Eigenvalue Calculation Using QR Algorithm with Wilkinson's Shift

Arnav Mahishi-ee24btech11006

1 Introduction

This report informs on the the implementation and analysis of the QR Algorithm for computing eigenvalues of matrices, specifically using Wilkinson's shift strategy. The QR Algorithm is preferred because it can handle non-symmetric matrices and complex values giving us efficient computation of the eigenvalues. The implementation outlined in this report includes a detailed description of Wilkinson's shift, which is employed to accelerate convergence by adjusting the diagonal elements of the matrix.

2 QR Algorithm with Wilkinson's Shift

The QR Algorithm with Wilkinson's shift modifies the traditional QR iteration to improve the convergence rate by incorporating a shift that is carefully computed. This shift is based on the bottom right 2×2 submatrix of the matrix A and is given by:

$$\text{shift} = A_{(n)(n)} - \operatorname{sgn}(d)\sqrt{d^2 + A_{(n)(n-1)}A_{(n-1)(n)}}, \tag{1}$$

where

$$d = \frac{A_{(n-1)(n-1)} - A_{(n)(n)}}{2}. \tag{2}$$

N is the size of the matrix.This strategy allows the algorithm to more rapidly converge to the eigenvalues of the matrix.

3 QR Algorithm Steps

The algorithm proceeds as follows:

3.1 Overview

This implementation computes the eigenvalues of a square matrix using the QR algorithm with Wilkinson's shift for improved convergence. The algorithm decomposes the matrix iteratively into orthogonal (Q) and upper triangular (R) matrices while shifting and updating the matrix to accelerate convergence.

3.2 Steps of the Algorithm

1) **Input:**
   - A square matrix A of size $n \times n$.
   - Maximum number of iterations, Max_iterations (default: 1000).
   - Convergence tolerance, Tolerance (default: 10^{-10}).
2) **Initialization:**
   - Set $A_0 = A$, the input matrix.
   - Initialize the iteration counter, iter = 0.
3) **Iterative Process:** Repeat the following steps until the matrix converges (sub-diagonal elements are below Tolerance) or iter = Max_iterations:
   a) **Compute Wilkinson's Shift:** For the bottom-right 2×2 submatrix:

   $$B = \begin{bmatrix} a_{n-2,n-2} & a_{n-2,n-1} \\ a_{n-1,n-2} & a_{n-1,n-1} \end{bmatrix},$$

   calculate:

   $$\text{shift} = A_{(n)(n)} - \operatorname{sgn}(d)\sqrt{d^2 + A_{(n)(n-1)}A_{(n-1)(n)}}, \tag{3}$$

   where

   $$d = \frac{A_{(n-1)(n-1)} - A_{(n)(n)}}{2}. \tag{4}$$

   where sign(d) is 1 if $d \geq 0$ and -1 otherwise and N is the size of the matrix
   b) **Shift the Matrix:** Subtract μ from the diagonal elements of A_k:

   $$\hat{A}_k = A_k - \mu I.$$

   c) **QR Decomposition:** Decompose $\hat{A}_k$ into Q_k (orthogonal) and R_k (upper triangular) using Gram-Schmidt:

   $$\hat{A}_k = Q_k R_k.$$

   d) **Update the Matrix:** Compute the next matrix:

   $$A_{k+1} = R_k Q_k + \mu I.$$

   e) **Check Convergence:** If all sub-diagonal elements $\hat{A}_k[i+1, i]$ are smaller than Tolerance, break the iteration loop.
4) **Extract Eigenvalues:** For the resulting matrix, if any 2×2 submatrix remains on the diagonal, compute its eigenvalues using the quadratic equation:

   $$\lambda^2 + b\lambda + c = 0,$$

   where $b = -(a_{n-2,n-2} + a_{n-1,n-1})$ and $c = a_{n-2,n-2}a_{n-1,n-1} - a_{n-1,n-2}a_{n-2,n-1}$. For diagonal elements, take $\lambda = a_{i,i}$.
5) **Output:** The eigenvalues of the matrix A.

3.3 Time Complexity

The QR algorithm with Wilkinson's shift involves several computational steps. Below is the breakdown of the time complexity for each component:

1) **Matrix Multiplication (`matmul` function):** The function multiplies two $n \times n$ matrices. For each element in the result matrix, it computes a dot product of two vectors of length n. This requires $O(n^3)$ operations.

$$\text{Time Complexity: } O(n^3)$$

2) **QR Decomposition (`gramschmidt` function):** The Gram-Schmidt process orthogonalizes the columns of an $n \times n$ matrix:
   - For each column j, it computes the projection against all previous columns, which involves $O(n^2)$ operations.
   - This is repeated for all n columns, leading to $O(n^3)$ operations in total.

   Time Complexity: $O\left(n^3\right)$
3) **Wilkinson's Shift:** Computing the shift involves solving a 2×2 eigenvalue problem, which is a constant-time operation, $O(1)$. This operation is repeated once per iteration. Time Complexity per Iteration: O(1)
4) **Iterative Updates:** Each iteration of the QR algorithm involves:
   - Subtracting the shift from the diagonal elements, $O\left(n^4\right)$.
   - Performing the QR decomposition, $O(n^3)$.
   - Matrix multiplication to update A_k, $O\left(n^3\right)$

   Thus, the cost per iteration is dominated by $O\left(n^3\right)$
   Time Complexity per Iteration: $O\left(n^3\right)$
5) **Convergence:** The algorithm typically converges in approximately $O(n)$ iterations for most matrices. Total Time Complexity: $O\left(n^4\right)$

The overall time complexity of the QR algorithm with Wilkinson's shift is dominated by the iterative QR decomposition and matrix multiplication steps, resulting in:

$$\text{Total Time Complexity: } O(n^4)$$

3.4 Memory Complexity

1) **Matrix Storage:**
   - The algorithm operates on an $n \times n$ matrix. The memory required to store the matrix is $O(n^2)$.
2) **Intermediate Matrices:**
   - During the computation, additional matrices such as Q, R, and intermediate results of matrix operations are created. Each matrix has a memory requirement of $O(n^2)$.
3) **Extra Data Structures:**
   - Temporary variables such as vectors (v), norms, and shifts are computed, but their memory usage is $O(n)$, which is negligible compared to the matrices.

Total Memory Complexity:

- The dominant factor is the storage and manipulation of $O(n^2)$ matrices.

Overall Memory Complexity is: $O(n^2)$.

3.5 Convergence Analysis

The QR algorithm with Wilkinson's shift converges rapidly to the eigenvalues of a matrix. Without shifts, the standard QR algorithm converges linearly, which can be slow, especially for matrices with eigenvalues that are close to each other. However, by applying Wilkinson's shift, the convergence is significantly accelerated.

Wilkinson's shift is computed as:

$$\sigma = A_{n,n} - \text{sign}(d)\sqrt{d^2 + A_{n-1,n}A_{n,n-1}}$$

where $d = \frac{A_{n-1,n-1} - A_{n,n}}{2}$. This shift focuses on the bottom-right 2x2 block of the matrix, which accelerates the process of diagonalizing the matrix by reducing off-diagonal elements more effectively.

The convergence of the QR algorithm with Wilkinson's shift is quadratic, meaning that the error in the eigenvalue approximation decreases exponentially as the iterations progress. Specifically, if ϵ_k represents the error at the k-th iteration, the error satisfies:

$$\epsilon_{k+1} \approx C \cdot \epsilon_k^2$$

for some constant C. This quadratic convergence leads to fast approximation of the eigenvalues, with the number of iterations required to reach a desired tolerance being relatively small.

Overall, the QR algorithm with Wilkinson's shift exhibits efficient convergence, especially for larger matrices, making it a preferred method for eigenvalue computation.

4 Comparison of Eigenvalue Computation Methods

The following list provides a comparison of various eigenvalue computation methods, including their pros, cons, and time complexity:

- **QR Algorithm with Wilkinson's Shift:**
  - **Pros:** Handles non-symmetric matrices well. Converges rapidly for large matrices. Efficient for complex and sparse matrices.
  - **Cons:** Computationally expensive ($O(n^4)$). Not ideal for very large n.
  - **Time Complexity:** $O(n^4)$.
- **Jacobi's Method:**
  - **Pros:** Simple and easy to implement. Works well for symmetric matrices.
  - **Cons:** Slow for large matrices, especially for dense matrices.
  - **Time Complexity:** $O(n^3)$.
- **Power Iteration:**
  - **Pros:** Simple and fast for finding the dominant eigenvalue.
  - **Cons:** Can only find the dominant eigenvalue. Convergence is slow.
  - **Time Complexity:** $O(n^2)$ per iteration.

- **Householder Method:**
  - **Pros:** Efficient for orthogonalization. Fast for large matrices.
  - **Cons:** Requires more memory and may have numerical stability issues for very large matrices.
  - **Time Complexity:** $O(n^3)$.
- **Givens Rotation:**
  - **Pros:** Effective for sparse matrices. Good for solving least squares problems.
  - **Cons:** Relatively slower for dense matrices. Not as robust as Householder.
  - **Time Complexity:** $O(n^3)$.

4.1 Why QR Algorithm with Wilkinson's Shift is the Best

Among all the methods listed, the **QR Algorithm with Wilkinson's Shift** stands out as the best choice for eigenvalue computation in general-purpose applications, especially when dealing with non-symmetric or complex matrices.

- **Convergence Speed:** The Wilkinson's shift significantly accelerates the convergence of the QR Algorithm, especially for matrices with a large number of eigenvalues. This results in faster computation compared to simpler methods like Power Iteration or Jacobi's method.
- **Versatility:** Unlike Power Iteration, which only computes the dominant eigenvalue, the QR Algorithm computes all eigenvalues, making it a more complete solution.
- **Handling of Complex Matrices:** The QR Algorithm is robust for both real and complex matrices, whereas methods like Jacobi are limited to symmetric matrices.
- **Numerical Stability:** The Wilkinson's shift, by focusing on the bottom-right 2x2 submatrix, stabilizes the QR process and reduces errors in numerical calculations, especially in the presence of near-degenerate eigenvalues.
- **Applicability to Large Matrices:** Although computationally expensive ($O(n^4)$), the QR Algorithm with Wilkinson's shift remains one of the most reliable methods for large matrices, especially in scientific computing and numerical simulations where accurate eigenvalue computation is crucial.

Thus, despite its higher computational cost, the QR Algorithm with Wilkinson's Shift is the best choice for most applications where accuracy and reliability are paramount, especially when dealing with non-symmetric and complex matrices.

5 Implementation of QR Algorithm with Wilkinson's Shift in Python

Below is the Python code implementing the QR algorithm with Wilkinson's shift, which allows for more rapid convergence:

Listing 1: QR Algorithm with Wilkinson's Shift in Python

```
import cmath
import time
def matmul(A, B):
    n = len(A)
    C = [[0] * n for _ in range(n)] # Create matrix C nxn
    for i in range(n):
        for j in range(n):
            C[i][j] = sum(A[i][k] * B[k][j] for k in range(n))
    return C

def gramschmidt(A):
    n = len(A)
    Q = [[0] * n for _ in range(n)] # Initialize Q and R as nxn matrices
    R = [[0] * n for _ in range(n)]
    for j in range(n):
        v = [A[i][j] for i in range(n)] # j-th column of A
        for k in range(j):
            R[k][j] = sum(Q[i][k].conjugate() * v[i] for i in range(n)) #projection of
                the current vector with the vector we are orthogalizing
            for i in range(n):
                v[i] -= R[k][j] * Q[i][k]
        norm = sum(v[i].conjugate() * v[i] for i in range(n))
        if abs(norm) < 1e-12: #if column is linearly independent of other column
            R[j][j] = 0
            for i in range(n):
                Q[i][j] = 0
        else:
            R[j][j] = cmath.sqrt(norm)
            for i in range(n):
                Q[i][j] = v[i] / R[j][j]
    return Q, R
def check(temp):
    k = 0
    n=len(temp)
    eigenvalues = [0 + 0j] * n
    while k < n:
        if k < n - 1 and abs(temp[k + 1][k]) > 1e-10:
            a11 = temp[k][k] #solving eigne using quadratic
            a12 = temp[k + 1][k]
            a21= temp[k][k + 1]
            a22= temp[k + 1][k + 1]

            b = -1.0 * (a11+a22)
            c = (a11*a22-a21*a12)
            D= b*b-4*c

            eigenvalues[k] = (-1*b + cmath.sqrt(D)) / 2.0
            eigenvalues[k + 1] = (-1*b- cmath.sqrt(D)) / 2.0
            temp[k + 1][k] = 0
            k += 2
        else:
            eigenvalues[k] = temp[k][k]
            k += 1
    return eigenvalues

def qr_algorithm_with_shift(matrix, Max_iterations=1000, Tolerance=1e−10):
    n = len(matrix)
    for iter in range(Max_iterations):
        if n > 1:
            d = (matrix[n−2][n−2] − matrix[n−1][n−1])/2.0
            if d.real>=0:
                sign=1.0
            else:
                sign=−1.0
            shift = matrix[n−1][n−1] − sign ∗ cmath.sqrt(d ∗ d + matrix[n−1][n−2] ∗
                 matrix[n−2][n−1])
        else:
            shift = matrix[n−1][n−1]

        for i in range(n):
            matrix[i][i] −= shift
        Q, R = gramschmidt(matrix)
        matrix = matmul(R, Q)
        for i in range(n):
            matrix[i][i] += shift
        flag=0
        if(all(abs(matrix[i+1][i])<=Tolerance for i in range(n−1))):
            break
    return check(matrix)

if __name__ == "__main__":
    matrix = [
    [1.0,2.0,5.0],[4.0,5.0,8.0],[2.0,4.0,3.0]#inputing matrix
]
    st=time.time()
    eigenvalues = qr_algorithm_with_shift(matrix)
    et=time.time()
    print("Eigen␣Values␣are:",eigenvalues)
    print("Runtime:",et−st,"seconds")
```

6 Conclusion

This report demonstrated the implementation of the QR Algorithm with Wilkinson's shift for eigenvalue computation. The method accelerates the convergence of the algorithm by effectively applying a shift derived from the bottom-right 2x2 submatrix of the matrix.

The QR algorithm is computationally expensive with a time complexity of $O(n^4)$ in general, but it is a powerful tool for computing eigenvalues of complex and non-symmetric matrices. Alternative methods like the power iteration and inverse iteration may be more efficient in certain cases, but the QR algorithm provides a versatile and robust solution handling complex entries and eigenvalues.

7 Bibliography

1) Wikipedia
2) Youtube Channel:Nick Space Cowboy
3) Youtube Channel:Veritasium
4) Numerical Linear Algebra by David Bau and Lloyd N Trefethen
5) Generative AI