Methods To Compute eigen values and eigen vectors

Jakkula Adishesh Balaji
AI24BTECH11016

Contents

1 Abstract **2**

2 Significance of eigen values **2**

3 Power Method **2**

3.1 Algorithm 2
3.2 Time complexity 3
3.3 Pseudocode 3
3.4 Memory complexity 3
3.5 Pros 3
3.6 Cons 3

4 Jacobi method **3**

4.1 Algorithm: 3
4.2 Time Complexity 4
4.3 Pseudocode 4
4.4 Memory complexity 4
4.5 Pros 4
4.6 Cons 5

5 Lanczos Algorithm **5**

5.1 Algorithm 5
5.2 Time Complexity 5
5.3 Pseudocode 5
5.4 Memory Complexity 5
5.5 Pros 5
5.6 Cons 5

6 QR-Algorithm **6**

6.1 Givens Rotations 6
6.1.1 Pseudocode 6

.

6.1.2 Memory Complexity 6
6.2 Gram-Schmidt 7
6.2.1 Pseudocode 7
6.2.2 Memory Complexity 7
6.3 Householder algorithm 7
6.3.1 Basic Idea 7
6.3.2 Householder Transformation 8
6.3.3 Memory Complexity 8
6.3.4 Advantages 8
6.3.5 Disadvantages 9

7 Conclusion **9**

1 Abstract

Applications of mathematics sometimes encounter the following equations: What are the singularities of $A - \lambda I$ where λ is a parameter? What is the behaviour of the sequence of vectors $A^j X_0{}_{j=0}^{\infty}$. Solutions for problems in different disciplines such as engineering, economics and physics can involve ideas related to these equations.

2 Significance of eigen values

Let A be an $n \times n$ matrix and X be a vector of dimension n. We must find scalars λ for which there exists vectors such that

$$AX = \lambda X \tag{1}$$

The linear transformation $T(X) = AX$ **scales** the vector X by a factor of lambda. Here
X is the **eigen vector** that corresponds to the **eigen value** λ
The standard equation is

$$(A - \lambda I)X = 0 \tag{2}$$

A trivial way to solve this would be take the determinant on both sides and equate $|A - \lambda I|$ to zero. But we will not ponder upon this method since it is very computationally expensive and numerically unstable for large matrices.
We will also look at the significance of complex eigen values:
Consider the eigen vector corresponding to the eigen values $\lambda = a + bi$, the eigen vector will be **scaled** by a factor of $||\lambda||$ and rotated by an angle of $\tan^{-1}\left(\frac{b}{a}\right)$. Complex eigen values have interesting real-life applications such as in mechanical vibrations in which the real part represents the damping rate and frequency of the oscillations is described by the complex part. This can also be extended to **RLC Circuits**

The rapid computation of eigen values stands as one of the most transformative innovations of the 20th century, revolutionizing science and technology, we will now explore some efficient eigen-value computing methods

3 Power Method

The power method is used to compute the **dominant eigen value** of a matrix and its corresponding eigen-vector

3.1 Algorithm

Consider the matrix A for which we must calculate the eigen values
Initialize a random vector x_0 and normalize it $\frac{x_0}{|x_0|}$ to prevent overflow
Compute

$$y_k = Ax_{k-1} \tag{3}$$

$$x_k = \frac{y_k}{||y_k||} \tag{4}$$

Estimate the dominant eigenvalue λ_k using the **Rayleigh quotient:**.

$$\lambda_k = x_k^T A x_k \tag{5}$$

Monitor changes in $|\lambda_k - \lambda_{k-1}|$, if it is less than ϵ stop and return the corresponding eigen vector λ_k and the corresponding eigen vector x_k

3.2 Time complexity

Each multiplication with the matrix A with x_{k-1} takes time $\mathcal{O}(n^2)$ and normalizing the vector y_k takes time $\mathcal{O}(n)$. For k iterations, the time complexity is roughly $\mathcal{O}(kn^2)$

3.3 Pseudocode

Algorithm 1 Power Method

initialize $\mathbf{x} = \mathbf{x_0}$
normalize $\mathbf{x_0} = \frac{\mathbf{x_0}}{||\mathbf{x_0}||}$
for $i = 1$ to k **do**
  $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$
  $\lambda = \frac{\mathbf{x}^T \cdot \mathbf{y}}{\mathbf{x}^T \cdot \mathbf{x}}$
  $\mathbf{x} = \frac{\mathbf{y}}{||\mathbf{y}||}$
end for
return $\lambda, \mathbf{x}$

3.4 Memory complexity

Storing the matrix A requires $\mathcal{O}(n^2)$ memory// Storing the vector v requires $\mathcal{O}(n)$ memory // Hence, total memory complexity is $\mathcal{O}(n^2)$

3.5 Pros

The algorithm is simple to implement, requiring only matrix multiplication. It is very efficient for large, sparse matrices and guarantees to converge to the dominant eigen value.
Since we only have to store x_k and A it is extremely memory efficient

3.6 Cons

The number of iterations depends largely on the choice of the initial vector x_0
The convergence rate can be slow since it depends on $|\frac{\lambda_1}{\lambda_2}|$ where λ_2 is the second most dominant eigen value
It only computes a single eigen vector

4 Jacobi method

The Jacobi method is an iterative algorithm used to find all eigenvalues and eigenvectors of a symmetric matrix. It is based on the idea of repeatedly applying rotations (Jacobi rotations) to zero

out the off-diagonal elements of the matrix, transforming it into a diagonal matrix whose diagonal entries are the eigenvalues

4.1 Algorithm:

- Start with $A^{(0)} = A$ and identity matrix $V = I$.
- At each iteration, find the largest off-diagonal element A_{pq} in $A^{(k)}$ and compute the Jacobi rotation matrix $P^{(k)}$, which is used to zero out A_{pq}.
- Update the matrix:
$$A^{(k+1)} = P^{(k)\top} A^{(k)} P^{(k)}.$$
- The eigenvectors accumulate as:
$$V^{(k+1)} = V^{(k)} P^{(k)}.$$
- Repeat until all off-diagonal elements are sufficiently close to zero.

Jacobi Rotation: The Jacobi rotation matrix $P^{(k)}$ is constructed as:

$$P^{(k)} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & c & -s & & \\ & & s & c & & \\ & & & & \ddots & \\ & & & & & \end{bmatrix}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$, with θ chosen to zero out the largest off-diagonal element A_{pq}. The angle θ is computed as:

$$\theta = \frac{1}{2} \tan^{-1} \left(\frac{2A_{pq}}{A_{pp} - A_{qq}} \right).$$

4.2 Time Complexity

Per iteration: Each Jacobi rotation involves searching for the largest off-diagonal element, which takes $\mathcal{O}(n^2 - n)$ and then performing the rotation which also takes $\mathcal{O}(n^2)$ time.
Overall complexity would be $\mathcal{O}(n^3)$ since it usually requires n iterations

4.3 Pseudocode

Algorithm 2 Jacobi Method

```
Initialize x_0 with zeros
Set iteration counter k = 0
repeat
    for each row i = 1 to n do
        sum = 0
        for each column j = 1 to n, j ≠ i do
            sum = sum + A[i][j] · x_j^(k)
        end for
        x_i^(k+1) = (b_i − sum) / A[i][i]
    end for
    Calculate the norm ||x^(k+1) − x^(k)||
    if norm < ε then
        Exit the loop
    end if
    Increment iteration counter k
until convergence or k ≥ k_max
```

4.4 Memory complexity

Storing the matrix A requires $\mathcal{O}(n^2)$ memory
Storing the vector solution v_i requires $\mathcal{O}(n)$ memory
Temporary storage for next iteration v_{i+1} requires $\mathcal{O}(n)$ memory

4.5 Pros

Simple to implement for symmetric matrices.
Highly precise for small to moderate sized matrices.

4.6 Cons

Slow for large matrices
Requires high number of iterations
Limited to symmetric matrices

5 Lanczos Algorithm

5.1 Algorithm

The Lanczos algorithm is an iterative method for approximating eigenvalues and eigenvectors of large, symmetric matrices. It reduces a matrix $A \in \mathbb{R}^{n \times n}$ to a tridiagonal matrix T_k, which approximates the eigenvalues of A. Given a normalized initial vector v_1, the algorithm iterates as follows:

- Set $\beta_0 = 0$, $v_0 = 0$.
- For $j = 1, 2, \ldots$:

$$
\begin{aligned}
w_j &= Av_j - \beta_{j-1} v_{j-1}, \\
\alpha_j &= v_j^\top w_j, \\
w_j &= w_j - \alpha_j v_j, \\
\beta_j &= \|w_j\|, \quad v_{j+1} = \frac{w_j}{\beta_j}.
\end{aligned}
$$

The matrix T_k formed from α_j and β_j is:

$$
T_k = \begin{bmatrix}
\alpha_1 & \beta_1 & 0 & \cdots & 0 \\
\beta_1 & \alpha_2 & \beta_2 & \cdots & 0 \\
0 & \beta_2 & \alpha_3 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & \alpha_k
\end{bmatrix}.
$$

The eigenvalues of T_k approximate the largest (or smallest) eigenvalues of A.

5.2 Time Complexity

Each iteration requires $\mathcal{O}(\text{nnz}(A))$ operations for sparse matrices, where $\text{nnz}(A)$ is the number of non-zero elements. The overall complexity is $\mathcal{O}(k \cdot \text{nnz}(A))$ for k iterations.

5.3 Pseudocode

5.4 Memory Complexity

Storing the matrix A requires $\mathcal{O}(n^2)$ memory
Storing $k+1$ vectors of size n requires $\mathcal{O}(kn)$ memory
Storing the tridiagonal matrix requires $\mathcal{O}(k)$ memory Hence total memory complexity is $\mathcal{O}(n^2 + kn)$

Algorithm 3 Lanczos Algorithm for Eigenvalue Computation

Chose an initial random vector v_1 and normalize it

$v_1 = \frac{v_1}{||v_1||}$
Set $\beta_0 = 0$
Set $\alpha_0 = 0$
for $i = 1$ to k **do**
    Compute $w_i = Av_i$
    Subtract the component along v_{i-1}
    $w_i' = w_i - \beta_{i-1}v_{i-1}$
    Compute α_i as $\alpha_i = v_i^T w_i'$
    $\beta_i = ||w_k''||$
    if $\beta_i \neq 0$
        continue;
    else
        break $v_{k+1} = \frac{w_k''}{\beta_k}$
end for
Return: Tridiagonal of of T

5.5 Pros

Efficient for large, sparse matrices; low memory usage; good convergence for extreme eigenvalues.

5.6 Cons

Numerical instability without reorthogonalization; works only for symmetric matrices; struggles with degenerate eigenvalues.

6 QR-Algorithm

One of the most crucial innovations of the 20th century. It is currently the most widely used method for computing eigen-values for a general matrix. **We will be using this algorithm with the householder reflections method for computing the eigen values**.
QR algorithm is an iterative algorithm that decomposes the matrix A into the product of an orthogonal matrix Q and an upper triangular matrix R, then updating the matrix by multiplying R and Q. Mathematically,

$$A_k = QR \tag{6}$$
$$A_{k+1} = RQ \tag{7}$$

Pseudocode After several iterations, the matrix A_n converges to an upper triangular matrix. The

Algorithm 4 QR method

for $i = 1$ to k **do**
    decompose A_i as Q_i and R_i
    set $A_{i+1} = R_i \cdot Q_i$
end for
return: A_k

diagonal entries of this matrix are the eigenvalues of the original matrix A. We will now explore some methods for QR decomposition

6.1 Givens Rotations

Used when the matrix is sparse or has a special structure. It works by zeroing out individual elements by simple 2D rotations. For each element i, j a rotation matrix $G(i, j, \theta)$ is applied.

6.1.1 Pseudocode

Algorithm 5 Given's rotations

for $j = 1$ to $n - 2$ **do**
  for $i = n$ to $j + 2$ **do**
    Compute Givens rotation for T[i, j]:
    $r = sqrt(T[j, j]^2 + T[i, j]^2)$
    $c = \frac{T[j,j]}{r}$
    $s = \frac{T[i,j]}{r}$
    Construct the matrix
    $G = I$
    $G[j, j] = c, G[j, i] = s$
$G[i, j] = -s, G[i, i] = c$

    Apply Givens rotation
    $T = G^T \cdot T \cdot G$
    Update $Q = Q \cdot G$
  end for
end for
return: T, Q

6.1.2 Memory Complexity

Storing the matrix A requires $\mathcal{O}(n^2)$ memory
Storing the vector v for each Givens rotation requires $\mathcal{O}(n)$ memory
Storing the Givens rotation matrix requires $\mathcal{O}(n)$ memory
Hence, the total memory complexity is $\mathcal{O}(n^2)$

6.2 Gram-Schmidt

One of the oldest algorithms to perform QR decompositions. It is relatively simple and is accurate for small matrices. In the Gram-Schmidt process, you take each column of matrix A, project it onto the previously computed orthogonal vectors, and subtract the projection to make the vector orthogonal to the earlier ones. This method is faster for smaller matrices or when high numerical stability is not required.

6.2.1 Pseudocode

Initialize Q as a null matrix
Initialize R as a null matrix
for $j = 1$ to n **do** $v = A_j$ column vector of A
  for $i = 1$ to $j - 1$ **do**
    $R[i, j] = Q_i^T \cdot v$
    $v- = R[i, j] \cdot Q_i$
  end for
  $R[j, j] = ||v||$
  $Q_j = v/R[j, j]$
end for$Q$, R

6.2.2 Memory Complexity

Storing the matrix A requires $\mathcal{O}(n^2)$ memory
Storing the orthogonal vectors requires $\mathcal{O}(kn)$ memory
Storing the coefficients for the projections requires $\mathcal{O}(n)$ memory
Hence, the total memory complexity is $\mathcal{O}(n^2)$

6.3 Householder algorithm

The householder algorithm is a numerically stable and efficient method for computing the QR factorization of a matrix A. It involves a series of orthogonal transformations (Householder reflections) that reduce the matrix A to upper triangular form.

6.3.1 Basic Idea

For a given matrix $A \in \mathbb{R}^{m \times n}$, the goal is to compute an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ and an upper triangular matrix $R \in \mathbb{R}^{m \times n}$ such that:

$$A = QR$$

At each step, a Householder transformation is applied to zero out the elements below the diagonal in one column of the matrix.

6.3.2 Householder Transformation

A Householder transformation H is defined as:

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

6.3.3 Memory Complexity

Storing the matrix A requires $\mathcal{O}(n^2)$ memory
Storing the Householder vectors requires $\mathcal{O}(n)$ memory
Storing the orthogonal matrix Q requires $\mathcal{O}(n^2)$ memory
Hence, the total memory complexity is $\mathcal{O}(n^2)$

where $\mathbf{v}$ is a vector chosen such that the transformation zeros out specific elements below the diagonal of a column in the matrix. For a given vector $\mathbf{x}$, the vector $\mathbf{v}$ is computed as:

$$\mathbf{v} = \mathbf{x} + \text{sign}(x_1)\|\mathbf{x}\|_2\mathbf{e}_1$$

where $\mathbf{e}_1$ is the first standard basis vector.

Pseudocode

Input: Matrix A of size $n \times n$.

Output: Orthogonal matrix Q of size $n \times n$. Upper triangular matrix R of size $n \times n$.

$Q \leftarrow I_n$
$R \leftarrow A$
for $k = 1$ to $n - 1$ **do**
  $x \leftarrow R[k : n, k]$
  $\|x\| \leftarrow \sqrt{\sum x_i^2}$
  Define e_1 as a vector of size $n - k + 1$ with $e_1[1] = 1$ and other entries 0
  $v \leftarrow x + \text{sign}(x_1) \cdot \|x\| \cdot e_1$
  $v \leftarrow v/\|v\|$
  $H \leftarrow I - 2 \cdot v \cdot v^\top$
  $R[k : n, k : n] \leftarrow H \cdot R[k : n, k : n]$

Expand H to size $n \times n$ (fill with identity values outside the submatrix)
$Q \leftarrow Q \cdot H^\top$
end for
Output: Q, R

1. Start with the matrix $A \in \mathbb{R}^{m \times n}$. 2. For each column k of A, construct a Householder matrix H_k that zeros out the elements below the diagonal in that column. 3. Apply the transformation $A \leftarrow H_k A$ to update the matrix. 4. Continue this process for each column, resulting in a matrix that is upper triangular. 5. The product of all the Householder matrices $Q = H_1 H_2 \dots H_n$ is orthogonal, and the resulting matrix is upper triangular.

6.3.4 Advantages

- **Numerical Stability**: The Householder algorithm is more numerically stable than methods like Gram-Schmidt since it does not suffer from computational overhead. In the Gram-Schmidt method, the computation of each column vector depends on all its preceding vectors. So, although the time complexity of both the methods is the same, householder algorithm doesnt suffer from this computational overhead and is stable for larger matrices.
- **Efficiency for Dense Matrices**: It requires fewer operations than Givens rotations and is well-suited for dense matrices.
- **Orthogonality**: The orthogonal matrix Q is obtained as the product of orthogonal Householder transformations.

6.3.5 Disadvantages

- **Not Suitable for Sparse Matrices**: For sparse matrices, applying Householder transformations can destroy the sparsity structure, leading to inefficiency.

7 Conclusion

The Householder algorithm is a highly efficient and numerically stable method for performing QR factorization, and it plays a crucial role in modern numerical linear algebra. Due to its robustness, particularly in dealing with large and dense matrices, it is widely regarded as one of the most reliable techniques for QR decomposition. In the context of the software assignment, I employed **QR decomposition with Householder transformations** for computing the eigenvalues of a given matrix

References

1. *3Blue1Brown:Essence of linear algebra*. Link.
2. Wikipedia - *Eigenvalues Algorithm* Link.
3. Yang, *Applied Numerical Methods Using MATLAB*