W zadaniu miałam za zadanie znaleźć wartości własne macierzy

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & -1 \end{bmatrix}$$

trzema różnymi metodami: metodą Rayleigha, potegową oraz algorytmem QR.

**Macierz ta jest symetryczna, lecz nie jest dodatnio określona.**

**Wartości własne.**

Wektor x jest wektorem własnym macierzy A jeśli istnieje taka liczba λ, że $Ax = \lambda x$. λ to wartość własna macierzy A.

Jeśli macierz da się zdiagonalizować, to elementy na diagonali to wartości własne tej macierzy. (Będę to wykorzystywać w ostatniej metodzie pozyskiwania wartości własnych).

**Metoda potęgowa**

$$\begin{cases} Ay_k = z_k & (1a) \\ y_{k+1} = \frac{z_k}{\|z_k\|} & (1b) \end{cases}$$

Pierwszą metodą do znajdywania wartości własnych którą omówię jest metoda potęgowa. Warunkiem tej metody jest symetryczność macierzy A. Metoda ta polega na iterowaniu wielokrotnie (1), którego wynik zbiega do unormowanego wektora własnego, który z kolei odpowiada największej wartości własnej. Gdy iteracja zbiegnie się do punktu stałego, możemy wyliczyć wartość własną.

A więc dzięki tej metodzie odnajdujemy największą wartość własną (dla dodatnich wartości własnych). Aby obliczyć kolejną wykorzystamy poprzedni wektor własny który został już wyliczony. Wykorzystamy dodatkowo w obliczeniach $e_1$- poprzednio znaleziony wektor własny. Należy pamiętać jednak o tym, że aby odnalezienie kolejnej wartości było możliwe, nowe wektory $y_k$ muszą być prostopadłe do poprzednio znalezionego wektora własnego.

$$\begin{cases} Ay_k = z_k & (2a) \\ z_k = z_k - e_1(e_1^T z_k) & (2b) \\ y_{k+1} = \frac{z_k}{\|z_k\|} & (2c) \end{cases}$$

Wtedy wartość własna (jej moduł) to będzie:

$$|\lambda| = \left| \frac{z^k}{y^{k-1}} \right|$$

W skrócie kod mojego programu prezentuje się nastepująco (wektor kolejnych przybliżeń nazwałam v ):

Algorytm dla jednej (wykonywany 3 razy - dla każdej osobno) wartości własnej:

1.wybieram dowolony wektor $y_k$którego norma wynosi 1

2. obliczam kolejne przybliżenie k-tego wektora własnego

3. wyliczam z niego lambdę

4.Sprawdzam jak bardzo jest podobna do lambdy z poprzedniej iteracji - jeśli różnią się o mniej niż epsilon to kończę

Odnalezione w ten sposób wartości własne (w 22 iteracjach) wynoszą:

$$\lambda_1 = 8.5485128519 \qquad \lambda_2 = 4.5740872266 \qquad \lambda_3 = 0.0255743726$$

Jednak pozbawione są one swojego znaku, jako że otrzymaliśmy jedynie ich moduł. Ujemne wartosci własne rozpoznajemy po tym, że po ustabilizowaniu się kierunku wektora własnego, jego współrzędne zmieniają znak w kolejnych iteracjach.

Po wydrukowaniu ich w poszczególnych iteracjach możemy wnioskować, że druga wartość własna ma wartość ujemną.

**Złożoność metody:**

Koszt jednej iteracji wyliczania wektora własnego wynosi O($N^2$). (Iterację tę w sumie dla wszystkich trzech wektorów własnych wykonuje ok. 20 razy, ale nadal koszt wynosi O($N^2$)). W tej metodzie, nie obliczam żadnego układu równań, wykonuję mało "skomplikowane" obliczenia pod względem czasowym, dzięki czemu koszt tej metody jest dużo mniejszy w stosunku do kolejnych przedstawionych tu metod. Kosztowność tej metody polega na wymaganej co kilka iteracji ortogonalizacji wektora. Być może możnaby przyspieszyć działanie programu gdyby zastosować inne wartości wektorów początkowych.

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


double liczenieNormy(double *tab,int size){
double norma = 0;
double temp=0;
int i=0;
for(i;i<size;++i){
temp=tab[i]*tab[i];
norma+=temp;
}
norma=sqrt(norma);
return norma;
}

double skalarne(double *tab, double *tab2,int size){
double sum=0;
for(int i=0;i<size;++i){
sum+=tab[i]*tab2[i];
}
return sum;
}

int main(){

double A[3][3]={{1.0,2.0,3.0},{2.0,4.0,5.0},{3.0,5.0,-1.0}};
int N = sizeof(A[1])/sizeof(double);
int i=0,j=0,k=0;
double epsilon=0.00000001;
double w[3]={1,1,1};
double v[3]={1,1,1};
double u[3]={1,1,1};
double lambdaN[]={0.0,0.0,0.0};
double lambdaN_1[]={0.0,0.0,0.0};
double suma[3]={0.0,0.0,0.0};
double wszystkieV[2][3]={{1,1,1},{1,1,1}};
double tablicaV[3][3]={{1,1,1},{1,1,1},{1,1,1}};
printf("n wynosi %d\n",N);



for(k=0;k<N;k++){
do{
for(i=0;i<N;++i){
w[i]=0;
for(j=0;j<N;++j){
w[i]+=A[i][j]*tablicaV[k][j];
}
}
```

```
lambdaN_1[k]=lambdaN[k];
for(i=0;i<N;++i){
u[i]=w[i]/liczenieNormy(w,N);
}


lambdaN[k]=liczenieNormy(w,N)/liczenieNormy(tablicaV[k],N);
for(i=0;i<k;++i){
for(j=0;j<N;++j){
suma[j]+=skalarne(tablicaV[i],u,N)*tablicaV[i][j];
}
}

for(i=0;i<N;++i){
tablicaV[k][i]=u[i]-suma[i];
suma[i]=0;
}

}while(fabs(lambdaN_1[k]-lambdaN[k])>epsilon);
}


for(i=0;i<3;i++){
  printf("to powinna byc lambda %.10f\n",lambdaN[i]);

}

return 0;
}
```

**Metoda Rayleigha**

Metoda Rayleigha jest kolejną metodą którą znajdujemy największą wartość własną (co do modułu), wywodząca się z metod potęgowych (potęgowej i odwrotnej potęgowej). Tutaj, w przeciwieństwie do metody potęgowej która nieco przypomina metodę iteracji prostej, wymagane jest rozwiązanie układu równań. Ja ten układ rozwiązuję metodą LU zaimplementowaną w bibliotece matematycznej GSL. Aby przyspieszyć odnalezenie wektorów i wartości własnych zastepuje macierz A -> $(A-\lambda I)$.Dzięki temu wystarcza mi około 10 iteracji w sumie na odnalezienie wszystkich wartości, czyli w tym wypadku 2 razy mniej niż w poprzedniej metodzie.

**Algorytm programu** dla każdej wartości własnej prezentuje się następująco (dla wektora początkowego v i przybliżonej wartości $\lambda$ ):

1.Obliczam $(A-\lambda I)$. Następnie wykonuję rozkład LU $(A-\lambda I) \cdot w = v$ aby obliczyć wektor w.

2.Obliczam nowe lambda (jest to tzw iloraz Rayleigha) : $\lambda = v_k^T A v_k$

3.Sprawdzam czy zachodzi $|\lambda_k^{n-1} - \lambda_k^n| < \epsilon$ (błędu obliczeń)

4.Jeśli tak, to kończę, w przeciwnym wypadku wykonuję powyższe jeszcze raz.

Wyniki odnalezione tą metodą w sumie w 12 iteracjach to:

$$\lambda_1 = 8.5485128532$$
$$\lambda_2 = -4.5740872259$$
$$\lambda_3 = 0.0255743726$$

**Złożoność i analiza algorytmu:**

Metoda ta jest ulepszeniem metody potęgowej, lecz aby móc ją zastosować musimy wiedzieć mniej więcej jakie wartości własne posiada macierz - inaczej nie jesteśmy w stanie "naprowadzić" algorytmu w odpowiedni przedział poszukiwań.

Złożoność algorytmu jest większa niż zwykłej metody potęgowej, bo jest $O(N^3)$, ponieważ rozwiązujemy tutaj układ równań metodą LU, reszta operacji nie przewyższa złożonością LU.

Jeśli chodzi o optymalizację, to starałam się nie kopiować nigdzie wektorów ani macierzy i korzystałam z gsl_matrix_view, które to są tymczasowymi obiektami do operowania na macierzach. Gdyby macierz była dodatnio określona zastosowałabym rozkład Cholesky'ego zamiast LU.

```c
#include <stdio.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_matrix.h>
#include <stdlib.h>
#include <math.h>

double mnozenieSkalarne(double *tab, double *tab2,int size){
double sum=0;
for(int i=0;i<size;++i){
sum+=tab[i]*tab2[i];
}
return sum;
}

double liczenieNormy(double *tab,int size){
double norma = 0;
double temp=0;
int i=0;
for(i;i<size;++i){
temp=tab[i]*tab[i];
norma+=temp;
}
norma=sqrt(norma);
return norma;
}

int main(){
int counter=0;
 double a[] = { 1,2,3,2,4,5,3,5,-1};
 double copyA[9]={0,0,0,0,0,0,0,0,0};
 double A[3][3] = { {1,2,3},{2,4,5},{3,5,-1}};
 double vK[]={1,0,0};
 double lambdaN[]={8.0,-4.0,0.0};
 double lambdaN_1[]={0,0,0};
 double I[]={1,0,0,0,1,0,0,0,1}; // jako wektor a to macierz 3x3
 double w[]={0,0,0}; // niewiadoma
 double mnoz[3]={0,0,0};
 int i=0,j=0,k=3,n=0;
 int N = sizeof(A[0])/sizeof(double);
 double epsilon=0.00000001;
void mnozenieMacierzWektor(){
  for(i=0;i<N;++i){
   mnoz[i]=0;
for(j=0;j<N;++j){
   mnoz[i]+=A[i][j]*vK[j];
  }
 }

 }

 void lu(){
```

```
gsl_matrix_view m = gsl_matrix_view_array (copyA, 3, 3);
gsl_vector_view v = gsl_vector_view_array (vK, 3);
gsl_vector *x = gsl_vector_alloc (3);
int s;
gsl_permutation * p = gsl_permutation_alloc (3);
gsl_linalg_LU_decomp (&m.matrix, p, &s);
gsl_linalg_LU_solve (&m.matrix, p, &v.vector, x);
//printf ("x = \n");
//gsl_vector_fprintf (stdout, x, "%g");

for(i=0;i<3;++i){
w[i]=gsl_vector_get(x,i);
}

gsl_permutation_free (p);
gsl_vector_free (x);

 }
 void A_lambdaI(){
for(i=0;i<(3*N);++i){
copyA[i]=a[i]-lambdaN[k]*I[i];
}
 }
// wlasciwy kod
for(k=0;k<N;++k){
vK[0]=1;
vK[1]=0;
vK[2]=0;
  do{
    A_lambdaI();
   lu();
   //solve (A - lambda *I)w= v; dla w

   for(i=0;i<N;++i){
   vK[i]=w[i]/(liczenieNormy(w,N));
   }
lambdaN_1[k]=lambdaN[k];
mnozenieMacierzWektor();
    lambdaN[k]=mnozenieSkalarne(vK,mnoz,N);
counter++;
  } while(fabs(lambdaN_1[k]-lambdaN[k])>epsilon);
printf("lambda %d %.10f\n",k,lambdaN[k]);
 }
printf("Counter %d \n",counter);
 return 0;
}
```

**Algorytm QR**.

Niech macierz A posiada faktoryzację QR, A = QR.

Iloczyn czynników Q i R wziętych w odwrotnej kolejnosci jest równy:

$$A = QR$$
$$Q^{-1}AQ = RQ$$

Wiemy, że macierz Q w rozkładzie QR jest macierzą ortogonalną, więc równanie możemy zapisać

$$Q^T AQ = RQ$$

Widzimy, ze wymnożenie czynników faktoryzacji QR w odwróconej kolejnosci stanowi ortogonalną transformację podobieństwa macierzy A.

Jeśli iterowalibyśmy wielokrotnie powyższe równanie

$$A_1 = Q^1 R^1$$

A dalej

$$A_2 = R^1 Q^1 = Q^2 R^2$$
$$A_3 = R^2 Q^2 = Q^3 R^3$$

W ogólności byłoby to : $A_n = R^{n-1}Q^{n-1} = Q^n R^n$

Kazda prawa strona $Q^k R^k$ reprezentuje faktoryzację QR dokonywaną w k-tym kroku iteracji, a macierz $A_n$ jest ortogonalnie podobna do macierzy $A$.

Co ciekawe, zachodzi pewna zależność, że jeśli A jest macierzą diagonalizowalną a wartości własne są rzeczywiste i różne od siebie, to $A_n$ jest zbieżna do macierzy trójkątnej, z wartościami własnymi na diagonali.

Dzięki tej metodzie, możemy liczyć od razu wszystkie wartości własne, a nie jak wcześniej rozpoczynać od największej (lub najmniejszej) i potem dzięki już znalezionej szukać następnej.

**Algorytm** mojego programu prezentuje się następująco:

1. Dokonuję rozkładu QR (funkcjami z biblioteki GSL, potrzebuję do tego dwóch metod, ponieważ pierwsza metoda - dekompozycji robiona jest metodą Householdera i muszę jeszcze użyć metody 'unpack' aby uzyskać oddzielnie macierze Q i R)

2. Mnożę macierze R*Q (funkcją dgemm)

3. Porównuję za każdym razem elementy na diagonali z tymi z poprzedniej iteracji i jeśli różnią się o mniej niż $\epsilon$ (wszystkie 3) to kończę.

Wyniki odnalezione tą metodą w sumie w 12 iteracjach to:

$$\lambda_1 = 8.5485128520 \quad \lambda_2 = -4.5740872246 \quad \lambda_3 = 0.0255743726$$

**Złożoność**

Złożoność tej metody jest $O(N^3)$ ponieważ najdroższą operacją wykonywaną za każdym razem jest faktoryzacja QR. Jest to ogółem metoda, która najmniej się opłaca, stosuje się ją zazwyczaj gdy macierze są rzadkie lub gdy nie ma innego wyjścia.

Mnie zbiegła ona w w 18 iteracji. Okazała się więc najmniej opłacalną do rozwiązania tego przykładu.

Być może, aby metoda była bardziej optymalna, mój program mógłby użyć jakiejś jednej metody z biblioteki GSL, która od razu liczy iloczyn RQ i robi to efektywniej niż kiedy ja liczę dekompozycję, rozdzielenie na 2 macierze i mnoże je w odrotnej kolejności.

# Podsumowanie - błędy

W ogólności, porównując wszystkie 3 metody i ich wyniki ze sobą :

| | | |
|---|---|---|
| 8.5485128519 | $(\pm)$4.5740872266 | 0.0255743726 |
| 8.5485128532 | $-$ 4.5740872259 | 0.0255743726 |
| 8.5485128520 | $-$ 4.5740872246 | 0.0255743726 |

Widzimy, że wszystkie wyznaczają te same wartości aż do $10^{-8}$. Potem już wystepują rozbieżności (oprócz najmniejszej wartości własnej, tu w obliczeniach wyszły identyczne z dokładnością do $10^{-10}$)

# Uruchomienie programów

1. Aby uruchomić program w języku c na linuxie należy przejść do katalogu w którym znajdują się pliki i wpisać w terminalu komendę: –> gcc <nazwaProgramu>.c -o <nazwaProgramu>.x -lgsl -lgslcblas -lm

2. A następnie: –> ./<nazwaProgramu>.x

```
#include <stdio.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <stdlib.h>
#include <math.h>


double mnozenieSkalarne(double *tab, double *tab2,int size){
double sum=0;
for(int i=0;i<size;++i){
sum+=tab[i]*tab2[i];
}
return sum;
}

double liczenieNormy(double *tab,int size){
double norma = 0;
double temp=0;
int i=0;
for(i;i<size;++i){
temp=tab[i]*tab[i];
norma+=temp;
}
norma=sqrt(norma);
return norma;
}


int main(){
  int counter=0;
double a[] = { 1,2,3,2,4,5,3,5,-1};
int i=0,j=0,n=0;
gsl_matrix *Q =gsl_matrix_alloc(3,3);
gsl_matrix *R =gsl_matrix_alloc(3,3);
gsl_matrix_view m = gsl_matrix_view_array (a, 3, 3);
int N = 3;
double lambdaN[]={0,0,0};
double lambdaN_1[]={0,0,0};
double czyWszystkie=0;
double epsilon=0.00000001;

void qrDecomp(){
gsl_vector *tau = gsl_vector_alloc (3);
gsl_linalg_QR_decomp(&m.matrix, tau);
gsl_linalg_QR_unpack(&m.matrix,tau,Q,R);

}

while(1){
qrDecomp();
```

```
//RQ
gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,1.0,R,Q,0.0,&m.matrix);
  for(n=0;n<3;++n){
lambdaN_1[n]=lambdaN[n];
for(j=0;j<3;++j){
 if(n==j)lambdaN[j]=gsl_matrix_get(&m.matrix,n,j);
}
  }

czyWszystkie=0;
for(n=0;n<3;++n){
if(fabs(lambdaN_1[n]-lambdaN[n])<epsilon) czyWszystkie++;
}
counter++;
if(czyWszystkie==3) break;
}

printf("Ilosc iteracji %d \n",counter);
printf(" Wartosci wlasne \n");
for(i=0;i<3;++i){
 printf(" %.10f \n",lambdaN[i]);
}
  return 0;
}
```