Eigen Value Caluclation

Golla Shriram - AI24btech11010

1 Introduction

If A is $n \times n$ matrix and $\mathbf{v}$ be a non-zero vector of length n. If it satisfies

$$A\mathbf{v} = \lambda\mathbf{v} \tag{1}$$

then λ is known as **eigen value** which may be real or imaginary and $\mathbf{v}$ is **eigen vector**. Eigen values and Eigen vectors are very much useful in understanding the linear transformations and have many other applications in engineering and science.

Eigen values are the roots for characteristic polynomial given by $det(A - \lambda I) = 0$. The idea of directly finding the roots is not viable as the computation difficulty largely increases with n as stated by **Abel–Ruffini theorem** that it's impossible to solve all polynomial equations (of degree 5 or higher) using a general formula made up of basic operations and also there are unavoidable round-off error which cause sensitive changes to the roots of characteristic polynomial. Hence numerical methods are used to approximate the eigen value.

2 Algrothim Selection

Many iterative algorithms were devised to solve the eigenvalue problem by producing sequences that converge to the eigenvalues.

- QR algorithm
- Divide-and-conquer
- Laguerre iteration
- Jacobi eigenvalue algorithm
- Arnoldi and Lanczos algorithms

2.1 QR - Algorithm

The QR algorithm uses **Schur decomposition** of a matrix. It is best applied on **dense matrices**. The overall complexity is $\mathcal{O}(n^3)$ floating point operations. It has **cubic convergence rate**. It is not compatible for sparse matrices.

2.2 Divide and Conquer

The divide and conquer algorithm as name suggest involves partitioning of eigenvalue problem into smaller eigenvalue problems, solving them and combining all the solutions to get desired solutions. The overall complexity is $\mathcal{O}(n^3)$. It is best applied on Hermitian matrices. It has **faster Cubic convergence**. It has high memory requirements .

2.3 Laguerre iteration

This algorithm is based on Laguerre method was designed for polynomials with real zeros and when these are distinct it gives strong convergence right from any starting value but it also applies well in complex plane. It is generally used to find a subset of eigenvalues instead of all eigen values with more degree pf accuracy.The overall complexity is $\mathcal{O}(n^2)$ for single eigenvalue and $\mathcal{O}(n^3)$ for all eigenvalues. It has **cubic convergence**.

2.4 Jacobi eigenvalue algorithm

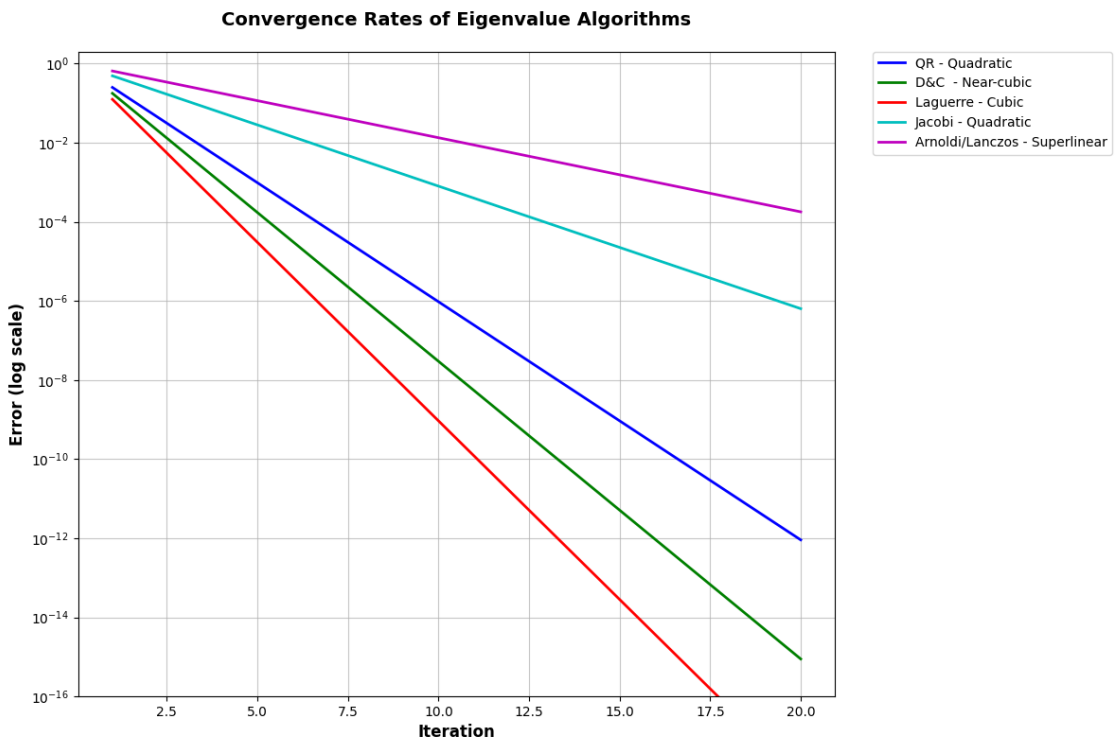
Jacobi eigenvalue algorithm is an iterative method to calculate the eigenvalues of a real symmetric matrix by a sequence of Jacobi rotations. The complexity is $\mathcal{O}(n^3)$. It has **quadratic convergence**. It can handle small matrices but when it comes for large matrices it is not viable.

2.5 Arnoldi and Lanczos algorithms

Arnoldi and Lanczos algorithms both are based on Krylov subspace methods for finding eigen values. These methods are extremely good for finding eigen values of Large sparse matrices. Lanczos best works on **symmetric** mactries where Arnoldi works on **non symmetric** mactries . The complexities are $\mathcal{O}(mn^2)$ for Arnoldi and $\mathcal{O}(mn)$for Lanczos for m steps.

Algorithm	**Matrix Type**	**Complexity**	**Convergence Rate**
QR Algorithm	Dense	$\mathcal{O}(n^3)$	Cubic (with shifts)
Divide and Conquer	Symmetric Tridiagonal	$\mathcal{O}(n^3)$	Faster Cubic
Laguerre Iteration	Polynomials/Subset	$\mathcal{O}(n^2)$	Cubic (simple roots)
Jacobi Algorithm	Real Symmetric	$\mathcal{O}(n^3)$	Quadratic
Arnoldi	General Sparse	$\mathcal{O}(mn^2)$	Superlinear
Lanczos	Symmetric Sparse	$\mathcal{O}(mn)$	Superlinear

Table 1 Comparison of Eigenvalue Algorithms



On comparing above algorithms each algorithm has its own pros and cons but if i were to choose an eigen value solving algorithm i would choose QR algorithm because of its versatility with both symmetric and non - symmetric matrix , faster rate of convergence and its ability for handiling large set of matrices.

Algorithm Implementation

QR- Algorithm

The process involves of decomposition A into Q and R such that $A = QR$, Q is an orthogonal matrix and R is upper triangular matrix. The idea is to start with a matrix A and decompose it into Q and R then we construct another matrix A_1 such that $A_1 = RQ$. We then again decompose the matrix into Q_1 and R_1 such that $A_1 = Q_1R_1$ where Q_1 is an orthogonal matrix and R_1 is upper triangular matrix. We iterate k times until A_k becomes converges closer to diagonal form, the diagonal elements of A_k approximate the eigenvalues of the original matrix A.

The QR decomposition can be done using

- Gram-Schmidt Process
- Householder Reflections
- Rotations

- Cholesky Decomposition

House holder Transformations

The most efficient and widely used is House holder Transformations it is preferred because of its stability and efficiency.

Start with A $n \times n$. We then extract the first column vector x_1 of A:

$$x_1 = \begin{pmatrix} A_{11} \\ A_{21} \\ \vdots \\ A_{n1} \end{pmatrix}$$

We then caluclate norm of x_1 :

$$||x|| = \sqrt{A_{11}^2 + A_{21}^2 + \cdots + A_{n1}^2}$$

Next we define a householder vector $\mathbf{v_1}$ to create a reflection that eliminates the elements below the diagonal in the first column. This vector is given by

$$v_1 = x_1 - ||x||e_1$$

where $e_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$. We then find the unit vector in direction of v_1. Then House holder matrix

$$H_1 = I - 2v_1 v_1^T e_1$$

Apply the Householder transformation matrix to obtain a new matrix $A_1 = H_1 A$ we then apply Householder transformation to handle the submatrix of A_1 to generate H_2 until we find an A_k is transformed into upper triangular matrix i.e $A_k = R$ and $Q = H_1 H_2 ... H_k$ will result an orthogonal matrix such that

$$A = QR$$

By using these values of Q and R we find values of A_1 and then consequently A_k whose diagonal elements could be approximated as the eigen values of A

3 Testing of code

In the following section we are going to compare the Basic Qr algorithm implementation with state of art - Numpy, linalg eigen value function.

Matrix	Eigenvalues by Numpy	Eigenvalues by QR
$\begin{pmatrix} 1 & 5 \\ 6 & 9 \end{pmatrix}$	-1.78232998 11.78232998	-1.782330 11.782330
$\begin{pmatrix} 6 & 4 & 1 \\ 7 & 12 & 9 \\ 23 & 87 & 1 \end{pmatrix}$	36.68518825 4.32581544 -22.0110037	36.685188 -22.011004 4.325815
$\begin{pmatrix} 3 & 4 & 1 & 6 & 9 \\ 2 & 0 & 6 & 2 & 3 \\ 5 & 7 & 1 & 7 & 8 \\ 12 & 6 & 5 & 5 & 1 \\ 41 & 8 & 7 & 1 & 3 \end{pmatrix}$	29.84163693 -18.76990641 4.93433348 0.89484631 -4.90091031	29.841637 -18.769906 4.934333 -4.900910 0.894846
$\begin{pmatrix} 34 & 92 & 56 & 74 & 61 & 15 & 88 & 47 & 13 & 29 \\ 26 & 74 & 58 & 91 & 49 & 36 & 59 & 11 & 82 & 43 \\ 53 & 71 & 34 & 60 & 80 & 2 & 17 & 68 & 99 & 21 \\ 14 & 27 & 63 & 41 & 55 & 9 & 85 & 95 & 30 & 70 \\ 76 & 38 & 52 & 73 & 23 & 78 & 66 & 83 & 65 & 10 \\ 69 & 77 & 28 & 94 & 42 & 50 & 7 & 31 & 12 & 75 \\ 8 & 33 & 57 & 20 & 40 & 86 & 97 & 60 & 64 & 4 \\ 24 & 64 & 1 & 81 & 90 & 46 & 48 & 13 & 93 & 53 \\ 79 & 98 & 85 & 17 & 16 & 18 & 72 & 25 & 22 & 67 \\ 37 & 5 & 84 & 62 & 39 & 51 & 19 & 3 & 50 & 28 \end{pmatrix}$	496.1665913 -82.44133521 + 23.88452113i -82.44133521 - 23.88452113i -24.98345636 + 60.29862066i -24.98345636 - 60.29862066i 1.59109661 + 69.58203827i 1.59109661 - 69.58203827i 33.15250429 + 48.03333842i 33.15250429 - 48.03333842i 65.19579005	496.166591 -82.441335 + 23.884521i -82.441335 - 23.884521i 1.591097 + 69.582038i 1.591097 - 69.582038i -24.859804 + 60.877278i -24.859804 - 60.877278i 64.948485 33.152504 + 48.033338i 33.152504 - 48.033338i

The code can handle really well upto certain limit when we take large values the efficiency of code is reduced and takes time, we also need to increase number of iterations and to get better values reduce tolerance.