SOFTWARE ASSIGNMENT
Eigenvalue Calculation

EE24BTECH11012 - Bhavanisankar G S

I. What are eigen values ?

- Almost all vectors change direction, when they are multiplied by a matrix A. Certain exceptional vectors x are in the same direction as $A\mathbf{x}$. These vectors are the **eigenvectors**. Multiply an eigenvector by A, and the vectors $A\mathbf{x}$ is a number λ times the original x.

$$\boxed{A\mathbf{x} = \lambda\mathbf{x}}$$

  The number λ is an **eigen value** of A.
- We are interested in finding the non-zero eigenvalues of a given matrix (Since, $\mathbf{x} = 0$ becomes trivial).

$$(A - \lambda I)\,\mathbf{x} = 0$$
$$\implies \boxed{|A - \lambda I| = 0}$$

II. Properties

- All the eigenvectors corresponding to a particular eigenvalue form a subspace of the vector-space. This subspace is often called the **eigenspace** of the vectorspace. If the matrix is symmetric, the eigenvectors corresponding to different eigenvalues are perpendicular.
- When a matrix gets squared, the corresponding eigenvalues get squared but the eigenvectors remain the same. This property is useful in finding higher powers of the matrix.
- Symmetric matrices have only real eigenvalues.
- Spectral radius of a matrix $A, \rho(A)$ is the supremum of the absolute values of the eigenvalues of A. The eigenvector corresponding to the largest (by absolute value) eigenvector (**Stable equilibrium distribution**) is also known as the **Perron-Frobenius eigenvector**.

III. Motivation

- **Stability Analysis** : Eigenvalues provide insight into the stability of systems. For example, in differential equations and dynamical systems, the sign and magnitude of eigenvalues determine whether the system's equilibrium points are stable or unstable.
- **Principal Component Analysis** : In data science and machine learning, eigenvalues (and their corresponding eigenvectors) are used in PCA(Principal Component Analysis) to reduce dimensionality. The eigenvalues represent the variance captured by each principal component (PC), and the largest eigenvalues correspond to the most important features in the data.
- **Quantum mechanics** : In quantum mechanics, eigenvalues correspond to observable quantities, such as energy levels in systems like atoms or molecules. Solving the SchrÃ¶dinger equation involves finding the eigenvalues of the Hamiltonian matrix.
- **Mechanical engineering** : In mechanical engineering, eigenvalues are used to determine resonant frequencies in systems like beams, bridges, or buildings. The eigenvalues correspond to the natural frequencies of vibration.
  Detour: In 1831, a suspension bridge in Broughton collapsed when the soldiers (initially marching) started walking normally over it. Had they been walking with a frequency equal to one of the eigen values corresponding to the bridge, the latter would not have collapsed.

- **Markov chains and page rank** : Eigenvalues are essential in algorithms like PageRank (used by Google) and Markov chains, where the largest eigenvalue corresponds to the stationary distribution of a system.
- **Graph theory** : In graph theory, eigenvalues of the Laplacian matrix are used to determine graph properties, like the number of connected components or to perform spectral clustering, which is used for community detection in large networks.
  Hence, providing efficient algorithms to calculate eigenvalues becomes necessary.

IV. Algorithms to calculate eigenvalues

I. JACOBI-EIGENVALUE THEOREM

- Jacobi-eigenvalue theorem is an iterative method to calculate the eigenvalues and eigenvectors of a **real symmetric matrix** by a sequence of Jacobi rotations.
- **Jacobi rotation** is an orthogonal transformation which zeroes a pair of off-diagonal elements of a matrix A.

$$A \to A' = J(p,q)^T A J(p,q) : A'_{pq} = A'_{qp} = 0$$

- The orthogonal matrix $J(p,q)$ which eliminates the element A_{pq} is called the **Jacobi rotation matrix**. It is equal to identity matrix except for the four elements with indices pp, pq, qp, qq

$$J(p,q,\theta) = \begin{pmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix}$$

- After a Jacobi rotation, $A \to A' = J^T A J$, the matrix elements of A' become

$$A'_{pq} = A'_{qp} = \left(c^2 - s^2\right) A_{pq} + sc\left(A_{pp} - A_{qq}\right) \tag{1}$$

$$A'_{pr} = A'_{rp} = cA_{pr} - sA_{qr} \text{ for } r \neq p,q \tag{2}$$

$$A'_{qr} = A'_{rq} = sA_{pr} + cA_{qr} \text{ for } r \neq p,q \tag{3}$$

$$A'_{pp} = c^2 A_{pp} - 2scA_{pq} + s^2 A_{qq} \tag{4}$$

$$A'_{qq} = s^2 A_{pp} + 2scA_{pq} + s^2 A_{qq} \tag{5}$$

$$A'_{rs} = A_{rs} \forall r \neq p,q \wedge s \neq p,q \tag{6}$$

$$\tag{7}$$

where, $s = \sin\theta, c = \cos\theta$, j is orthogonal, A and A' have the same Frobenius norm, howver we can choose θ such that $A'_{pq} = 0$, in which case A' has a large sum of squares on the diagonal.

$$A'_{pq} = (\cos 2\theta) A_{pq} + \frac{1}{2}(\sin 2\theta)\left(A_{pp} - A_{qq}\right)$$

By setting this to zero, we have

$$\tan 2\theta = \frac{2A_{pq}}{A_{qq} - A_{pp}}$$

- The method is repeated until the matrix becomes almost diagonal. The diagonal elements are then the eigenvalues of the matrix A.

- **Time complexity analysis** :

  1) CLASSICAL METHOD : with each rotation the largest of the remaining off-diagonal elements is zeroed.
  2) CYCLIC METHOD : the off-diagonal elements are zeroed in strict order, e.g., row after row.
  3) Although the classical method allows the least number of rotations, it is typically slower than the cyclic method since searching for the largest element is an $O(n^2)$ operation. The count can be reduced by keeping an additional array with indexes of the largest elements in each row. Updating this array after each rotation is only an $O(n)$ operation.
  4) A *sweep* is a sequence of Jacobi rotations applied to all non-diagonal elements. Typicaly the method converges after a small number of sweeps. The operation count is $O(n)$ for a Jacobi rotation and $O(n^3)$ for a sweep.

II. POWER ITERATION METHOD

- The power-iteration method is useful in cases where it is needed to find the largest eigenvalue and the corresponding eigenvector.
- This method can also be used to find the lease eigenvalue. If λ is the largest eigenvalue of A, then $\frac{1}{\lambda}$ is the lease eigenvalue of A^{-1}.
- **Algorithm** :
  1) Read the matrix A .
  2) Set initial vector $X_0 = \begin{pmatrix} 1 & 1 & \dots & 1 \end{pmatrix}^T$ of n components.
  3) Find the product $Y = AX_0$
  4) Find the largest element (in magnitude) of the vector Y and let it be λ.
  5) Divide all the elements of Y by λ and take it as X_1, i.e., $X_1 = \frac{Y}{\lambda}$
  6) Let $X_0 = \begin{pmatrix} x_{01} & x_{02} & \dots & x_{0n} \end{pmatrix}$ and $X_1 = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \end{pmatrix}$. If $|x_{0i} - x_{1i}| > \epsilon$ for at least i then set $X_0 = X_1$ and go to Step 3.
  7) Print λ as largest eigenvalue and corresponding eigenvector X_1 of A.
- **Time complexity**
  $O(n^2)$ per iteration, where n is the size of the matrix. Convergence may require many iterations, particularly if the largest eigenvalue is closest to the secod largest in magnitude.
- This method is best for large and sparse matrices where only the largest eigenvalue is required (as in PCA).

III. LANCZOS METHOD

- This method can be used to find the largest or a few dominant eigenvalues of a sparse symmetric matrix.
- The basic procedure uses a recursion to produce a set of vectors, referred to as **Lanczos vectors**, and scalars that form a tridiagonal matrix. The tri-diagonal matrix can then be easily solved for its eigenvalues which are used to compute a few of the eigenvalues of the original problem.
- The basic Lanczos algorith solves the standard eigenvalue problem $Ax = \lambda x$ using thee basic Lanczos recursion described below resulting in a reduced eigenvalue problem $Tq = \lambda q$, where T is a tri-diagonal matrix consisting of α s on the diagonal and β s on the off-diagonals.
- **Algorithm** :
  1) Initialization :
     Choose a starting vector v_1, where v_1 is normalized, $|v_1| = 1$.
     Set $\beta_1 = 0$ and $v_0 = 0$

2) Iteration :
   for $i = 1, 2, 3, \ldots, m$ then ;

$$\alpha_i = v_i^T w$$
$$c = w - \alpha_i H v_i$$
$$\beta_{i+1} = \sqrt{c^T H c}$$
$$v_{i+1} = \frac{c}{\beta_{i+1}}$$

– The order N of A may be 10,000 or more while order m is typically equal to twice the number of eigenvalues desired, usually less than 50.
– The eigenvalues of a tridiagonal matrix can be easily obtained using other methods like QL algorithm, a fast and efficient method for the solution of tridiagonal matrices.
– **Time complexity analysis** :
  For sparse matrices, where k is the number of iterations (often much smaller than n) the order is $O(k \cdot n^2)$
– This method is best used for sparse, symmetric matrices, where only a few eigenvalues are needed.

IV. ARNOLDI ITERATION

– The Arnoldi iteration is an efficient method for finding the eigenvalues of extremely large matrices. Instead of using standard methods, the iteration uses **Krylov subspaces** to approximate how a linear operator acts on vectors. With this approach, this method facilitates computation of eigenvalues of enourmous matrices without needing to physically create the matrix in memory.
– In linear algebra, the order-r Krylov subspace generated by an $n \times n$ matrix A and a vector b of dimension n is the linear subspace spanned by the images of b under the first r powers of A (starting from $A^0 = I$), i.e.,

$$\kappa_r(A, b) = \text{ span } \left\{b, Ab, A^2b, \ldots, A^{r-1}b\right\}$$

– A major strength of this method is that it can run on a linear operator, even without knowing the matrix representation of the operator.
– The algorithm begins by initializing a matrix H (an **upper Hessenberg matrix**) and a matrix Q which will be filled with the basis vectors of the Krylov space. It aims to find orthogonal basis of the Krylov subspace using the **Modified Gram-schmidt algorithm** .
– **Algorithm** :

1) Initialization :
   $Q \rightarrow$ empty(size(b), $k+1$)
   $H \rightarrow$ zeroes($k+1, k$)
   $Q_{:,0} \rightarrow \frac{b}{||b||}$
2) Iteration :
   for $j = 0, \ldots, k-1$ **do**

$$H_{i,j} \rightarrow Q_{:,i}^H Q_{:,j+1}$$
$$Q_{:,j+1} \rightarrow Q_{:,j+1} - H_{i,j} Q_{:,i}$$
$$H_{j+1,j} \rightarrow \left\|Q_{:,j+1}\right\|$$

3) Return :
   if $|H_{j+1,j}| < tol$ **then**

```
    **return** $H_{:j+1,:j+1}, Q_{:,:j+1}$
```

$$Q_{:,j+1} \to Q_{:,j+1}/H_{j+1,j}$$

```
    **return** $H_{:-1,:,Q}$
```

- **Time Complexity analysis**
  $O(k \cdot n^2)$ for sparse matrices, where k is the number of iterations (often much smaller than n)
- This method is best for sparse, symmetric matrices where only a few eigenvalues are needed.

V. BISECTION METHOD

- Assume that $f(x)$ is a continuous function on $[a, b]$ and that there exists a number $\epsilon \in [a, b]$ such that $f(\epsilon) = 0$. If $f(a)$ and $f(b)$ have opposite signs and $x_n = \frac{a_n+b_n}{2}$ represents the sequence of midpoints generated by the bisection method, then

$$|\epsilon - x_n| \leq \frac{b-a}{2^{n+1}} \text{ for } n = 0, 1, 2, \ldots$$

  and therefore the sequence x_n converges to the root ϵ i.e.,

$$\lim_{x\to\infty} x_n = \epsilon$$

- Bisection method is generally used for finding the eigenvalues of a symmetric tridiagonal matrix. Since, the eigenvalues are real, we can search them in the real line.

$$A = \begin{pmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ b_1 & a_2 & b_2 & \cdots & 0 \\ 0 & b_2 & a_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & 0 & b_{n-1} & a_n \end{pmatrix}$$

- **Advantages of using a symmetric matrix**

  1) The eigenvalues of $A^{(k)}$ are distinct as every $A^{(k)}$ is symmetric tridiagonal and let them be denoted by $x_1^{(k)} < x_2^{(k)} < \cdots < x_k^{(k)}$. The crucial property that makes bisection powerful is that these eigenvalues strictly interlace, satisfying the inequalities $x_j^{(k+1)} < x_j^{(k)} < x_{j+1}^{(k+1)}$ for $k = 1, 2, \ldots, m-1$ and $j = 1, 2, 3, \ldots, k-1$.
  2) The number of negative eigenvalues is equal to the number of sign changes in the sequence $1, |A^{(1)}|, |A^{(2)}|, \ldots, |A^{(k)}|$. We define the sign change to mean a transition from $+$ to $-$ or $-$ to $+$ but not from $+$ or $-$ to 0.
  3) $|A^{(k)}| = a_k|A^{(k-1)}| - b_{(k-1)}^2|A^{(k-2)}|$

- **Algorithm description**
  The interval $[a, b]$ is divided into two intervals, $[a, c]$ and $[c, b]$ such that the length of each interval is $\frac{b-a}{2}$ and $c = \frac{a+b}{2}$. If $f(c) = 0$, then c is an exact root.
  Now if $f(c) \neq 0$ and $f(a) \cdot f(c) < 0$, then the root lies in $[a, c]$ and is taken as the new interval and hence the other case. After the n_{th} iteration, we get $x_n = \frac{a_n+b_n}{2}$ and is taken as the approximate value of the root (here, the eigenvalue)
- **Time complexity analysis**
  $O(n^3)$ for dense matrices.
  $O(n)$ for symmetric tridiagonal matrices.
- This method is best used for symmetric tridiagonal matrices. Also in scenarios that require isolation of eigenvalues rather than finding all eigenvalues, as it can zoom in on specific intervals.

VI. Divide and Conquer Method

- The algorithm works recursively breaking down a problem into two or more subproblems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.
-

$$T = \begin{pmatrix} \begin{pmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ b_1 & a_2 & b_2 & \cdots & 0 \\ 0 & b_2 & a_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & b_{m-1} & a_m \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ b_m \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & \cdots & 0 & b_m \end{pmatrix} & \begin{pmatrix} a_{m+1} & b_{m+1} & 0 & \cdots & 0 \\ b_{m+1} & a_{m+2} & b_{m+2} & \cdots & 0 \\ 0 & b_{m+2} & a_{m+3} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & 0 & b_{n-1} & a_n \end{pmatrix} \end{pmatrix}$$

$$\begin{pmatrix} \begin{pmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ b_1 & a_2 & b_2 & \cdots & 0 \\ 0 & b_2 & a_3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & b_{m-1} & a_m \pm b_m \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ b_m \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & \cdots & 0 & b_m \end{pmatrix} & \begin{pmatrix} a_{m+1} \pm b_m & b_{m+1} & 0 & \cdots & 0 \\ b_{m+1} & a_{m+2} & b_{m+2} & \cdots & 0 \\ 0 & b_{m+2} & a_{m+3} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & 0 & b_{n-1} & a_n \end{pmatrix} \end{pmatrix} + \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & b_m & 0 & \cdots & 0 \\ 0 & 0 & \cdots & b_m & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

$$T = \begin{pmatrix} T_1 & \\ & T_2 \end{pmatrix} + \rho \mathbf{u}\mathbf{u}^T$$

with $\mathbf{u} = \left[\frac{\pm \mathbf{e_m}}{\mathbf{e_1}}\right]$ and $\rho = \pm b_m$ and $\mathbf{e_m}$ is a vector of length $m \approx \frac{n}{2}$ and $\mathbf{e_1}$ is a vector of length $n-m$.
- We solve the half-sized eigenvalue problems,

$$T_i = Q_i \Lambda_i Q_i^T, Q_i^T Q_i = I, i = 1, 2$$

  These two spectral decompositions can be computed by ant algorithm.
- **Time complexity analysis** :
  $O(n^2)$ for symmetric tridiagonal matrices. It merges eigenvalues using rank-one update.
- Best used for symmetric tridiagonal matrices.
- Not useful for small matrices as it involves recursion and can occupy much space .

VII. QR Algorithm

- Let $A = A_1$ be a square matrix and also let its **QR decomposition** be $Q_1 R_1$. Now let us define another matrix $A_2 = R_1 Q_1$. Next the QR factorization of A_2 be $Q_2 R_2$. Similarly we can define $A_3, A_4, A_5, \dots$
  where $A_k = Q_k R_k$ (QR factorization of A_k), and $A_{k+1} = R_k Q_k$, i.e.,

$$A_{k+1} = R_k Q_k = Q_k^* Q_k R_k Q_k = Q_k^* A_k Q_k = Q_k^{-1} A_k Q_k$$

$$A_{k+1} = (Q_1 Q_2 \dots Q_k)^* A (Q_1 Q_2 \dots Q_k)$$

If the process is continued for a long time then the matrices A_k becomes upper triangular (not always, though). In other words, $\lim_{k\to\infty} (A_k)_{ij} = 0$ for $j < i$, while the diagonal elements of the matrices A_k converge to the eigenvalues of the matrix A.

- **Theorem** :
  Suppose A be a square and also suppose that A is invertible and all its eigenvalues are distinct in modulus i.e., the algorith gives us a **Schur factorization** of A. Then, there exists (at least) one invertible matrix P such that
  $$A = P\Lambda P^{-1}$$
  with $\Lambda = diag(\lambda_1, \lambda_2, , \lambda_n)$ and $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > 0$. Suppose that the matrix P^{-1} has a LU factorization, then the sequence of matrices (A_k) is such that
  $$\lim_{k\to\infty} (A_k)_{ii} = \lambda_i, 1 \le i \le n$$
  $$\lim_{k\to\infty} (A_k)_{ij} = 0, 1 \le j < i \le n$$
  These conditions are sufficient for the matrix to converge.

A. QR decomposition using Householder reflections

- The QR decomposition factors a matrix A (of size $m \times n$) into an orthogonal (or unitary) matrix Q and an upper triangular matrix R such that :
  $$A = QR$$
- Q is orthogonal if A is real $\left(Q^T Q = I\right)$ or unitary if A is complex $(Q^* Q = I)$, where Q^* denotes the conjugate transpose of Q.
- R is an upper-triangular matrix.
- Let u and v be orthonormal vectors and let x be in the span of $\{u, v\}$ so that $x = c_1 u + c_2 v$. Consider the matrix $H = I - 2uu^T$. We can show that $Hx = x$ and hence reflection about v in the direction u can be represented as a matrix multiplication with H.
  For orthonormal u and v, the matrix $H = I - 2uu^T$ is a **Reflection** or **Householder matrix** .
- OBSERVATIONS :
  $$Q_r R_r = Q_{r-1}(Q_r R_r)R_{r-1} = Q_{r-1} A_r R_{r-1}$$
  $$Q_{r-1}^T A Q_{r-1} = A_r$$
  $$Q_r R_r = A^r$$
- It can be shown that QR iteration converges. The rate of convergence depends on ratios $\left(\frac{\lambda_j}{\lambda_i}\right)^r$ for $j \ne i$ where r is the iteration number and λ_j and λ_i are the j^{th} and i^{th} eigenvalues of A. For complex eigenvalues, we observe a slow convergence, since the eigenvalues appear as cojugate pairs of equal magnitude.
  If the magnitudes of the largest eigenvalues are not well-separated one can apply a **shifted QR** to accelerate convergence.
  The shifted QR :
  $$(A_r - k_r I) = Q_r R_r$$
  where, $A_{r+1} = R_r Q_r + k_r I$

- **Algorithm**
  Let $A_0 = A$, we iterate $i = 0$ repeat
  Choose a shift s_1

$$A_i - s_i I = Q_i R_i \text{ (QR decomposition)}$$

$$A_{i+1} = R_i Q_i + s_i I$$

$$i = i + 1$$

  until convergence.
- **Time complexity analysis**
  $O(n^3)$ or $O(k \cdot n^2)$ for square matrix of $n \times n$ and rectangular matrix of $k \times n (k > n)$.
- **Advantages**

  1) Numerical stability
  2) All eigenvalues (both real and complex) can be calculated.
  3) Versatility (both symmetric and non-symmetric matrices)
- This method is best used for Symmetric/Hermitian matrices, dense matrices and also for Numerical Software Libraries (LAPACK).

B. QR decomposition using Gram-Schmidt algorithm

- Gram-Schmidt algorithm starts with n independent vectors (usually the columns of the matrix A). It produces n orthonormal vectors (columns of Q).
- For practical reasons, having an orthonormal basis simplifies life partly because of the presence of many $\mathbf{w}_i \cdot \mathbf{w}_j$ terms that becomes zero.
- **Algorithm** :

  1) Let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ be a basis.
  2) The Gram-Schmidt process iteratively constructs from the already constructed orthonormal set $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n$ which spans a linear sub-space $\mathbf{V}_{i-1}$.
  3) The new vector $\mathbf{u}_i$ is orthogonal to the linear space $\mathbf{V}_{i-1}$. The vector is then normalized.
- **Time complexity Analysis** :
  $O(n^3)$ for a $n \times n$ matrix.
  Quadratic convergence for well-separated eigenvalues. Convergence is faster if the matrix is already nearly upper-triangular.
- Best used for scenarios where the algorithm needs to be simple.
- **Advantages** :

  * Simplicity
  * Flexibility
  * Less memory usage
- **CODE** :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <complex.h>
#include <time.h>
```

```
void matrix_mult(complex double** A, complex double** B, complex double** C, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            C[i][j] = 0.0;
            for (int k = 0; k < n; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void caseCheck(complex double **temp, complex double *eigenvalues, int n) {
        int k=0 ;
        // To handle the special case when some of the off-diagonal elements are not zero.
        // This can typically be the case when the matrix has real elements, but has complex
            eigenvalues that are conjugates of each other.
        while (k < n) {
            if ((k < n - 1) && (cabs(temp[k + 1][k]) > 1e-10)) {
               // In such cases, the eigenvalues of the 2 by 2 block matrix becomes the
                    eigenvalues of the matrix, at that position.
                double complex x1 = temp[k][k];
                double complex x2 = temp[k + 1][k];
                double complex y1 = temp[k][k + 1];
                double complex y2 = temp[k + 1][k + 1];

                // Solve quadratic equation: x^2 - (x1 + y2)x + (x1*y2 - x2*y1) = 0
                double complex b = -1.0 * (x1 + y2);
                double complex c = (x1 * y2 - x2 * y1);

                eigenvalues[k] = (-1.0*b + csqrt(b*b - 4.0*c)) / 2.0 ;
                eigenvalues[k+1]= (-1.0*b - csqrt(b*b - 4.0*c)) / 2.0 ;
                temp[k + 1][k] = 0;

                // Skip to next block
                k += 2;
            }
            else
            {
                // For single eigenvalue case (diagonal entry)
                eigenvalues[k] = temp[k][k] ;
                k++;
            }

    }
}
```

```
void gram_schmidt(complex double** A, complex double** Q, complex double** R, int n) {
    for (int j = 0; j < n; j++)
    {
        // extract vector v from the columns of A
        for (int i = 0; i < n; i++)
        {
            Q[i][j] = A[i][j];
        }

        // Gram-Schmidt process to create orthogonal vectors
        for (int k = 0; k < j; k++)
        {
            // Find an orthogonal vector to the given vector
            complex double dot_product = 0.0;
            for (int i = 0; i < n; i++)
            {
                dot_product += conj(Q[i][k]) * Q[i][j];
            }
            R[k][j] = dot_product;

            for (int i = 0; i < n; i++)
            {
                Q[i][j] -= R[k][j] * Q[i][k];
            }
        }

        // Normalize the j-th column
        R[j][j] = 0.0;
        for (int i = 0; i < n; i++)
        {
            R[j][j] += creal(Q[i][j])*creal(Q[i][j]) + cimag(Q[i][j])*cimag(Q[i][j]);
        }
        R[j][j] = sqrt(R[j][j]);

        for (int i = 0; i < n; i++)
        {
            Q[i][j] /= R[j][j];
        }
    }
}

void qr_decomposition(complex double** A, double complex* eigenvalues, int n)
{
    complex double** Q = (complex double**)malloc(n * sizeof(complex double*));
    complex double** R = (complex double**)malloc(n * sizeof(complex double*));
    complex double** temp = (complex double**)malloc(n * sizeof(complex double*));


    for (int i = 0; i < n; i++)
    {
        Q[i] = (complex double*)malloc(n * sizeof(complex double));
        R[i] = (complex double*)malloc(n * sizeof(complex double));
        temp[i] = (complex double*)malloc(n * sizeof(complex double));
    }

    for (int i = 0; i < 1000; i++)
    {
        gram_schmidt(A, Q, R, n);
        matrix_mult(R, Q, temp, n);

      for (int i = 0; i < n; i++)
      {
        for (int j = 0; j < n; j++)
        {
            A[i][j] = temp[i][j];
        }
    }
    }

    caseCheck(A, eigenvalues, n) ;
    for (int i = 0; i < n; i++) {
        free(Q[i]);
        free(R[i]);
        free(temp[i]);
    }
    free(Q);
    free(R);
    free(temp);
}


int main()
{

    int m, n ;
    printf("Enter the size of matrix: ( m n ) ") ;
    scanf("%d %d", &m, &n) ;
    if ( m != n )
    {
      // Eigenvalues can be calculated only for square matrices.
      printf("The matrix entered is not square. Eigenvalues can't be calculated\n") ;
      return 0 ;
    }
    complex double** A = (complex double**)malloc(n * sizeof(complex double*));
    for (int i = 0; i < n; i++)
    {
        A[i] = (complex double*)malloc(n * sizeof(complex double));
    }
    printf("Enter the elements row-wise ( real imag ): \n") ;
    for ( int i=0; i<n; i++)
    {
        for ( int j=0; j<n; j++)
        {
            double real, imag ;
            fscanf(stdin, "%lf %lf", &real, &imag) ;
            A[i][j] = real + imag*I ;
        }
    }
    clock_t start = clock() ;
    int countS = 0 ;
    for ( int i=0; i<n; i++)
    {
        for ( int j=i; j<n; j++)
        {
            if ( creal(A[i][j]) == creal(A[j][i]) && cimag(A[i][j]) == cimag(A[j][i]))
            {
                countS += 1 ;
            }
        }
    }
    if ( countS == n*(n+1)/2 )
    {
        // Prints if the matrix is symmetric
        printf("The matrix entered is a symmetric matrix.\n") ;
    }

    double complex* eigenvalues = (double complex*)malloc(n * sizeof(double complex));

    qr_decomposition(A, eigenvalues, n);

    printf("Eigenvalues:\n");
    for (int i = 0; i < n; i++)
    {

        printf("%.20lf + %.20lfi\n", creal(eigenvalues[i]), cimag(eigenvalues[i]));
    }
    clock_t end = clock() ;
    printf("Duration of code run: %.10f\n", (double)(end-start)/CLOCKS_PER_SEC ) ;

    // Free allocated memory
    for (int i = 0; i < n; i++)
    {
        free(A[i]);
    }
    free(A);
    free(eigenvalues);

    return 0;
}
```

V. Remarks

1) Algorithms for computing eigenvalues and eigenvectors are necessarily iterative.
2) Power iteration and its variants approximate one eigenvalue-eigenvector pair.
3) QR iteration transforms matrix into triangular form by orthogonal similarity, producing all eigenvalues and eigenvectors simultaneously.
4) Preliminary orthogonal similarity transformation to **Hessenberg** or **tridiagonal** form greatly enhances efficiency of QR iteration.
5) Krylov subspace methods (Lanczos, Arnoldi) are especially useful for computing eigenvalues of large sparse matrices.

VI. Glossary

- **Principal eigenvalue** : The eigenvalue with the largest absolute value in a matrix.
- **Eigenspace** : The vectorspace spanned by the orthogonal eigenvectors of a real, symmetric matrix.
- **Algebraic multiplicity** : The number of times an eigenvalue λ appears as root of the characteristic polynomial of a matrix.
- **Geometric multiplicity** : Dimension of the eigenspace of a matrix.
- **Spectral radius** : The largest absolute value of the eigenvalues of a matrix.
- **Rayleigh Quotient** : A formula used to approximate the eigenvalue associated with a given vector v for a matrix A. The Rayleigh Quotient is given by $R(A, v) = \frac{v^T A v}{v^T v}$
- **Schur factorization** : Factorization that expresses any square matrix A as the product of a unitary matrix U and an upper triangular matrix T

$$A = UTU^*$$

- **Hessenberg Form** : Matrix which is almost triangular.
  UPPER HESSENBERG - elements below first sub-diagonal is 0
  LOWER HESSENBERG - elements above first-sub-diagonal is 0.

VII. Bibliography

1) Websites
2) Generative AI models
3) Books