MATRIX THEORY

EE1030
SOFTWARE ASSIGNMENT

Homa Harshitha Vuddanti
(EE24BTECH11062)

Aim : To write a code using C/rust/python, to compute eigenvalues of a given matrix, using a suitable algorithm.

Algorithms chosen:

1) QR algorithm
2) Power iteration with deflation

Chosen language: C

1) **QR algorithm** : The QR algorithm uses the idea of QR decomposition to iteratively solve for eigenvalues.

   QR decomposition:
   We are supposed to write the given matrix A(nxn) as the product of two matrices, Q and R, such that Q is an orthogonal matrix and R is an upper triangular matrix.
   We can use **Gram-Schmidt process** for calculating Q and R,
   Say the columns of A are $a_1, a_2, \ldots, a_n$ and the columns of Q are $q_1, q_2, \ldots, q_n$, then q_1 is the normalized version of a_1, the rest of q_2 are obtained by subtracting a_2 from projection of a_2 and q_1, and normalizing it.
   This goes on till q_n.
   After we obtain Q, we can get the matrix R by

$$R = Q^\top A \tag{1}$$

   Now for QR algorithm, we first write $A = QR$, and then a matrix $A_1 = RQ$, we can now consider A_1 to be our new matrix A and keep doing this till it converges to an upper triangular matrix, whose diagonal elements will give us the eigenvalues.

   a) **C code on computing eigenvalues using QR algorithm** :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
void qr_decomposition(double** A,double** Q,double** R,int n){
    for(int j=0;j<n;j++){
        for(int i=0;i<n;i++){
          //here we are copying jth column of A to jth column of Q.
            Q[i][j]=A[i][j];
        }
        for(int k= 0;k<j;k++){
            R[k][j]=0;
            for(int i=0;i<n;i++){
              //projection of Q on A
                R[k][j]+=Q[i][k]*A[i][j];
            }
            for(int i=0;i<n;i++){
              //subtracting vector from projection to get orthogonal
                Q[i][j]-=R[k][j]*Q[i][k];
            }
        }
        // making Q as the orthogonal vector
        R[j][j]=0;
        for(int i=0;i<n;i++){
            R[j][j]+=Q[i][j]*Q[i][j];
        }
        R[j][j] = sqrt(R[j][j]);
        for(int i=0;i<n;i++){
          //making it unit vector.
            Q[i][j]=Q[i][j]/R[j][j];
        }
    }
}

void qr_algorithm(double** A,int n) {
    double** Q=(double**)malloc(n*sizeof(double*));
    double** R =(double**)malloc(n*sizeof(double*));
    for(int i=0;i<n;i++) {
        Q[i]=(double*)malloc(n*sizeof(double));
        R[i]=(double*)malloc(n*sizeof(double));
    }

    for(int iter=0;iter<1000;iter++) {
      // taking maximum iterations
        qr_decomposition(A,Q,R,n);
        //temp for storing A_1
        double** temp =(double**)malloc(n*sizeof(double*));
        for(int i=0;i<n;i++){
            temp[i]=(double*)malloc(n*sizeof(double));
```

```
        }

        for(int i=0;i<n;i++) {
            for (int j=0;j<n;j++) {
                temp[i][j]=0;
                for (int k=0;k<n;k++) {
                  //temp=RQ
                    temp[i][j]+=R[i][k]*Q[k][j];
                }
            }
        }

        for (int i=0;i<n; i++) {
            for (int j=0;j<n;j++) {
             // A_1=temp=RQ
                A[i][j]=temp[i][j];
            }
        }

        // Free temp
        for (int i=0;i<n;i++) {
            free(temp[i]);
        }
        free(temp);
    }

    // Free Q,R
    for (int i=0;i<n;i++) {
        free(Q[i]);
        free(R[i]);
    }
    free(Q);
    free(R);
}

int main() {
    int n;
    scanf("%d",&n);
    double** A=(double**)malloc(n * sizeof(double*));
    for(int i=0;i<n;i++) {
        A[i]=(double*)malloc(n * sizeof(double));
    }
    for(int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            scanf("%lf",&A[i][j]);
```

```
        }
    }
    qr_algorithm(A,n);
    for(int i=0;i<n;i++){
        printf("%lf\n",A[i][i]); // taking diagonal of A for eigen values
    }

    // Free
    for(int i=0;i<n;i++){
        free(A[i]);
    }
    free(A);

    return 0;
}
```

b) **Example:** Input:
2
4 1
2 3
Output:
5.000000
2.000000
Which is the same as expected values.

c) **Pros and cons:** It gives us all the eigenvalues for a non-symmetric matrix as well and handles closely spaced eigenvalues, however, it can be computationally expensive.

d) **Time complexity:**
qr_decomposition: $O\left(n^2\right)$
matrix multiplication: $O\left(n^3\right)$ qr_algorithm: $O\left(n^3\right)$ for each iteration.
Overall : $O\left(1000 * n^3\right)$ as qr_algorithm dominates.

2) **Power iteration with deflation :**

a) **Power iteration:** We start with a random vector v and we calculate

$$\lambda = \frac{v^\top A v}{v^\top v} \tag{2}$$

.
This gives an estimation of eigenvalue. We then normalize new vector Av. We repeat this until the eigenvalue reaches a stable value. This will give us λ_1 which is the dominant eigenvalue.

Deflation : We need to now remove the dominant value to calculate the next

eigenvalue. We do this using deflation. We take new

$$A_{deflated} = A - \lambda_1 v_1 v_1^\top \tag{3}$$

. This makes us left with one less eigenvalue. We now repeat power iteration with $A_{deflated}$.

b) **C code for calculating eigenvalues using Power iteration with deflation:**

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//dot product function
double dot_product(double *a,double *b,int n){
    double result=0;
    for(int i=0;i<n;i++){
        result+=a[i]*b[i];
    }
    return result;
}

//norm
void norm(double *a,int n) {
    double norm=0.0;
    for(int i=0;i<n;i++){
        norm+=a[i]*a[i];
    }
    norm=sqrt(norm);
    for(int i=0;i<n;i++){
        a[i]/=norm;
    }
}

//function to do deflation(doing A-lambda v v^T)
void deflate_matrix(double *A,int n,double lambda,double *v) {
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            A[i*n+j]-=lambda*v[i]*v[j];
        }
    }
}

//to find dominant eigenvalue
double dominant_eigenvalue(double *A,int n,double *v) {
    double *Av=(double *)malloc(n*sizeof(double));
```

```
    double d=0,d_old=0;

    //taking v as a random vector
    srand(42);
    for(int i=0;i<n;i++){
        v[i]=rand()/(double)RAND_MAX;
    }
    norm(v,n);

    //power iteration
    for(int iter=0;iter<1000;iter++){
        //Av
        for(int i=0;i<n;i++){
            Av[i]=0;
            for(int j=0;j<n;j++){
                Av[i]+=A[i*n+j]*v[j];
            }
        }
        d_old=d;
        d=dot_product(v,Av,n);
        norm(Av,n);
        //making v new=Av
        for(int i=0;i<n;i++){
            v[i]=Av[i];
        }

        //convergence check
        if(fabs(d-d_old)<1e-6){
            break;
        }
    }

    free(Av);
    return d;
}

//finding eigenvalues
void eigenvalues(double *A, int n, double *lambdas) {
    double *v=(double *)malloc(n*sizeof(double));
    double *A_copy=(double *)malloc(n*n*sizeof(double));
    //making copy of A
    for(int i=0;i<n*n;i++){
        A_copy[i]=A[i];
    }
    //freeing dominant eigenvalue
```

```
    for(int i=0;i<n;i++){
        lambdas[i]=dominant_eigenvalue(A_copy,n,v);
        deflate_matrix(A_copy,n,lambdas[i],v);
    }

    free(v);
    free(A_copy);
}
int main() {
    int n;
    scanf("%d",&n);
    double *A= (double *)malloc(n*n*sizeof(double));
    double *lambdas=(double *)malloc(n*sizeof(double));
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            scanf("%lf",&A[i*n+j]);
        }
    }

    //getting eigenvalues
    eigenvalues(A,n,lambdas);
    for (int i = 0; i < n; i++) {
        printf("%f\n", lambdas[i]);
    }

    //free
    free(A);
    free(lambdas);

    return 0;
}
```

c) **Example :**
Input:
3
6 2 1
2 3 1
1 1 1
Output:
7.287992
2.133074
0.578933
Which are close to the actual eigenvalues.

d) **Pros and cons:** It is efficient for large matrices, finds multiple eigenvalues.

However, it does not work well with non-symmetric values.

e) **Time complexity:**
power iteration i.e, dominant_eigenvalue function:
Matrix multiplication: $0(n)$
Normalization : $O(n)$
Convergence : $O(n)$
for this function, overall : $O\left(max\_iter * n^2\right)$
deflate_matrix function : $O\left(n^2\right)$
Total time complexity: $O\left(1000 * n^3\right)$

Comparing the algorithms:

1) For an input :
3
1 0 0
0 2 0
0 0 3
The QR algorithm code gives an output of
1.000000
2.000000
3.000000
as expected, However, the power iteration with deflation gives
2.999999
2.000000
1.000000
Therefore, QR algorithm is accurate for small matrices.
2) 3
1 1 0
0 1 0
0 0 2
Since it is non symmetric matrix, QR algorithm works best, it gives
1
1
2
accurately, whereas Power iteration gives
2.000000
1.001003
0.998997

3) Time taken:(if we add time.h and use clock())
For an example input :
4
6 3 1 2
3 8 5 1
1 5 9 4

2 1 4 7

QR algorithm takes: 0.001191 seconds

Power iteration with deflation takes: 0.000013 seconds.

Evidently, Power iteration takes much less time. One reason for this is that due to deflation, Power iteration code reduces complexity of matrix after each iteration.

Conclusion :

QR algorithm is more accurate for all general matrices, However it is expensive to compute, and works best for small sized matrices. Power iteration with deflation takes much less time compared to QR algorithm, but is not accurate enough for general matrices. It is much less complex.

Hence, I would choose QR algorithm as it is more accurate in giving eigenvalues for all general matrices even though it takes more amount of time.