भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

# Eigenvalue Calculation


**AUTHOR**

Ronit Ranjan
AI24BTECH11028


2024-11-18

# Contents

# 1 Various Algorithms

The thing with calculating eigenvalues of a matrix is that there is no algorithm which can calculate the eigenvalues of all matrices(symmetric and non symmetric) and its very time efficient as well. So a few algorithms i considered after looking up like more than 10 different algorithms are:

## 1.1 Divide-and-Conquer Algorithm

**Pros:**

- It is efficient for large and dense matrices.
- Its parallelizable which makes its suitable fro high-performance computing.
- Can compute all eigenvalues and eigenvectors

**Cons:**

- Can compute eigenvalues of symmetric matrices only.
- Complex to implement
- Less efficient for smaller matrices

## 1.2 Jacobi Algorithm

**Pros:**

- Its simple and easy to implement.
- Produces accurate results for symmetric matrices
- Focuses on rotations instead of multiplication which is computationally cheaper

**Cons:**

- It is computationally expensive for large matrices. It follows $O(n^4)$ if all eigenvalues have to be computed.
- It does not work directly for genral matrices

## 1.3 QR Algorithm

**Pros:**

- Relatively starightforward.
- Works on both symmetric and non symmetric matrices
- Gives decent accuracy for Small matrices

**Cons:**

- Computationally heavy for big matrices(Time complexity = $O(n^3)$).
- Though it works for all matrices, but for non symmetric matrices it converges very slowly.
- Can have poor accuracy for matrices that have closed spaced eigenvalues

## 1.4 QR with Hessenberg Reduction

**Pros:**

- It is better than QR algorithm because it follows time complexity of $O(n^2)$
- Works for general matrices
- Can handle both real and complex eigenvalues
- Eigenvectors can also be calculated.

**Cons:**

- The initial transformation to Hessenberg form has a computational cost of $O(n^3)$ which might dominate for smaller matrices.
- Convergence rate might be slow for matrices with closely spaced eigenvalues.
- For small matrices it might not be that efficient.

## 1.5 QR with householder reflections:

**Pros:**

- Efficient for Dense matrices
- Low memory usage

**Cons:**

- It is not very optimized for eigenvalue problems.
- For small matrices other algorithms may outperform because of overhead in Householder.

# 2 Implementing the Algorithm

## 2.1 Attempt 1

Spoiler Alert: I failed in implementing this.
So the idea was to implement 4 different Algorithm for 4 types of matrices i.e Symmetric/Non Symmetric and Small/Large. So I choose the below Algorithm for each type of matrix.

| | Small | Large |
|---|---|---|
| Symmetric | Jacobi | Divide and Conquer |
| Non-Symmetric | QR | QR with hessenberg |

### 2.1.1 But Why?

The thing is no algorithm is completely perfectly in itself. The one that is best among them is QR with hessenberg because it could give all eigenvalues even for medium/large dense matrices, but it has a problem of initial overhead which makes it not that efficient for small matrices. We have Jacobi and QR for small matrices which was much efficient than QR with hessenberg for small matrices. Also I thought it would be fun to implement 4 different algorithm in 4 different C programs and there will be one master program which will take the input and will run the most adequate algorithm based on the user's input.

### 2.1.2 Implementation

So I wrote all the 4 algorithms in C. The code source is below:

1. Jacobi.
2. QR.
3. Divide and Conquer.
4. QR with hessenberg.

Then I tried making the master C program which will call the others programs. I tried using "system" function in C. So what I planned was I will scan the input, print the input into a .txt file, the suitable program will run based on the input and that program will read the input from the .txt file and will calculate the eigenvalues.

### 2.1.3 So what is the problem?

Skill issue and Time constraint. I tried making the program, it compiled successfully but i think the value didn't get passed to the respective programs properly due to some pointer issue or something. I tried fixing the issue, but because the deadline was close and I started losing hope and eventually dropped this idea.

## 2.2 Attempt 2

So after failing in the above task, I attempted this approach using only one Algorithm.

### 2.2.1 Algorithm

So I am gonna try implementing QR with Hessenberg. Various reasons choosing this algorithm was:

- Works for all type of matrices.
- Can handle both real and complex eigenvalues.
- Ideal for dense matrices when computational efficiency is very critical.
- Faster iteration because due to Hessenberg form.
- Lower memory usage than QR with Householder Reflection.

Code: QR with Hessenberg Reduction.

## 2.3 How does the code work?

### 2.3.1 Libraries

- `stdio.h` and `stdlib.h` for input/output.
- `math.h` for mathematical function.
- `complex.h` for calculations in calculating complex eigenvalues.
- `time.h` to measure time to calculate eigenvalues.

### 2.3.2 Functions

- `allocate matrix` to allocate memory for a 2D matrix(dynamically).
- `freeMatrix` to free the memory for the matrix when it is no longer needed.
- `hessenbergReduction` to reduce the matrix into Hessenberg form
- `qrDecomposition` to orthogonalize H into Q and R using Householder reflectors.
- `qrIteration` Updates H with $R * Q$(to ensure eigenvalues converge along the diagonal). If the subdiagonal entries are below tolerance, then the iterations stop. If the subdiagonal entries are 0, diagonal elements are real eigenvalues. If subdiagonal entries persist, it extracts complex eigenvalues from $2 \times 2$ blocks.
- `main function` Takes the size of matrix $(n)$ and the element of the matrix rowise and stores it in array $A$. Finally after using `hessenbergReduction` and `qrIteration`, stores the final eigenvalues in an matrix `eigenvalues`.

### 2.3.3 Matrix behind it

The QR algorithm is an iterative method that applies the QR decomposition of a matrix $A$ and forms a sequence of matrices $A_k$ defined as:

$$A_k = Q_k R_k, \quad A_{k+1} = R_k Q_k,$$

where $Q_k$ is an orthogonal matrix and $R_k$ is an upper triangular matrix obtained from the QR decomposition of $A_k$.

As $k \to \infty$, the sequence $A_k$ converges to an upper triangular matrix whose diagonal entries are the eigenvalues of $A$.

**Hessenberg Reduction** The efficiency of the QR algorithm can be greatly improved by reducing the input matrix $A$ to an upper Hessenberg form before applying the QR steps. A Hessenberg matrix $H$ is a square matrix with zero entries below the first subdiagonal:

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1n} \\ h_{21} & h_{22} & h_{23} & \cdots & h_{2n} \\ 0 & h_{32} & h_{33} & \cdots & h_{3n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{n-1,n} & h_{nn} \end{bmatrix}.$$

This transformation is achieved using orthogonal similarity transformations:

$$H = Q^T A Q,$$

where $Q$ is a product of Householder reflectors.

**Householder Reduction to Hessenberg form** Given a matrix $A$, the Hessenberg reduction is performed iteratively. At each step, we eliminate elements below the first subdiagonal using Householder reflectors. Let $v$ be a vector orthogonal to the subdiagonal elements, and define:

$$P = I - 2\frac{vv^T}{v^Tv}.$$

The similarity transformation $A \mapsto PAP^T$ results in a Hessenberg matrix. At each iteration of the QR algorithm, we decompose the current matrix $H_k$ into $Q_kR_k$, where:

- $Q_k$: Orthogonal matrix (product of Givens rotations or Householder reflectors).
- $R_k$: Upper triangular matrix.

The updated matrix is then computed as:

$$H_{k+1} = R_kQ_k.$$

The QR algorithm, applied iteratively to a Hessenberg matrix, converges to a quasi-upper triangular matrix. For real matrices, this means:

- Real eigenvalues appear on the diagonal.
- Complex conjugate eigenvalue pairs appear in 2x2 blocks.

## 2.4 Output

The program gives 3 things as output

- **Number of Iterations**: Number of QR iterations performed.
- **Time**: Time taken to calculate the eigenvalues.
- **Eigenvalues** in form $a + \iota b$

### 2.4.1 Example

- **For a 5x5 matrix:** The data for matrix is 5x5 matrix
- **For a 100x100 matrix:** The data for matrix is 100x100 MATRIX

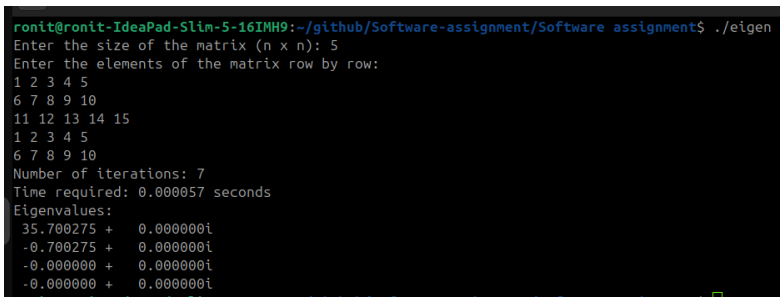


Figure 1: Output for 5x5 matrix

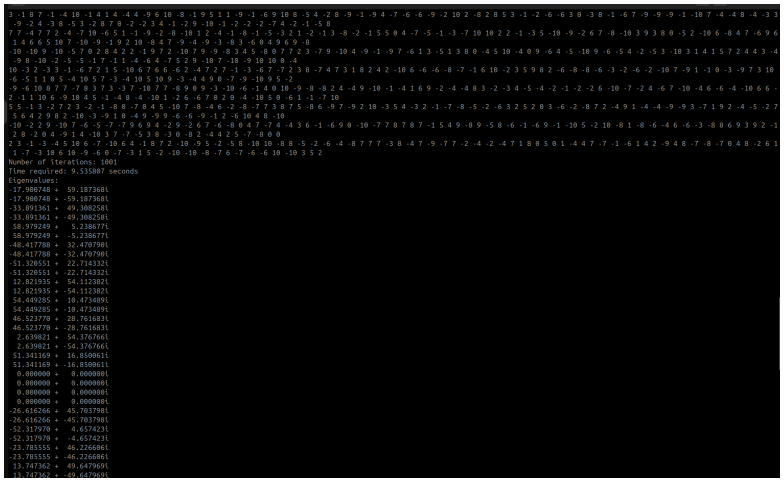

Figure 2: Output for 100x100 matrix