or K and M that has the same size
%as A

S_K=zeros(m,n);
S_M=zeros(m,n);

%Initialize the loop to go through the whole points and construct the
%sensitivity matrix.

%The S matrix is build by S=[dL_1/da_1.....dL_n/da_n], every row is formed
%by the multiplication of X'*dK/da*X. So in the first row, we will have
%the elements resutl of multiplying the first eigenvector by the
%derivatives of every K or M matrix.

for i=1:n
    for j=1:n
        S_K(i,j) = V(:,i)'*dK(:,:,j)*V(:,i);
        S_M(i,j) = V(:,i)'*(-D(i,i)*dM(:,:,j))*V(:,i);
    end
end

%Define an error in the Young Modulus E and the rho value, assigning to
%DeltaE and DeltaRho to check the effect in the increment of eigenvalues
%DeltaL

%The problem to be solved is DL=S*DE

%Initialize vector of input errors that will multiply the
%sensitivity matrices
D_E = zeros(n,1);
D_Rho = zeros(n,1);

%Define the magintude of the error that we want to study
e_E = 10^-2;
e_Rho = 10^-2;


%Define the vector of errors for E and rho
D_E(:,1) = e_E;
D_Rho(:,1) = e_Rho;
```

```
%Compute the value of the deviation experienced in the eigenvalues
%for the stifness and mass matrix
dL_K= S_K*D_E;
dL_M= S_M*D_Rho;


end
```

The results of the sensitivity matrices for K and M are:

$$S_K = \begin{bmatrix} 0.0181 & 0.1727 & 0.5287 & 0.9140 \\ 0.2144 & 0.7447 & 0.0824 & 0.4338 \\ 0.2531 & 0.0196 & 0.3126 & 0.0861 \\ 0.0598 & 0.0462 & 0.0199 & 0.0023 \end{bmatrix} \tag{21}$$

$$S_M = \begin{bmatrix} -0.1019 & -0.3376 & -0.7370 & -1.0014 \\ -0.5380 & -0.4577 & -0.5079 & -0.4633 \\ -0.2044 & -0.3224 & -0.0548 & -0.3137 \\ -0.0079 & -0.0386 & -0.0632 & -0.0613 \end{bmatrix} \tag{22}$$

Then, to analyse the sensitivity of our eigenvalues to perturbations or changes in the Young Modulus or density, an error of 10^{-2} has been introduced in ΔE and $\Delta\rho$, the deviations are showed in the following results:

$$\Delta\lambda_K = \begin{bmatrix} 0.0163 \\ 0.0148 \\ 0.0067 \\ 0.0013 \end{bmatrix} \tag{23}$$

$$\Delta\lambda_M = \begin{bmatrix} -0.0218 \\ -0.0197 \\ -0.0090 \\ -0.0017 \end{bmatrix} \tag{24}$$

It can be observed that the introduced changes have a greater effect in the first and second eigenvalues, and the less sensible to changes is the last one.

With this sensitivity matrix, a engineering team in a design process could determine how some changes in the properties of the system could lead to variations in the eigenvalues which are key parameters to satisfy engineering design restrictions.

References

[1] L. N. L. N. Trefethen. *Numerical linear algebra.* Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[1]