Software Assignment: Eigenvalue Calculation

AI24BTECH11031 - Shivram S

Contents

1 Abstract

The computation of eigenvalues is a fundamental task in various fields. Eigenvalues help us understand the behaviour of systems and reveal key information about the properties of linear transformations.

This report discusses various algorithms to compute the eigenvalues of a matrix, focusing on the QR algorithm, and discusses its computational complexity, numerical stability, and suitability for different kinds of matrices. Implementations of the algorithms are also presented.

2 Eigenvectors and Eigenvalues

The eigenvectors of a matrix are vectors which do not change direction when transformed by the matrix. That is, for a matrix $\mathbf{M}$, its eigenvectors are solutions to the equation

$$\mathbf{M}\mathbf{p} = \lambda\mathbf{p} \tag{1}$$

$\mathbf{p}$ is the eigenvector, and the scalar λ is called the eigenvalue. If $\mathbf{M}$ is not a square matrix, then for any vector $\mathbf{v}$, $\mathbf{M}\mathbf{v}$ will not have the same dimension as $\mathbf{v}$. Hence, a non-square matrix does not have any eigenvectors or eigenvalues.

For a square matrix, we may rewrite the above equation as

$$(\mathbf{M} - \lambda\mathbf{I})\mathbf{p} = 0 \tag{2}$$

If $\mathbf{p}$ is nonzero, then we can say that

$$|\mathbf{M} - \lambda\mathbf{I}| = 0 \tag{3}$$

For an $n \times n$ matrix, the above determinant can be expanded to an n^{th} degree polynomial in λ. The solutions of this polynomial are the eigenvalues of the matrix. The polynomial always has n roots in the complex plane, counting multiplicities. For real matrices, the complex eigenvalues always appear in conjugate pairs.

3 Algorithm Selection

There are several algorithms that can be used to calculate the eigenvalues of a matrix. Some of these algorithms are listed below.

Algorithm	Description
Power Iteration	Iteratively finds the largest eigenvalue by repeatedly multiplying the matrix with a vector.
Inverse Iteration	Finds the smallest eigenvalue by inverting the matrix and applying power iteration.
QR Algorithm	Uses QR decomposition iteratively to compute all eigenvalues of a matrix.
Lanczos Algorithm	Reduces a large sparse symmetric matrix to tridiagonal form for eigenvalue computation.
Arnoldi Iteration	Generalizes the Lanczos method for non-symmetric matrices to compute a few eigenvalues.
Jacobi Method	Rotates pairs of rows and columns to diagonalize the matrix iteratively.
Davidson Algorithm	Specialized for large sparse matrices, builds a subspace to approximate eigenvalues.

Table 1: Eigenvalue Calculation Algorithms and Their Descriptions

Methods such as Divide and Conquer or the Jacobi Method are applicable only for symmetric matrices. Other algorithms such as the

Davidson Algorithm and Arnoldi Iteration only compute a small set of eigenvalues.

If we wish to compute all the eigenvalues of a $n \times n$ matrix, the QR algorithm is the preferred due to its reliability and its ability to find all eigenvalues for an arbitrary matrix.

4 QR Algorithm

The QR Algorithm is an iterative algorithm which starts with a matrix $\mathbf{A_0} := \mathbf{A}$, and then calculates the iterative result $\mathbf{A_{k+1}}$ by first decomposing $\mathbf{A_k}$ as the product of an orthogonal matrix $\mathbf{Q}_k$ and an upper-triangular matrix $\mathbf{R}_k$ such that

$$\mathbf{A_k} = \mathbf{Q_k}\mathbf{R_k} \tag{4}$$

and then using $\mathbf{Q_k}$ and $\mathbf{R_k}$ to compute $\mathbf{A_{k+1}} = \mathbf{R_k}\mathbf{Q_k}$. This process is repeated until the result converges to an upper-triangular matrix, in which case the eigenvalues are given by the diagonal elements.

The QR algorithm works on the principle of matrix similarity. Two matrices are similar if they represent the same transformation but in different bases. That is, if two matrices $\mathbf{A}$ and $\mathbf{B}$ are similar, then there exists some invertible matrix $\mathbf{P}$ (called the change-of-basis matrix) such that

$$\mathbf{B} = \mathbf{P^{-1}}\mathbf{A}\mathbf{P} \tag{5}$$

We can show that $\mathbf{A_{k+1}}$ and $\mathbf{A_k}$ are similar matrices, and thus have the same eigenvalues.

$$\begin{aligned}
\mathbf{A_{k+1}} &= \mathbf{R_k}\mathbf{Q_k} && (6)\\
&= \mathbf{Q_k^{-1}}\mathbf{Q_k}\mathbf{R_k}\mathbf{Q_k} && (7)\\
&= \mathbf{Q_k^{-1}}\mathbf{A_k}\mathbf{Q_k} && (8)
\end{aligned}$$

$\mathbf{Q_k}$ is invertible because it is orthogonal ($\mathbf{Q_k^{-1}} = \mathbf{Q}^\top$).

The QR decomposition of any matrix $\mathbf{M}$ can be done in multiple ways.

4.1 Using the Gram-Schmidt process

The Gram-Schmidt process is used to construct an orthonormal basis from a set of vectors. We first define the projection of a vector $\mathbf{v}$ along a vector $\mathbf{u}$ as

$$\mathrm{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{u}\cdot\mathbf{v}}{\mathbf{u}\cdot\mathbf{u}}\mathbf{u} \tag{9}$$

Subtracting the projection of $\mathbf{v}$ along $\mathbf{u}$ from $\mathbf{v}$ gives us a vector which is orthogonal to $\mathbf{u}$. We can verify this by showing that their dot product is zero.

$$\begin{aligned}
(\mathbf{v} - \mathrm{proj}_{\mathbf{u}}(\mathbf{v}))\cdot\mathbf{u} &= \mathbf{v}\cdot\mathbf{u} - \mathrm{proj}_{\mathbf{u}}(\mathbf{v})\cdot\mathbf{u} && (10)\\
&= \mathbf{u}\cdot\mathbf{v} - \frac{\mathbf{u}\cdot\mathbf{v}}{\mathbf{u}\cdot\mathbf{u}}(\mathbf{u}\cdot\mathbf{u}) && (11)\\
&= \mathbf{u}\cdot\mathbf{v} - \mathbf{u}\cdot\mathbf{v} && (12)\\
&= 0 && (13)
\end{aligned}$$

Given a set of vectors $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}$, the Gram-Schmidt process gives us a set of orthogonal vectors $\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_n}$ as follows:

$$\begin{aligned}
\mathbf{u_1} &= \mathbf{v_1} && (14)\\
\mathbf{u_2} &= \mathbf{v_2} - \mathrm{proj}_{\mathbf{u_1}}(\mathbf{v_2}) && (15)\\
\mathbf{u_3} &= \mathbf{v_3} - \mathrm{proj}_{\mathbf{u_1}}(\mathbf{v_3}) - \mathrm{proj}_{\mathbf{u_2}}(\mathbf{v_3}) && (16)\\
&\vdots && (17)\\
\mathbf{u_k} &= \mathbf{v_k} - \sum_{j=1}^{k-1}\mathrm{proj}_{\mathbf{u_j}}(\mathbf{v_k}) && (18)
\end{aligned}$$

We can then normalize these vectors to obtain a set of orthonormal vectors $\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$ where

$$\mathbf{e_k} = \frac{\mathbf{u_k}}{\|\mathbf{u_k}\|} \tag{19}$$

We can express this in pseudocode as

```
begin
    for $i \leftarrow 1$ to $n$ do
        $\mathbf{u}[i] \leftarrow \mathbf{v}[i]$;
        for $j \leftarrow 1$ to $(i-1)$ do
            $\mathbf{u}[i] \leftarrow \mathbf{u}[i] - \mathrm{proj}(\mathbf{u}[j], \mathbf{v}[i])$;
        end
    end
    for $i \leftarrow 1$ to $n$ do
        $\mathbf{e}[i] = \mathbf{u}[i] / \|\mathbf{u}[i]\|$;
    end
end
```

Algorithm 1: Gram-Schmidt Process

Given an $n \times n$ matrix $\mathbf{A}$, we can construct the vectors $\mathbf{v_1}, \ldots, \mathbf{v_n}$ from the columns of the matrix. That is,

$$\mathbf{A} = \begin{pmatrix}\mathbf{v_1} & \mathbf{v_2} & \ldots & \mathbf{v_n}\end{pmatrix} \tag{20}$$

We can apply the Gram-Schmidt procedure to get the orthogonal matrix

$$\mathbf{Q} = \begin{pmatrix}\mathbf{e_1} & \mathbf{e_2} & \ldots & \mathbf{e_n}\end{pmatrix} \tag{21}$$

Since the vectors $\mathbf{e_1}, \ldots, \mathbf{e_n}$ are orthonormal, we can write each column of $\mathbf{A}$ in terms of the orthonormal basis as

$$\begin{aligned}
\mathbf{v_1} &= (\mathbf{e_1}\cdot\mathbf{v_1})\mathbf{e_1} && (22)\\
\mathbf{v_2} &= (\mathbf{e_1}\cdot\mathbf{v_2})\mathbf{e_1} + (\mathbf{e_2}\cdot\mathbf{v_2})\mathbf{e_2} && (23)\\
&\vdots && (24)\\
\mathbf{v_k} &= \sum_{i=1}^{k}(\mathbf{e_i}\cdot\mathbf{v_k})\mathbf{e_i} && (25)
\end{aligned}$$

Thus, the matrix $\mathbf{R}$ is

$$\mathbf{R} = \begin{pmatrix}
\mathbf{e_1}\cdot\mathbf{v_1} & \mathbf{e_1}\cdot\mathbf{v_2} & \ldots & \mathbf{e_1}\cdot\mathbf{v_n}\\
0 & \mathbf{e_2}\cdot\mathbf{v_2} & \ldots & \mathbf{e_2}\cdot\mathbf{v_n}\\
\vdots & \vdots & \ddots & \vdots\\
0 & 0 & \ldots & \mathbf{e_n}\cdot\mathbf{v_n}
\end{pmatrix} \tag{26}$$

Thus, the matrix $\mathbf{A}$ has been decomposed to $\mathbf{QR}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper-triangular matrix.

4.1.1 Modified Gram-Schmidt process

When performing the Gram-Schmidt process, we calculate each orthogonal vector $\mathbf{u_k}$ as

$$\mathbf{u_k} = \mathbf{v_k} - \sum_{j=1}^{k-1} \mathrm{proj}_{\mathbf{u_j}}(\mathbf{v_k}) \tag{27}$$

This method, called the Classical Gram-Schmidt, accumulates numerical errors during the projection step, due to which the final result may not be exactly orthogonal. To avoid these errors, we sequentially orthonormalize each vector instead of summing the projections:

$$\mathbf{u_k}^{(0)} = \mathbf{v_k} \tag{28}$$
$$\mathbf{u_k}^{(1)} = \mathbf{u_k}^{(0)} - \mathrm{proj}_{\mathbf{u_1}}\left(\mathbf{u_k}^{(0)}\right) \tag{29}$$
$$\vdots \tag{30}$$
$$\mathbf{u_k}^{(k)} = \mathbf{u_k}^{(k-1)} - \mathrm{proj}_{\mathbf{u_{k-1}}}\left(\mathbf{u_k}^{(k-1)}\right) \tag{31}$$
$$\tag{32}$$

The final result $\mathbf{u_k}^{(k)}$ is more numerically stable and orthogonal than the results produced by Classical Gram-Schmidt.

4.2 Using Householder transformations

A Householder transformation describes a reflection about a hyperplane containing the origin. Each hyperplane can be defined by its normal vector, $\mathbf{v}$, and the reflection of any vector $\mathbf{x}$ about this hyperplane is given by

$$\mathbf{x}' = \mathbf{x} - 2(\mathbf{v}\cdot\mathbf{x})\mathbf{v} \tag{33}$$

This transformation can be expressed in matrix form as

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}} \tag{34}$$

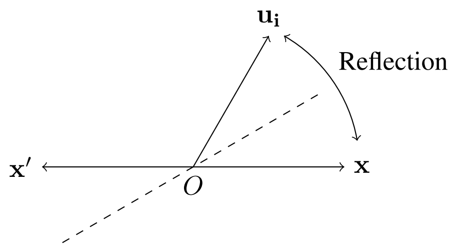
We can verify that the Householder matrix is an orthogonal matrix

$$\mathbf{H}^\top\mathbf{H} = \left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}}\right)^\top\left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}}\right) \tag{35}$$
$$= \left(\mathbf{I}^\top - 2\frac{\left(\mathbf{v}\mathbf{v}^\top\right)^\top}{\mathbf{v}^\top\mathbf{v}}\right)\left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}}\right) \tag{36}$$
$$= \left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}}\right)\left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}}\right) \tag{37}$$
$$= \mathbf{I}^2 - 4\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}} + 4\left(\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}}\right)^2 \tag{38}$$
$$= \mathbf{I} - 4\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}} + 4\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}} \tag{39}$$
$$= \mathbf{I} \tag{40}$$

To transform $\mathbf{A}$ into an upper-triangular matrix we have to compute Householder matrices that, when applied to $\mathbf{A}$, will zero out elements below the principal diagonal. For each column vector of $\mathbf{A}$, we consider the subvector of elements starting from the principal diagonal.

$$\begin{array}{c}\left(\begin{array}{|c|c|c|c|} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right) \\ \begin{array}{cccc} \mathbf{u_1} & \mathbf{u_2} & \mathbf{u_3} & \mathbf{u_4} \end{array}\end{array}$$

For each subvector $\mathbf{u_i}$, we have to construct a Householder matrix that zeroes out all components except the first component. We can do this by reflecting the vector onto the x-axis.



The normal to the hyperplane, $\mathbf{v}$ is given by

$$\mathbf{v_i} = \mathbf{u_i} - \|\mathbf{u_i}\|\,\mathbf{e_1} \tag{41}$$

The constructed Householder matrix $\mathbf{H_{sub}}$ is of order $(n-i+1)\times(n-i+1)$. To facilitate multiplication with the matrix $\mathbf{A}$, we extend the matrix as

$$\mathbf{H_i} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{H_{sub}} \end{pmatrix} \tag{42}$$

We can now write obtain the upper-triangular matrix $\mathbf{R}$:

$$\mathbf{R} = \mathbf{H_1}\mathbf{H_2}\ldots\mathbf{H_n}\mathbf{A} \tag{43}$$

and the orthogonal matrix $\mathbf{Q}$ is given by

$$\mathbf{Q} = \mathbf{H_n}^\top\mathbf{H_{n-1}}^\top\ldots\mathbf{H_1}^\top \tag{44}$$
$$= \mathbf{H_n}\mathbf{H_{n-1}}\ldots\mathbf{H_1} \tag{45}$$

It can be easily verified that $\mathbf{A} = \mathbf{QR}$

$$\mathbf{QR} = \left(\mathbf{H_n}^\top\ldots\mathbf{H_1}^\top\right)\left(\mathbf{H_1}\ldots\mathbf{H_n}\mathbf{A}\right) \tag{46}$$
$$= \mathbf{H_n}^\top\ldots\mathbf{H_1}^\top\mathbf{H_1}\ldots\mathbf{H_n}\mathbf{A} \tag{47}$$
$$= \mathbf{H_n}^\top\ldots\mathbf{H_2}^\top\mathbf{H_2}\ldots\mathbf{H_n}\mathbf{A} \tag{48}$$
$$\vdots \tag{49}$$
$$= \mathbf{A} \tag{50}$$

Thus, the matrix $\mathbf{A}$ has been decomposed to $\mathbf{QR}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper-triangular matrix.

The algorithm has been expressed in pseudocode in [Algorithm 2].

5 Generalization

The algorithms described above works for real-valued matrices which have real eigenvalues. To extend the algorithm to real-valued matrices with complex eigenvalues, and to complex matrices, slight modifications need to be made to the algorithm.

5.1 Complex eigenvalues - quasi-triangular matrices

If a real matrix has complex eigenvalues, then QR algorithm converges to an upper quasi-triangular matrix rather than an upper-triangular matrix. In an upper quasi-triangular matrix, the entries

```
begin
    $\mathbf{Q} \leftarrow \mathbf{I}_{n\times n}$;
    $\mathbf{R} \leftarrow \mathbf{A}$;
    for $i \leftarrow 1$ to $n$ do
        $\mathbf{u_i} \leftarrow \mathbf{A}[i:,i]$;
        $\mathbf{v_i} \leftarrow \mathbf{u_i} - \|\mathbf{u_i}\|\,\mathbf{e_1}$;
        $\mathbf{H_{sub}} \leftarrow \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}}$;
        $\mathbf{H_i} \leftarrow \begin{pmatrix}\mathbf{I} & \mathbf{0}\\ \mathbf{0} & \mathbf{H_{sub}}\end{pmatrix}$;
        $\mathbf{Q} \leftarrow \mathbf{Q}\mathbf{H_i}$;
        $\mathbf{R} \leftarrow \mathbf{H_i}\mathbf{R}$;
    end
end
```

Algorithm 2: QR Decomposition using Householder Transformations

below the first subdiagonal are zero. The below matrix is an example.

$$\begin{pmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 0 & 9 & 10 & 11\\ 0 & 0 & 12 & 13\end{pmatrix} \tag{51}$$

To find the eigenvalues of a quasi-triangular matrix, we move along the diagonal. If the subdiagonal element to the right of a particular diagonal element is zero, then the diagonal element is an eigenvalue of the matrix.

If the subdiagonal element is non-zero, then we consider a 2×2 block with the top-left corner at the diagonal element, and derive a pair of eigenvalues from it.

$$\begin{pmatrix}\boxed{2} & 0 & 0 & 0\\ 0 & 1 & 2 & 0\\ 0 & -2 & 1 & 0\\ 0 & 0 & 0 & \boxed{1}\end{pmatrix}$$

The above 4×4 Hessenberg matrix gives us two eigenvalues 2 and 1, and a 2×2 block $\begin{pmatrix}1 & -2\\ 2 & 1\end{pmatrix}$. For a block $\mathbf{B}$, we can write the eigenvalue equation as

$$\left|\mathbf{B} - \lambda\mathbf{I}\right| = 0 \tag{52}$$

$$\lambda^2 - \operatorname{tr}(\mathbf{B})\lambda + |\mathbf{B}| = 0 \tag{53}$$

$$\lambda = \frac{\operatorname{tr}(\mathbf{B})}{2} \pm \frac{1}{2}\sqrt{\operatorname{tr}(\mathbf{B})^2 - 4|\mathbf{B}|} \tag{54}$$

Hence, for the above matrix, the eigenvalues represented by the 2×2 block are

$$\lambda = \frac{1+1}{2} \pm \frac{1}{2}\sqrt{(1+1)^2 - 4(1\times 1 - 2\times -2)} \tag{55}$$

$$= 1 \pm 2i \tag{56}$$

The procedure can be expressed in presudocode as

5.2 Complex matrices - Schur decomposition

For complex matrices, we use the conjugate transpose $\mathbf{M^H}$ instead of the transpose $\mathbf{M}^\top$. On each iteration of the QR algorithm, we decompose $\mathbf{A}$ into a unitary matrix $\mathbf{Q}$ and an upper-triangular matrix $\mathbf{R}$. A matrix $\mathbf{M}$ is unitary if $\mathbf{M^H}\mathbf{M} = \mathbf{I}$.

```
begin
    $\Lambda \leftarrow \{\}$ ;
    $i \leftarrow 1$;
    while $i \le n$ do
        $\mathbf{B} = \mathbf{A}[i:i+2, i:i+2]$;
        if $\mathbf{B}[2,1] = 0$ then
            $\lambda = \mathbf{B}[1,1]$;
            $\Lambda \leftarrow \Lambda \cup \{\lambda\}$;
            $i \leftarrow i+1$
        else
            $a = \frac{1}{2}\operatorname{tr}(\mathbf{B})$;
            $b = \frac{1}{2}\sqrt{\operatorname{tr}(\mathbf{B})^2 - 4\left|\mathbf{B}\right|}$;
            $(\lambda_1, \lambda_2) \leftarrow (a+b, a-b)$;
            $\Lambda \leftarrow \Lambda \cup \{\lambda_1, \lambda_2\}$;
            $i \leftarrow i+2$
        end
    end
end
```

Algorithm 3: Extracting eigenvalues from a quasi-triangular matrix

The algorithm will converge to the **Schur form** of $\mathbf{A}$, which is

$$\mathbf{A} = \mathbf{Q}\mathbf{H}\mathbf{Q^H} \tag{57}$$

Here, $\mathbf{Q}$ is a Unitary matrix and $\mathbf{H}$ is an upper Hessenberg matrix that is similar to $\mathbf{A}$.

We can define the dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$ having complex entries as $\mathbf{a^H}\mathbf{b}$. Thus, we can extend the Gram-Schmidt process to vectors having complex entries by defining the projection of a vector $\mathbf{v}$ along a vector $\mathbf{u}$ as

$$\operatorname{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{u^H}\mathbf{v}}{\mathbf{u^H}\mathbf{u}}\mathbf{u} \tag{58}$$

Similarly, we can extend Householder transformations to complex-valued matrices by defining the outer product of two vectors $\mathbf{a}$ and $\mathbf{b}$ as $\mathbf{a}\mathbf{b^H}$. This allows us to compute the Householder matrix as

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v^H}}{\mathbf{v^H}\mathbf{v}} \tag{59}$$

6 Convergence

As the number of iterations tends to infinity, the QR algorithm converges to the matrix of eigenvalues $\mathbf{\Lambda}$, i.e.,

$$\lim_{k\to\infty} \mathbf{A_k} = \mathbf{\Lambda} \tag{60}$$

On each iteration of the QR algorithm we are calculating the result $\mathbf{A_{k+1}}$ as $\mathbf{A_{k+1}} = \mathbf{Q_k}^\top\mathbf{A_k}\mathbf{Q_k}$.

$\mathbf{A}$ can be decomposed as $\mathbf{A} = \mathbf{X}\mathbf{\Lambda}\mathbf{X^{-1}}$, where $\mathbf{X}$ is the matrix of eigenvectors of $\mathbf{A}$. Similarly, the iterative result $\mathbf{A_k}$ produced by the QR algorithm can be written as

$$\mathbf{A_k} = \mathbf{V}^\top\mathbf{\Lambda_k}\mathbf{V} \tag{61}$$

for some orthogonal matrix $\mathbf{V}$ where

$$\mathbf{\Lambda_k} = \mathbf{\Lambda} + \text{off-diagonal terms} \tag{62}$$

The off-diagonal terms decay as

$$|(\mathbf{A_k})_{ij}| = C\left(\frac{\lambda_i}{\lambda_j}\right)^k \tag{63}$$

Suppose the eigenvalues of $\mathbf{A}$ are ordered as $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. Then the off-diagonal terms tend to zero as k tends to infinity if $i < j$. Hence $\mathbf{A}$ converges to an upper-triangular matrix.

7 Optimizations

7.1 Shifts

As stated earlier, the off-diagonal terms of $\mathbf{A_k}$ decay as

$$|(\mathbf{A_k})_{ij}| = C\left(\frac{\lambda_i}{\lambda_j}\right)^k \tag{64}$$

If the eigenvalues are more separated from each other, then the QR algorithm converges faster. If we shift all the eigenvalues such that $\lambda_i' = \lambda_i - \mu$, then the convergence rate becomes

$$|(\mathbf{A_k})_{ij}| = C\left(\frac{\lambda_i - \mu}{\lambda_j - \mu}\right)^k \tag{65}$$

If we choose μ close to λ_j, then this shift accelerates the convergence. The shift is performed as

$$\mathbf{A_k'} = \mathbf{A_k} - \mu\mathbf{I} \tag{66}$$
$$\mathbf{Q_k R_k} = \mathbf{A_k'} \tag{67}$$
$$\mathbf{A_{k+1}} = \mathbf{R_k Q_k} + \mu\mathbf{I} \tag{68}$$

There are many ways to calculate the shift amount μ.

Method	Description
Simple Spectral Shift	Choose μ as the bottom right element of $\mathbf{A}$.
Wilkinson Shift	Choose μ as an eigenvalue of the 2×2 matrix in the bottom-right corner.
Rayleigh Shift	Choose $\mu = \frac{\mathbf{v}^\top \mathbf{A}\mathbf{v}}{\mathbf{v}^\top \mathbf{v}}$, where $\mathbf{v}$ is an approximation to an eigenvector of $\mathbf{A}$.
Inverse Iteration Shift	Solve $(\mathbf{A} - \mu\mathbf{I})\mathbf{x} = \mathbf{b}$, and use the result to improve μ.

Table 2: Various methods for calculating the shift amount

The simple spectral shift is the easiest to implement with the Gram-Schmidt process, and the Wilkinson shift is preferred for Householder transformations.

However, when the optimization was implemented, tests showed a regression in performance. Hence the optimization was disabled in the final tests.

7.2 Deflation

Once an eigenvalue has converged sufficiently, the off-diagonal terms near it become negligible. Once this happens, the matrix can be effectively split into a smaller submatrix that excludes the row and column corresponding to the converged eigenvalue.

Once an eigenvalue has converged sufficiently, we can consider it to be converged, and only iterate upon the remaining eigenvalues. We can create a smaller submatrix that excludes the row and column corresponding to the converged eigenvalue.

We can consider an eigenvalue λ_i to be converged if the elements next to it are close to zero. We can compute the new submatrix as:

$$\mathbf{A} \leftarrow \begin{pmatrix} \mathbf{A}[:i, :i] & \mathbf{A}[:i, i+1:] \\ \mathbf{A}[i+1:, :i] & \mathbf{A}[i+1:, i+1:] \end{pmatrix} \tag{69}$$

Similarly, for a 2×2 block of eigenvalues, we can check if the elements close to the block are zero, and remove both rows and columns corresponding to the block. Deflation of 2×2 blocks produces a significant boost in performance.

8 Computational Complexity

8.1 Gram-Schmidt process

Projecting a vector requires the computation of the dot products $\mathbf{u} \cdot \mathbf{u}$ and $\mathbf{u} \cdot \mathbf{v}$. Each of them requires n multiplications, and has $\mathcal{O}(n)$ time complexity.

To orthogonalize the i^{th} vector, we have to perform i such projections. So, to orthogonalize n vectors, the time complexity is

$$\sum_{i=1}^{n} i \times \mathcal{O}(n) = \frac{n(n+1)}{2} \times \mathcal{O}(n) = \mathcal{O}(n^3) \tag{70}$$

Now, we need to normalize each of the orthogonal vectors. Each vector requires n multiplications and n divisions to be orthogonalized, leading to a time complexity of $\mathcal{O}(n)$. Hence, to normalize all n vectors, the time complexity is $\mathcal{O}(n^2)$.

Now, after we determine $\mathbf{Q}$ through the Gram-Schmidt process, we need to compute $\mathbf{R}$, which requires a matrix multiplication of $\mathcal{O}(n^3)$ time complexity. Hence, the overall time complexity of one QR iteration using the Gram-Schmidt process is $\mathcal{O}(n^3)$.

In terms of memory, the implementation of the Gram-Schmidt process only stores there matrices - $\mathbf{A}$, $\mathbf{Q}$, and $\mathbf{R}$, which are reused on every iteration. Hence the overall space complexity is $kn^2 + kn^2 + kn^2 = \mathcal{O}(n^2)$.

8.2 Householder transformations

To compute the Householder vector for the $(n-k-1)^{th}$ column, we first need to compute the magnitude of the subvector, which requires k operations. The vector can then be normalized in k operations. Computing the Householder matrix requires k^2 operations.

We can multiply the Householder matrix with $\mathbf{Q}$ and $\mathbf{R}$ each in k^2 operations by only modifying the relevant submatrix. Hence the time complexity for zeroing one column is $\mathcal{O}(k^2)$.

Now, repeating this for n columns gives us a time complexity of

$$\sum_{k=1}^{n} \mathcal{O}(k^2) = \mathcal{O}(n^3) \tag{71}$$

In terms of memory, the implementation stores three matrices $\mathbf{A}$, $\mathbf{Q}$, and $\mathbf{R}$. Two additional vectors of n elements are reuired for storing the Householder vector $\mathbf{v}$ and temporary results during matrix multiplication. Hence the overall space complexity is $kn^2 + kn^2 + kn^2 + kn + kn = \mathcal{O}(n^2)$.

8.3 The QR Algorithm

The computational complexity of both the Gram-Schmidt process and Householder transformations is $\mathcal{O}(n^3)$.

Usually, the QR algorithm takes around $\mathcal{O}(n)$ iterations to isolate one eigenvalue. Hence to isolate all n eigenvalues it takes around $\mathcal{O}(n^2)$ iterations.

Using Wilkinson shifts can greatly reduce the number of iterations taken to isolate each eigenvalue. The number of iterations needed to isolate all the eigenvalues will vary between $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$.

Hence, the overall time complexity of QR algorithm will vary between $\mathcal{O}(n^4)$ and $\mathcal{O}(n^5)$. Employing deflation will not reduce the time complexity of the algorithm, but can reduce the constant factor.

9 Implementation

The QR algorithm was implemented in C using both the Gram-Schmidt process and Householder transformations. The implementation can be found in the `src` directory. The code includes a Makefile which can be used to compile the programs. To compile, run the commands

```
make all
```

`src/gram_schmidt.c` uses the Gram-Schmidt process to perform the decomposition and `src/householder.c` uses Householder transformations. `src/common.c` contains linear algebra subroutines used by both implementations.

The program in `src/generate.c` generates a random matrix of a given size. Once compiled, it can be run using the command

```
./bin/generate ⟨size⟩ ⟨output file⟩
```

The matrix will be written to the specified file. The QR algorithm implementations can be executed using the commands:

```
./bin/gram_schmidt ⟨size⟩ ⟨input file⟩
./bin/householder ⟨size⟩ ⟨input file⟩
```

The commands will print the computed eigenvalues to standard output.

9.1 Implementation Details

This section provides a short description of various parts of the implementation

`common.c` provides common linear algebra subroutines used by both the implementations.

- `common.c` first defines an adjustable `TOLERANCE`, which can be changed to tweak the precision of the algorithm.
- The `mat_read` function is used to read a matrix from a file. Such a matrix can be generated by the `./bin/generate` command or the `mat_print` function.
- `mat_mul` is used to multiply two $n \times n$ matrices.
- `mat_trans` computes the transpose of a matrix.
- `mat_conjt` computes the conjugate transpose of a matrix.
- `mat_print` prints an $n \times n$ matrix.
- `normalize` normalizes a vector.
- `diagonalize` extracts the eigenvalues of a quasi-triangular matrix and stores them in its diagonal elements.

`deflate.c` implements functions related to the deflation of matrices.

- `get_real_deflatable` returns the index (row and column) of a real eigenvalue which can be deflated.
- `get_complex_deflatable` returns the index (row and column) of a 2×2 block of eigenvalues which can be deflated.
- `remove_row_col` removes a specified row and column from a matrix.
- `deflate` attempts to deflate a specified matrix, and returns the size of the matrix after deflation.

`gram_schmidt.c` and `householder.c` share some functions:

- `converged` checks if a matrix has converged, i.e., all the entries below the first subdiagonal are close to zero.
- `qr_reduce` repeatedly applies QR decomposition to a matrix and also deflates it.
- The `main` function handles command-line arguments, reads the matrix from a file, executes the QR algorithm, and prints the eigenvalues.

In `gram_schmidt.c`, QR decomposition is performed using the `gram_schmidt_qr` function. This function calls `gram_schmidt`, which orthogonalizes $\mathbf{A}$ to get $\mathbf{Q}$, then computes $\mathbf{R}$ as $\mathbf{R} = \mathbf{Q}^\top \mathbf{A}$.

In `householder.c`, QR decomposition performed using the `householder_qr` function, which calls `householder_vector` to compute the vector $\mathbf{v}$, then updates $\mathbf{Q}$ and $\mathbf{R}$ by multiplication with the Householder matrix.

9.2 Performance Optimizations

The performance of the Gram-Schmidt process and Householder transformations were improved through vectorization and multithreading.

For vectorization, the `-mavx` compiler option was used. I also experimented with manually writing AVX intrinsics, but the resulting program was much slower than the compiler-generated one.

Multithreading was done using the OpenMP API to provide parallelism wherever possible. However, due to the added cost of synchronization, it produced a performance gain only for large matrices.

Additional optimizations were made in the Gram-Schmidt implementation. Essentially, each vector was normalized before the projections of other vectors on it were computed. This removed the need to normalize the vector later, and also removed the need to recompute the magnitude of the vector after each projection by simplifying the projection formula to

$$\operatorname{proj}_{\mathbf{u}}(\mathbf{v}) = (\mathbf{v} \cdot \mathbf{u})\mathbf{u} \tag{72}$$

9.3 Performance Analysis

The performance of both the programs was tested for $n \times n$ matrices of sizes ranging from $n = 5$ to $n = 100$. The script in `compare.sh` was used to collect performance data in `data/gram_schmidt.dat` and `data/householder.dat`.

Both programs were tested on the same matrices. Each line contains the size of the matrix followed by four readings taken for that size. The data was plotted using `plot.py`.

The performance of the Gram-Schmidt implementation varies a lot based on the matrix. For some matrices it takes 20-30 times as much time as other matrices. Due to this inconsistency, the Householder implementation is preferred.

The performance of the Householder implementation was measured for matrices of size up to $n = 250$. The data was collected using `householder250.sh` into `data/householder250.dat`, after which `plot.py` was used to plot the data.

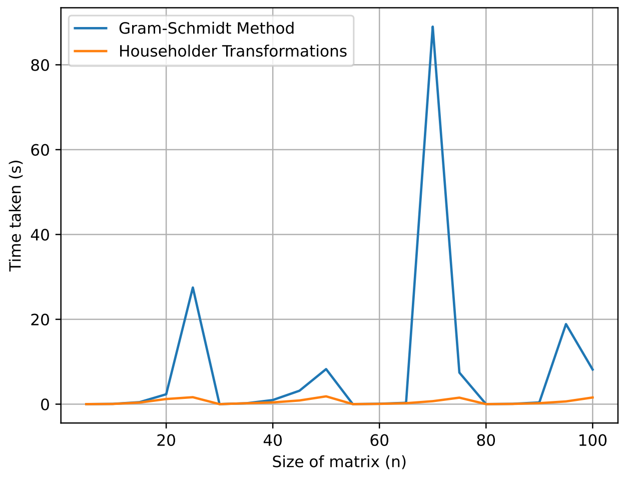


Figure 1: Performance comparison between Gram-Schmidt and Householder implementations

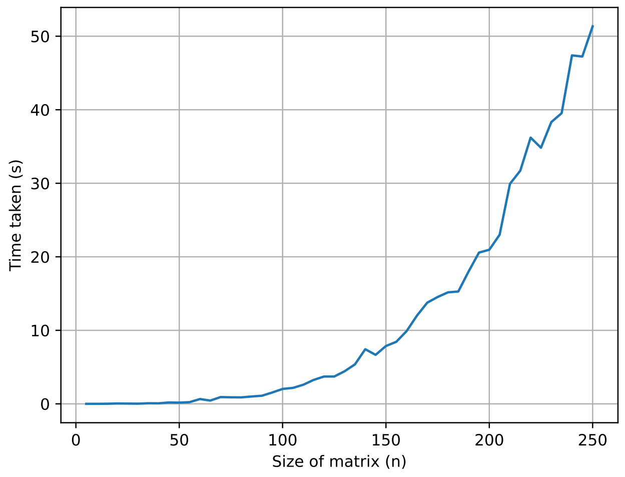


Figure 2: Performance of the Householder implementation for large matrices

10 Conclusion

Thus, we have implemented an algorithm to compute all the eigenvalues of an arbitrary $n \times n$ matrix using the Gram-Schmidt process and Householder transformations. Of these two, Householder transformations were found to be more reliable. We also optimized our algorithm using techniques such as deflation, leading to a significant performance increase.

In the future, further optimization methods such as Hessenberg reductions and Givens rotations could be explored, as they might be a faster and more accurate way to decompose the matrix.

Overall, working on this project provided valuable insights into various aspects of linear algebra and matrix theory, and introduced me to the inner working of scientific computing libraries such as NumPy, LAPACK and eigen.h.

References

[1] Peter Arbenz. *The QR Algorithm*. URL: `https://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter4.pdf`.

[2] Greg Fass. *Convergence of the QR Algorithm*. URL: `https://sites.math.rutgers.edu/~falk/math574/lecture9.pdf`.

[3] Greg Fass. *The QR Algorithm*. URL: `https://math.iit.edu/~fass/477577_Chapter_11.pdf`.

[4] Gilbert Strang. “Linear Algebra and its Applications, Fourth Edition”. In: 2022. Chap. 7.3: Computation of Eigenvalues.

[5] Hannes Thiel. *The QR Algorithm and other methods to compute the eigenvalues of complex matrices*. URL: `http://hannesthiel.org/wp-content/OtherWriting/QR.pdf`.