Ruprecht-Karls-Universität Heidelberg

Eigenwertproblematik

Felix Schweizer

Bericht des wöchentlichen Fortschritts bei der Bearbeitung des Themas *Eigenwertptoblematik*, welches unter der Leitung von *Prof. Dr. P. Bastian* innerhalb eines Softwarepraktikums für Anfänger an der Universität Heidelberg (WS 2011/2012) angeboten wurde

---

Einleitung

Ziel des Softwarepraktikums ist es, numerische Algorithmen für C++ zu implementieren. Es handelt sich dabei um Algorithmen zur Bestimmung von Eigenwerten reeller (evtl auch komplexer), quadratischer Matrizen.

Die Eigenwerte $\lambda_i = 1, \ldots, n$ einer $(n \times n)$-Matrix A erfüllen alle die Gleichung

$$A\vec{x} = \lambda\vec{x}$$

wobei $\vec{x}$ ein Eigenvektor von A ist.
Die Bestimmung solcher Eigenwerte spielt unter anderem in der Physik bei Schwingungsproblemen, in der Statistik bei Varianzproblemen und beim Lösen von Differentialgleichungen eine wichtige Rolle.

Um die Eigenwerte, kurz EW einer Matrix A analytisch zu bestimmen, muss man die Nullstellen des *charakteristischen Polynoms* $P(\lambda) :=\det(A - \lambda I)$ bestimmen. I ist die Einheitsmatrix. Dazu muss man jedoch auch die Koeffizienten des Polynoms bestimmen. Außerdem ist das Bestimmen von Nullstellen eines Polynoms höheren Grades analytisch sehr schwer und numerisch sehr instabil.
Aus diesem Grund werden die EW einer Matrix auf andere numerische Art gelöst, welche im Folgenden beschrieben sind.

Einarbeitung in die Theorie **Zeitraum: 19.10. – 26.10.2011**

Zur Einarbeitung in die theoretischen Grundlagen der Eigenwertproblematik wurde das Buch *Numerische Mathematik* (5. Auflage) von `Hans-Rudolf Schwarz` und `Norbert Köckler` benutzt. Ziel war es das Thema „Eigenwertprobleme“ (S.215 – S.270) zu wiederholen und die Theorie hinter den vorgestellen Algorithmen zu verstehen.
Das Bearbeiten des Themas erfolgte bis Seite 253 **5.5.3 QR-Doppelschritt, komplexe Eigenwerte**. Da zunächst nur mit dem *klassischen Jacobi- Verfahren* gearbeitet werden soll, werden im Folgenden die hinter diesem Verfahren stehenden Algorithmen vorgestellt.

Das *klassische Jacobi-Verfahren* arbeitet nur mit rellen, quadratischen Matrizen. Für eine relle, quadratische Matrix A der Ordnung n gilt:

$$X^{-1}AX = D$$

wobei X eine orthogonale Matrix ist, welche die Eigenvektoren von A als Spalten besitzt. D ist eine Diagonalmatrix, welche die EW $\lambda_i\, i = 1, \ldots, n$ von A auf der Diagonalen stehen hat.
Die Idee des Verfahrens ist es, die Matrix A durch orthogonale Transformationen auf Diagonalgestalt zu bringen. Dazu verwendet man einfache Transformationsmatrizen $U(p, q; \varphi)$ der Form

$$\begin{array}{ll} u_{ii} = 1,\ i \neq p, q & u_{pp} = u_{qq} = \cos\varphi \\ u_{pq} = \sin\varphi & u_{qp} = -\sin\varphi \\ u_{ij} = 0 \quad \text{sonst} & \end{array}$$

Diese Matrizen U sind orthogonal und entsprechen, als lineare Transformation aufgefasst, einer Drehung um den Winkel $-\varphi$ in der zweidimensionalen Ebene, welche durch die $p-$te und $q-$te Koordinatenrichtung aufgespannt wird. Deshalb nennt man $U(p, q; \varphi)$ auch Rotationsmatrix.

$$U(p,q;\varphi) = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & \cos\varphi & & & \sin\varphi & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & -\sin\varphi & & & \cos\varphi & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \leftarrow p \\ \\ \\ \\ \leftarrow q \\ \\ \\ \end{matrix}$$

Nun wird zunächst $A' = U^T A (= U^{-1} A)$ gebildet. Die Matrixeinträge ändern sich wie folgt

$$\left.\begin{aligned} a'_{pj} &= a_{pj}\cos\varphi - a_{qj}\sin\varphi \\ a'_{qj} &= a_{pj}\sin\varphi + a_{qj}\cos\varphi \\ a'_{ij} &= a_{ij} \text{ für } i \neq p, q \end{aligned}\right\} j = 1, \ldots, n$$

Anschließend wird $A'' = A'U$ gebildet:

$$\left.\begin{aligned} a''_{ip} &= a'_{ip}\cos\varphi - a'_{iq}\sin\varphi \\ a''_{iq} &= a'_{ip}\sin\varphi + a'_{iq}\cos\varphi \\ a''_{ij} &= a'_{ij} \text{ für } j \neq p, q \end{aligned}\right\} i = 1, \ldots, n$$

Für die Elemente, welche zweimal verändert werden gilt

$$\begin{aligned} a''_{pp} &= a_{pp}\cos^2\varphi - 2a_{pq}\cos\varphi\sin\varphi + a_{qq}\sin^2\varphi \\ a''_{qq} &= a_{pp}\sin^2\varphi + 2a_{pq}\cos\varphi\sin\varphi + a_{qq}\cos^2\varphi \\ a''_{pq} &= a''_{qp} = (a_{pp} - a_{qq})\cos\varphi\sin\varphi + a_{pq}(\cos^2\varphi - \sin^2\varphi) \end{aligned}$$

Die Grundidee des *Jacobi-Verfahrens* besteht nun darin, in jedem einzelnen Transformationsschritt das momentan absolut größte Paar von Außendiagonalelementen $a_{pq} = a_{qp}$ auf null zu „drehen“ .
Daraus ergibt sich folgende Bedingung

$$(a_{pp} - a_{qq})\cos\varphi\sin\varphi + a_{pq}(\cos^2\varphi - \sin^2\varphi) = 0$$

dies führt zu

$$\cot(2\varphi) = \frac{\cos^2\varphi - \sin^2\varphi}{2\cos\varphi\sin\varphi} = \frac{a_{qq} - a_{pp}}{2a_{qp}} =: \Theta$$

(Existiert, da $a_{pq} \neq 0$ sonst nichts zu transformieren wäre).
Hieraus lassen sich nun die Werte $\cos\varphi$ und $\sin\varphi$ berechnen. Der Winkel φ an sich ist ja nicht weiter interessant.

$$t := \tan\varphi = \begin{cases} \frac{1}{\Theta + \operatorname{sgn}(\Theta)\sqrt{\Theta^2+1}} & \text{für } \Theta \neq 0 \\ 1 & \text{für } \Theta = 0 \end{cases}$$

$$\cos\varphi = \frac{1}{\sqrt{1+t^2}}, \qquad \sin\varphi = t\cos\varphi$$

Zusammen mit $r := \frac{\sin\varphi}{1+\cos\varphi}$ ergeben sich nun die endgültigen Formeln der neuen Matrixkomponenten

$$\mathbf{a''_{pp}} = \mathbf{a_{pp}} - \mathbf{a_{qp}} \tan\varphi \tag{1}$$
$$\mathbf{a''_{qq}} = \mathbf{a_{qq}} + \mathbf{a_{qp}} \tan\varphi \tag{2}$$
$$\mathbf{a'_{pj}} = \mathbf{a_{pj}} - \sin\varphi(\mathbf{a_{qj}} + \mathbf{r} \cdot \mathbf{a_{pj}}) \tag{3}$$
$$\mathbf{a'_{qj}} = \mathbf{a_{qj}} + \sin\varphi(\mathbf{a_{pj}} - \mathbf{r} \cdot \mathbf{a_{qj}}) \tag{4}$$
$$\mathbf{a''_{ip}} = \mathbf{a'_{ip}} - \sin\varphi(\mathbf{a'_{iq}} + \mathbf{r} \cdot \mathbf{a'_{ip}}) \tag{5}$$
$$\mathbf{a''_{iq}} = \mathbf{a'_{iq}} + \sin\varphi(\mathbf{a'_{ip}} - \mathbf{r} \cdot \mathbf{a_{iq}}) \tag{6}$$

$(i, j = 1, \ldots, n)$.

Das *Klassische Jacobi-Verfahren* ist also eine Folge $A^{(k)}$, $A^{(0)} = A$ mit der Vorschrift

$$A^{(k)} = U_k^T A^{(k-1)} U_k, \quad k = 1, 2, \ldots$$

wobei im $k-$ten Schritt das absolut größte Nicht-diagonalelement $a_{qp}^{(k-1)}$ der Matrix $A^{(k-1)}$ durch eine Jacobirotation $U_k = U(p, q; \varphi_k)$ zu null gemacht wird.
$A^{(k)}$ enthält dabei die Komponenten a''_{ij}, Matrix $A^{(k-1)}$ die Komponenten a_{ij} (für $i, j = 1, \ldots, n$) der Formeln $(1), (2), \ldots, (6)$.

Obwohl im Allgemeinen in einem der nächsten Schritte des Verfahrens diese Element wieder ungleich null wird gilt:

Die Folge der zueinander ähnlichen Matrizen $A^{(k)}$ des klassischen Jacobiverfahrens konvergiert gegen eine Diagonalmatrix D.

Ein Beweis hierzu findet sich auf Seite 222f des Buches *Numerische Mathematik*, welches zu Anfang bereits erwähnt wurde.

Rechenaufwand

Der Rechenaufwand des $klassischen - Jacobiverfahrens$ setzt sich aus der Bestimmung der Werte $\tan\varphi, \sin\varphi, \cos\varphi, \cot(2\varphi), r$ und den Formeln (1) bis (6) zusammen. Erstere benötigen 7 multiplikative Operationen und 2 Quadratwurzel-Auswertungen, für die Formeln (1) bis (6) werden $4(n-1)+2$ multiplikative Operationen benötigt.
Für eine Rotation sind somit $4n + 5$ multiplikative Operationen und 2 Quadratwurzel-Auswertungen vonnöten.
Ein weiterer wichtiger Teil des Rechenaufwandes ist jedoch den Vergleichsoperationen zu zu schreiben. Da in jedem Schritt des Verfahrens das größte Nicht-Diagonalelement gefunden werden muss, sind pro Rotation $(n^2-n)/2-1$ Vergleichsoperationen nötig! Aus diesem Grund wird in der Praxis auch das sogenannte $ZyklischeJacobi - Verfahren$ verwendet, welches immer wieder in gleicher Reihenfolge die Elemente der unteren Nebendiagonalen eliminiert. Aus Zeitgründen wird hier jedoch nicht näher auf dieses Verfahren eingegangen.

Wiederholung C++ **Zeitraum: 26.10. – 03.11.2011**

Da die numerischen Algorithmen mit der Programmiersprache C++ implementiert werden sollen, wurden im oben angegebenen Zeitraum die Grundlagen von C++ wiederholt. Verwendet wurde dabei das Buch *C++ Kurs, technisch orientiert* (4. Auflage) von `Professor Dipl.-Ing. Günter Schmitt`. Die Wiederholung erfolgte bis Seite 181. die bis dahin behandelten Themen sind:

1. Einführung
2. Grundlagen
   - Vereinbarungen
   - Programmierung von Formeln
   - Ein- und Ausgabe von Daten
3. Programmstrukturen
   - Vergleichsausdrücke
   - Programmverzweigungen
   - Programmschleifen
   - Hinweise auf Fehlerquellen
4. Zeiger
   - Anwendung von Zeigern
   - Hinweise auf Fehlermöglichkeiten
5. Funktionen
   - Unterprogrammtechnik
   - Funktionen mit Rückgabewert oder Referenzparameter
   - Rekursiver Aufruf von Funktionen
   - Auslagern von Funktionen
   - Makros
6. Felder (Arrays)
   - Ein- und mehrdimensionale Felder
   - Felder als Parameter und Funktionsergebnis

Desweiteren sollte ein kleines erstes C++ Programm geschrieben werden, welches eine Matrix A der Form

$$A = \begin{array}{c} \begin{matrix} 1 & 2 & \cdots & m \end{matrix} \\ \begin{pmatrix} B_n & -I_n & & \\ -I_n & B_n & \ddots & \\ & \ddots & \ddots & -I_n \\ & & -I_n & B_n \end{pmatrix} \end{array} \in \mathbb{R}^{N \times N}, \qquad B_n = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

erzeugt. Es gilt $N = n \cdot m$, dabei sind n, m integer Werte, die vom Benutzer eingegeben werden sollen.
Eine solche Matrix A ist natürlich symmetrisch und positiv definit, was sich leicht zum Beispiel mit den *Gerschgorin-Kreisen* verifizieren lässt. Auf eine solche Matrix lässt sich das *Klassische Jacobi-Verfahren* folglich anwenden. Zusätzlich lassen sich bei einer Matrix dieser speziellen Form die EW auch analytisch bestimmen und man bekommt dadurch eine gute Kontrolle des *Jacobi-Verfahrens*.
Die analytische Bestimmung der EW wird zu einem späteren Zeitpunkt genauer behandelt.

Das erstellte C++ Programm:

```
#include <iostream>
#include <iomanip>

int main()
{
  int n,m,i,j,p;
  std::cout << "Anzahl Bloecke: ";
  std::cin >> m;

  std::cout << "Dimension pro Block: ";
  std::cin >> n;
  int r[n*m][n*m];

  for(i=0; i<(n*m); i++)
           for(j=0; j<(n*m); j++)
           {
           if(i==j) r[i][j]=4;
           else if(i==(j+1) || j==(i+1) || i==(j+n) || j==(i+n) )
              {r[i][j]=-1; }
           else {r[i][j]=0;}
           }

for (i=1; i<m; i++)
  {
  A[i*n][i*n-1]=A[i*n-1][i*n]=0;
  }
  // matrix ausgeben

  for (i=0; i<(n*m); i++)
  {
    std::cout << "\n";
    for (j=0;j<(n*m);j++)
        {
        std::cout << std::setw(3) << r[i][j];
        }
   }
   std::cout << "\n";
   std::cin >> p;
   return 0;
}
```

Die Zeile `std::cin >> p;` ist überflüssig und diente mir nur zur Kontrolle der entstandenen Matrix.

Einarbeitung in hdnum **Zeitraum: 03.11. – 10.11.2011**

„hdnum“ ist eine C++-Bibliothek welche einige wichtige numerische Algorithmen der Numerik 0 enthält. Da das Ziel des Software Praktikums ist, diese Bibliothek zu erweitern darf natürlich mit bereits erstellten Programmen aus hdnum gearbeitet werden.
Aufgabe in dieser Woche war es, sich mit den Grundfunktionen aus hdnum vertraut zu machen und den C++ Code von letzter Woche mithilfe dieser Funktionen zu modifizieren.

Einige wichtige Standardfunktionen aus hdnum mit Erklärung und der modifizierte C++ Code auf den nächsten Seite. Leider kompiliert das C++ Programm momentan noch nicht bei meinem Rechner, so dass eine saubere Ausgabe der Matrix noch nicht kontrolliert werden konnte.

`DenseMatrix<double> A(n,m,p);`	erzeugt eine $n \times m$ Matrix A. Komponenten haben alle Wert p. Defaultwert von p ist 0.
`Vector<double> x(n,p);`	erzeugt Vektor x der Dimension n. Komponenten von x haben alle Wert p, Defaultwert ist 0
`A.colsize();`	gibt die Spaltenanzahl der Matrix A wieder
`A.rowsize();`	gibt die Zeilenanzahl der Matrix A wieder
`C.mm(A,B);`	entspricht $C = A * B$, A, B, C müssen aber bereits definiert sein
`A.mv(y,x);`	entspricht $y = A * x$, A, x, y bereits definiert
`identity(A);`	macht aus Matrix A Einheitsmatrix
`A(2,3)=42;`	entspricht a[2][3]=42;
`A/=s;` bzw. `A*=s;`	A wird mit Skalar s dividiert bzw. multipliziert
`A=m;`	alle Komponenten von A werden auf Wert m gesetzt
`x=m;`	alle Komponenten von x werden auf Wert m gesetzt
`spd(A);`	erzeugt positiv definite, symmetrische Matrix aus A
`A.sc(x,3);`	ersetzt 4. Spalte von A mit Vektor x
`A.sr(x,1);`	ersetzt 2. Reihe von A mit Vektor x
`C.umm(A,B);`	entspricht $C = C + A * B$
`A.umv(y,s,x);`	entspricht $y = s * A * x$ s Skalar, x Vektor A Matrix
`A.umv(y,x);`	entspricht $y = y + A * x$ y, x Vektor, A Matrix
`vandermonde(A,x);`	erzeugt aus A mit Vektor x eine Vandermonde Matrix
`y*x;`	gibt als Wert das Skalarprodukt von y und x wieder
`x.two_norm();`	entspricht $\|x\|_2$
`fill(x,m,d);`	definiert Vektor x neu, x[0]=m, nächster Wert jeweils um d erhöht bis Ende x
`unitvector(x,3);`	setzt alle Werte von x auf null außer x[3]=1
`A.update(s,B);`	entspricht $A = A + s * B$ wobei s Skalar
`A.norm_infty();`	gibt Reihensummennorm von A wieder
`A.norm_1();`	gibt Spaltensummennorm von A wieder
`x.size();`	gibt Länge des Vektors x wieder

```
#include <iostream>
#include <iomanip>
#include "hdnum.hh"

int main()
{
  int n,m,i,j,p;
  while(1) //Kontrollschleife
  {
           std::cout << "Anzahl Bloecke: ";
           std::cin >> m;  std::cin.seekg(0);
           if (std::cin.fail() || m<1)
           { std::cout << "Eingabefehler"; std::cin.clear(); continue; }
           else break;
  }
  while(1) //nächste Kontrollschleife
  {
           std::cout << "Dimension pro Block: ";
           std::cin >> n;  std::cin.seekg(0);
           if (std::cin.fail() || n<1)
           { std::cout << "Eingabefehler"; std::cin.clear(); continue; }
           else break;
  }
  hdnum::DenseMatrix<double> A(n*m,n*m);
  hdnum::identity(A);
  A *=4; // Matrix hat jetzt 4er auf der diagonalen

  for(i=0; i<(n*m); i++)
           for(j=0; j<(n*m); j++)
           {
           if(i==(j+1) || j==(i+1) || i==(j+n) || j==(i+n) )
              {A(i,j)=-1; }
           }
for (i=1; i<m; i++)
  {
  A(i*n,i*n-1)=A(i*n-1,i*n)=0;
  }
//Matrix fertig erstellt
  // matrix ausgeben
  A.iwidth(2);
  A.width(n*m);

   std::cout << A;
   std::cout << "\n";
   std::cin >> p;
   return 0;
}
```

Jacobi-Verfahren als C++ Code **Zeitraum: 17.11. – 24.11.2011**

In dieser Woche war die Aufgabe sich in Microsoft Visual C++ 2010 einzuarbeiten, um dann (endlich) das Jacobi Verfahren für Matrizen der Form wie sie im Kapitel **Wiederholung C++** vor gestellt wurden als C++ Code zu implementieren.

C++ Code des Jacobiverfahrens:

```
#define TIMER_USE_STD_CLOCK
#include <iostream>
#include <cmath>
#include <math.h>
#include <iomanip>
#include "hdnum.hh"
#include <fstream>


using namespace hdnum;

void drehung(DenseMatrix<double> &A, DenseMatrix<double> &U, int p, int q)
// symm. Matrix wird um phi gedreht A(p,q)=0
// U speichert die Rotationsmatrizen
{
size_t N=A.colsize();
double t;
double v =(A[q][q]-A[p][p])/(2*A[q][p]); //v=cot(2 phi)
if (v==0)  t=1;
else if (v<0)  t=1/(v-sqrt(v*v+1)); // t=tan( phi)
else  t =1/(v+sqrt(v*v+1));

double c=1/sqrt(1+t*t); //c=cos(phi)
double s=t*c; //s=sin(phi)
double r=s/(1+c); //hilfsvariable r
A[p][p]=A[p][p]-A[q][p]*t; //spezielle matrix einträge (eckpunkte)
  // werden gedreht
A[q][q]=A[q][q]+A[q][p]*t;
A[p][q]=A[q][p]=0;


for (int j=0; j<N; j++)
{
if (j!=p && j!=q) // diese werte wurden bereits gedreht
{
double h=A[j][p];
A[j][p]=A[p][j]=A[j][p]-s*(A[j][q]+r*A[j][p]);
A[j][q]=A[q][j]=A[j][q]+s*(h-r*A[j][q]);
}
```

```
}
// matrix A jetzt gedreht, so dass A(p,q)=A(q,p)=0

for (int j=0; j<N; j++) // U wird aufgebaut
{
double h=U[j][p];
U[j][p]=U[j][p]-s*(U[j][q]+r*U[j][p]);
U[j][q]=U[j][q]+s*(h-r*U[j][q]);
}



}

double maxfinden( const DenseMatrix<double> &A, int &p, int &q) // funktion sucht
      // maximum von A
{
size_t N=A.colsize();
double max=0;
for (int j=0; j<(N-1); j++) // letzte spalte uninteressant
{
for (int i=(j+1); i<N; i++)
{
if (abs(A[i][j]) > max)
{
max=abs(A[i][j]);
p=j; // koordinaten des neuen maximums
q=i;
}
}
}
return max;
}


void jacob(DenseMatrix<double> &A,DenseMatrix<double> &U,double tol)
   // jacobiverfahrensfunktion
{
size_t N=A.colsize();
identity(U);
while (1)
{
int p=5;
int q=3;
double max=maxfinden(A,p,q);
if (max > tol)
{
```

```
drehung(A,U,p,q);
}
else break;
}
}

int main()
{
//ausgabe in file
  std::ofstream fout;
  fout.open("Matrizen.txt");

  int n,m,i,j,k;
  double e;
  while(1) //Kontrollschleife
  {
           std::cout << "Anzahl Bloecke: \n";
           std::cin >> m;  //std::cin.seekg(0);
           if (std::cin.fail() || m<1)
           { std::cout << "Eingabefehler\n"; std::cin.clear(); continue; }
           else break;
  }
  while(1)
  {
           std::cout << "Dimension pro Block: \n";
           std::cin >> n;  //std::cin.seekg(0);
           if (std::cin.fail() || n<1)
           { std::cout << "Eingabefehler"; std::cin.clear(); continue; }
           else break;
  }
  while(1)
  {
           std::cout << "Toleranz eingeben:  ";
           std::cin >> e;  //std::cin.seekg(0);
           if (std::cin.fail() || e>1)
           { std::cout << "Eingabefehler"; std::cin.clear(); continue; }
           else break;
  }
  double tol=e;
  hdnum::DenseMatrix<double> A(n*m,n*m);
  hdnum::DenseMatrix<double> U(n*m,n*m);
  hdnum::identity(A);
  A *=4; // Matrix hat jetzt 4er auf der diagonalen

  for(i=0; i<(n*m); i++)
  {
           for(j=0; j<(n*m); j++)
```

```
            {
            if(i==(j+1) || j==(i+1) || i==(j+n) || j==(i+n) )
               {A[i][j]=-1; }
            }
}

for (i=1; i<m; i++)
  {
  A(i*n,i*n-1)=A(i*n-1,i*n)=0;
  }
//Matrix fertig erstellt

  jacob(A,U,tol); //Verfahren wird ausgeführt
A.iwidth(2);
  U.iwidth(2);
   A.scientific(false); // fixed point representation for all DenseMatrix objects
  U.scientific(false);
  A.width(8);
  U.width(8);
  A.precision(3);
  U.precision(3);
   std::cout << A << U;
   fout << A << U;
   fout.close();
   std::cout << "\n";
   std::cin >> k;
   return 0;
}
```

Die erste Zeile `#define...` dient lediglich dazu, das Aufrufen einer weiteren Unterdatei von densematrix.hh zu unterdrücken. Man schaue sich dazu den Quellcode von densematrix.hh an.

Jacobi für einfache Matrix **Zeitraum: 25.11. – 01.12.2011**

In dieser Woche bestand die Aufgabe das bereits programmierte Jacobi-Verfahren auf eine Matrix der Form

$$\begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}$$

anzuwenden, die Iterationsschritte zu zählen und die korrekten Eigenwerte analytisch mit zu bestimmen.
Dies geschieht mit der Formel:

$$4\sin^2\left(\frac{\mu\pi h}{2}\right)$$

wobei $1 \leq \mu \leq n$ und $h = \frac{1}{n+1}$ (n Dimension).

Der C++ Code:

```
#define TIMER_USE_STD_CLOCK
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
#include <math.h>
#include <iomanip>
#include "hdnum.hh"
#include <fstream>

using namespace hdnum;

void drehung(DenseMatrix<double> &A, DenseMatrix<double> &U, int p, int q)
 // symm. Matrix wird um phi
{ // U speichert die Rotationsmatrizen
size_t N=A.colsize();
double t;
double v =(A[q][q]-A[p][p])/(2*A[q][p]); //v=cot(2 phi)
if (v==0)   t=1;
else if (v<0)   t=1/(v-sqrt(v*v+1)); // t=tan( phi)
else  t =1/(v+sqrt(v*v+1));

double c=1/sqrt(1+t*t); //c=cos(phi)
double s=t*c; //s=sin(phi)
double r=s/(1+c); //hilfsvariable r
A[p][p]=A[p][p]-A[q][p]*t; //spezielle matrix einträge (eckpunkte) werden gedreht
A[q][q]=A[q][q]+A[q][p]*t;
A[p][q]=A[q][p]=0;
```

```
for (int j=0; j<N; j++)
{
if (j!=p && j!=q) // diese werte wurden bereits gedreht
{
double h=A[j][p];
A[j][p]=A[p][j]=A[j][p]-s*(A[j][q]+r*A[j][p]);
A[j][q]=A[q][j]=A[j][q]+s*(h-r*A[j][q]);
}
}

// matrix A jetzt gedreht, so dass A(p,q)=A(q,p)=0

for (int j=0; j<N; j++) // U wird aufgebaut
{
double h=U[j][p];
U[j][p]=U[j][p]-s*(U[j][q]+r*U[j][p]);
U[j][q]=U[j][q]+s*(h-r*U[j][q]);
}



}

double maxdiagfinden( const DenseMatrix<double> &A)
 // funktion such max. diag. element  von A
{
size_t N=A.colsize();
double maxdiag=0;
for (int j=0; j<N; j++)
{
if (abs(A[j][j]) > maxdiag)
{
maxdiag=abs(A[j][j]);
}
}
return maxdiag;
}


double maxfinden( const DenseMatrix<double> &A, int &p, int &q)
     // funktion such maximum von A
{
size_t N=A.colsize();
double max=0;
for (int j=0; j<(N-1); j++) // letzte spalte uninteressant
```

```
{
for (int i=(j+1); i<N; i++)
{
if (abs(A[i][j]) > max)
{
max=abs(A[i][j]);
p=j; // koordinaten des neuen maximums
q=i;
}
}
}


return max;
}


void jacob(DenseMatrix<double> &A,DenseMatrix<double> &U,double tol,int &k)
// jakobiverfahrensfunktion
{
size_t N=A.colsize();
identity(U);
while (1)
{
int p=0;
int q=0;
double max=maxfinden(A,p,q);
double maxdiag=maxdiagfinden(A);
if ((max/maxdiag) > tol)
{
drehung(A,U,p,q);
k=k+1;
}
else break;
}

 }


int main()
{
//ausgabe in file
  std::ofstream fout;
  fout.open("Matrizen2.txt");

  int n,i,j;
  int k=0;
```

```
double e,hilf;
while(1) //Kontrollschleife
{
         std::cout << "Dimension Matrix: \n";
         std::cin >> n;  //std::cin.seekg(0);
         if (std::cin.fail() || n<1)
         { std::cout << "Eingabefehler\n"; std::cin.clear(); continue; }
         else break;
}
while(1)
{
         std::cout << "Toleranz eingeben:  ";
         std::cin >> e;  //std::cin.seekg(0);
         if (std::cin.fail() || e>1)
         { std::cout << "Eingabefehler"; std::cin.clear(); continue; }
         else break;
}
double tol=e; // Toleranzfestlegen
hdnum::DenseMatrix<double> A(n,n);
hdnum::DenseMatrix<double> U(n,n);
hdnum::identity(A);
A *=2; // Matrix hat jetzt 2er auf der diagonalen

for(i=0; i<n; i++)
{
         for(j=0; j<n; j++)
         {
         if(i==(j+1) || j==(i+1) )
            {A[i][j]=-1; }
         } //Matrix fertig erstellt
}

A.iwidth(2);
A.scientific(false);
A.width(8);
A.precision(3);
std::cout<< "Eingabe Matrix A: \n" << A << "\n";
fout<< "Eingabe Matrix A: \n" << A<< "\n";

jacob(A,U,tol,k);

double *a= new double [n];
for (i=0; i<n; i++) // diag. elemente von A speichern
{
a[i]=A[i][i];
}
```

```
for (i=0; i<(n-1); i++) // feld a sortieren (aufsteigende werte)
{
for(j=i+1; j<n; j++)
{
if(a[i]>a[j])
{
hilf=a[i];
a[i]=a[j];
a[j]=hilf;
}
}
}
double *ew=new double [n]; // korrekte EW berechnen
for(i=1; i<(n+1); i++)
{
ew[i-1]=4*std::pow((std::sin(M_PI*i/(2*(n+1)))),2);
}

double *abw=new double [n]; //absolute Abweichungen berechnen
for (i=0; i<n; i++)
{
abw[i]=abs(a[i]-ew[i]);
}

A.iwidth(2);
U.iwidth(2);
A.scientific(false);
U.scientific(false);
A.width(8);
U.width(8);
A.precision(3);
U.precision(3);
std::cout << "Diagonalisierte Matrix A: \n" << A << "\n";
std::cout<<" Drehmatrix: \n" << U <<"\n";
 fout <<"Diagonalisierte Matrix A: \n" << A << "\n";
 fout <<" Drehmatrix: \n"<< U<< "\n";
 //fout.close();
 std::cout << "Anzahl Iterationen: \n" << k << "\n";
 fout << "Anzahl Iterationen: \n" << k << "\n";
 std::cout<< "Eigenwerte geordnet: \n";
 fout << "Eigenwerte geordnet: \n";
 for(i=0; i<n; i++)
 {
 std::cout<< std::setw(7)<< a[i];
 fout <<std::setw(7)<< a[i];
 }
 std::cout<< "\n korrekte Eigenwerte geordnet: \n";
```

```
   fout << "\n korrekte Eigenwerte geordnet: \n";
   for(i=0; i<n; i++)
   {
   std::cout<< std::setw(7)<< ew[i];
   fout <<std::setw(7)<< ew[i];
   }
   std::cout << "\n absolute Abweichung: \n";
   fout <<  "\n absolute Abweichung: \n";
   for(i=0; i<n; i++)
   {
   std::cout<< std::setw(7)<< abw[i];
   fout << std::setw(7)<< abw[i];
   }
   std::cout<< "\n";
   fout.close();
    return 0;
}
```

Beispiel einer Ausgabe für $n = 6$:

```
Eingabe Matrix A:

           0       1       2       3       4       5
  0    2.000  -1.000   0.000   0.000   0.000   0.000
  1   -1.000   2.000  -1.000   0.000   0.000   0.000
  2    0.000  -1.000   2.000  -1.000   0.000   0.000
  3    0.000   0.000  -1.000   2.000  -1.000   0.000
  4    0.000   0.000   0.000  -1.000   2.000  -1.000
  5    0.000   0.000   0.000   0.000  -1.000   2.000

Diagonalisierte Matrix A:

           0       1       2       3       4       5
  0    2.445   0.000  -0.000  -0.000  -0.000  -0.000
  1    0.000   1.555   0.000   0.000   0.000   0.000
  2   -0.000   0.000   3.802  -0.000   0.000   0.000
  3   -0.000   0.000  -0.000   0.198   0.000   0.000
  4   -0.000   0.000   0.000   0.000   3.247   0.000
  5   -0.000   0.000   0.000   0.000   0.000   0.753
```

Drehmatrix:

	0	1	2	3	4	5
0	0.521	0.521	0.232	0.232	-0.418	-0.418
1	-0.232	0.232	-0.418	0.418	0.521	-0.521
2	-0.418	-0.418	0.521	0.521	-0.232	-0.232
3	0.418	-0.418	-0.521	0.521	-0.232	0.232
4	0.232	0.232	0.418	0.418	0.521	0.521
5	-0.521	0.521	-0.232	0.232	-0.418	0.418

Anzahl Iterationen:
27
Eigenwerte geordnet:
0.198 0.753 1.555 2.445 3.247 3.802
korrekte Eigenwerte geordnet:
0.198 0.753 1.555 2.445 3.247 3.802
absolute Abweichung:
0.000 0.000 0.000 0.000 0.000 0.000

QR-Zerlegung **Zeitraum: 01.12. – 08.12.2011**

Die Aufgabe dieser Woche war, die **QR-Zerlegung** für beliebige quadratische Matrizen zu programmieren. Später in desem Praktikum wird diese Zerlegung für einen weiteren Algorithmus zur Berechnung von Eigenwerten verwendet und auf die bereits hier bekannten symmetrischen und insbesondere quadratischen Matrizen angewandt.

Der QR-Algorithmus:

Grundlage der QR-Zerlegung ist der Satz:

Jede quadratische Matrix **A** *lässt sich als Produkt einer orthogonalen Matrix* **Q** *und einer Rechtsdreiecksmatrix* **R** *in der Form*

$$\mathbf{A}=\mathbf{QR}$$

darstellen. Man bezeichnet die Faktorisierung als QR-Zerlegung *der Matrix* **A**.

Die Existenz dieser Zerlegung zeigt man mit Hilfe der bereits eingeführten Drehmatrizen $U(p,q;\varphi)$. Wie im Kapitel **Einarbeitung in die Theorie** bereits gezeigt, werden bei der Bildung von

$$A' = U^T A$$

nur die Komponenten der p-ten und q-ten Zeile linear kombiniert:

$$\left.\begin{aligned} a'_{pj} &= a_{pj}\cos\varphi - a_{qj}\sin\varphi \\ a'_{qj} &= a_{pj}\sin\varphi + a_{qj}\cos\varphi \\ a'_{ij} &= a_{ij} \text{ für } i \neq p,q \end{aligned}\right\} \; j = 1,\ldots,n \qquad (1)$$

Die Elemente der Matrix A werden nun sukzessive mit geeigneten Drehmatrizen eliminiert. Dies erfolgt zum Beispiel spaltenweise in der Reihenfolge

$$a_{21}, a_{31}, \ldots, a_{n1}, a_{32}, a_{42}, \ldots, a_{n2}, a_{43}, \ldots, a_{n,n-1}$$

wobei die Rotationsindexpaare entsprechend wie folgt gewählt werden:

$$(1,2),(1,3),\ldots,(1,n),(2,3),(2,4),\ldots,(2,n),(3,4),\ldots,(n-1,n)$$

Um das Element $a_{ij}^{(k-1)}$ der Matrix $\mathbf{A}^{(k-1)}$ zu eliminieren, wird eine $U_k(j,i;\varphi_k)$ Matrix zur Bildung von

$$A^{(k)} = U_k^T A^{(k-1)}, \qquad A^{(0)} = A$$

verwendet.
Aus der Forderung

$$\begin{aligned} a_{ij}^{(k)} = a_{jj}^{(k-1)}\sin\varphi_k + a_{ij}^{(k-1)}\cos\varphi_k &= 0 \\ \sin\varphi_k^2 + \cos\varphi_k^2 &= 1 \end{aligned}$$

lassen sich die Werte $\sin\varphi_k$ und $\cos\varphi_k$ berechnen. Dies geschieht mit der Formel

$$w := \operatorname{sgn}(a_{jj})\sqrt{a_{jj}^2 + a_{ij}^2} \tag{2}$$

$$\cos\varphi = \frac{a_{jj}}{w}, \qquad \sin\varphi = \frac{-a_{ij}}{w} \tag{3}$$

Ist $a_{ij}^{(k-1)} = 0$ so gilt natürlich $U_k = I$ (Einheitsmatrix).
Werden die Matrixelemente in obiger Reihenfolge auf Null gedreht, so erkennt man aus (1), dass ein einmal auf Null gedrehtes Element sich nicht mehr ändert. Nach $N = \frac{1}{2}n(n-1)$ Schritten ist also jedes Element a_{ij} $(i < j)$ der ursprünglichen Matrix A auf Null gedreht und die Matrix $A^{(N)}$ stellt eine obere Dreiecksmatrix R dar.
Es gilt also:

$$U_N^T U_{N-1}^T \dots U_2^T U_1^T A^{(0)} = R$$

Die Matrix $Q^T := U_N^T U_{N-1}^T \dots U_2^T U_1^T$ ist als Produkt orthogonaler Matrizen auch wieder orthogonal (insbesondere ist Q orthogonal).
Damit ist die Existenz der *QR-Zerlegung*

$$A^{(0)} = A = QR$$

gezeigt.

Der C++ Code bedient sich, wie der Existenzbeweis der *QR-Zerlegung*, der Drehmatrizen $U_k(j, i; \varphi_k)$ wobei analog zum eben gezeigten Beweis die Elemente der quadratischen Matrix A Spaltenweise eliminiert werden.
Später wird die *QR-Zerlegung* innerhalb eines Algorithmuses zur Bestimmung der Eigenwerte einer Matrix A angewandt, wobei A mindestens *Hessenberg-Form* zu haben hat. Der hier aufgeführte Code ist jedoch noch allgemein gehalten und kann auf jede quadratische Matrix A angewandt werden. Diese muss jedoch zu Beginn vom Benutzer zeilenweise eingegeben werden.

C++ Code der QR-Zerlegung:

```
#define TIMER_USE_STD_CLOCK
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
#include <math.h>
#include <iomanip>
#include "hdnum.hh"
#include <fstream>

using namespace hdnum;

void drehung(DenseMatrix<double> &A, DenseMatrix<double> &U,int p, int q)
// symm. Matrix wird um phi gedreht, A(q,p) = 0
// U speichert die Rotationsmatrizen
{
size_t N=A.colsize();
double w;
if (A(p,p)<0)
{
w=-sqrt(A(q,p)*A(q,p)+A(p,p)*A(p,p));
}
else w=sqrt(A(q,p)*A(q,p)+A(p,p)*A(p,p));
double c=A(p,p)/w; //c=cos(phi)
double s=-A(q,p)/w; //s=sin(phi)
double r=s/(1+c);

for (int j=p; j<N; j++)
{
double h=A[p][j];
A[p][j]=A[p][j]-s*(A[q][j]+r*A[p][j]);
A[q][j]=A[q][j]+s*(h-r*A[q][j]);
}
A(q,p)=0;

// matrix A jetzt gedreht, so dass A(q,p)=0

for (int j=0; j<N; j++) // U wird aufgebaut
{
double h=U[j][p];
U[j][p]=U[j][p]-s*(U[j][q]+r*U[j][p]);
U[j][q]=U[j][q]+s*(h-r*U[j][q]);
}
```

```
}

void zerlegung(DenseMatrix<double> &A,DenseMatrix<double> &U)
{
identity(U);
size_t N=A.colsize();
for (int i=0; i<(N-1); i++)
{
for (int j=i+1; j<N; j++)
{
if (A(j,i) !=0)
{
drehung(A,U,i,j);
}
}
}
}


int main()
{
//ausgabe in file
  std::ofstream fout;
  fout.open("MatrizenQR.txt");
// Eingabe der Matrix durch den Benutzer
  int n;
  std::cout<< "Dimension Matrix: \n"; std::cin>> n;
  hdnum::DenseMatrix<double> A(n,n);
  hdnum::DenseMatrix<double> U(n,n);
  hdnum::identity(U);
  for (int j=0; j<n; j++)
  {
  for (int i=0; i<n; i++)
  {
  std::cout<< "A("<<j <<","<<i <<" )= ";
  std::cin>> A(j,i); std::cout<< "\n";
  }
  }

  A.iwidth(2);
  A.scientific(false);
  A.width(10);
  A.precision(5);

  std::cout<< "eingegebene Matrix: \n" << A<< "\n";
  fout<< "eingegebene Matrix: \n" << A<< "\n";
```

```
  zerlegung(A,U);

  A.iwidth(2);
  U.iwidth(2);
  A.scientific(false);
  U.scientific(false);
  A.width(10);
  U.width(10);
  A.precision(5);
  U.precision(5);
  std::cout << "R: \n" << A << "\n";
  std::cout<<" Q: \n" << U <<"\n";
   fout <<"R: \n" << A << "\n";
   fout <<" Q: \n"<< U<< "\n";

  fout.close();
    return 0;
}
```

Beispielausgabe einer 4×4 - Matrix:

```
eingegebene Matrix:

             0          1          2          3
  0     0.00000    0.00000    0.00000    2.00000
  1    -2.00000    1.00000    0.00000    0.00000
  2     0.00000   -2.00000    1.00000    0.00000
  3     0.00000    0.00000   -2.00000    1.00000

R:

             0          1          2          3
  0     2.00000   -1.00000    0.00000    0.00000
  1     0.00000    2.00000   -1.00000    0.00000
  2     0.00000    0.00000    2.00000   -1.00000
  3     0.00000    0.00000    0.00000    2.00000

 Q:

             0          1          2          3
  0     0.00000    0.00000    0.00000    1.00000
  1    -1.00000    0.00000    0.00000    0.00000
  2     0.00000   -1.00000    0.00000    0.00000
  3     0.00000    0.00000   -1.00000    0.00000
```

QR-Algorithmus 1. Teil **Zeitraum: 09.12.2011 – 12.01.2012**

Wie bereits im vorherigen Kapitel erwähnt, kann der *QR-Algorithmus* zur berechnung von Eigenwerten einer Matrix A genutzt werden. A muss jedoch in Hessenbergform

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} & \dots & h_{1n} \\ h_{21} & h_{22} & h_{23} & h_{24} & \dots & h_{2n} \\ 0 & h_{32} & h_{33} & h_{34} & \dots & h_{3n} \\ 0 & 0 & h_{43} & h_{44} & \dots & h_{4n} \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & h_{n,n-1} & h_{nn} \end{pmatrix}$$

vorliegen und gegebenenfalls auf diese transformiert werden. Im folgenden wird jedoch mit der bereits bekannten Matrix

$$A = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}$$

gearbeitet. Diese befindet sich offensichtlich in Hessenbergform und ist zusätzlich symmetrisch. Da die Eigenwerte von A sich, wie im Kapitel *Jacobi für einfache Matrix* beschrieben, analytisch berechnen lassen, kann somit der zu schreibene C++ Code des *QR-Algorithmus* kontrolliert werden.

Sei H eine Matrix in Hessenbergform auf die die *QR-Transformation* angewandt wird. Die Folge von ähnlichen Matrizen:

$$H_k = Q_k R_k, \qquad H_{k+1} = R_k Q_k, \quad k = 1, 2, \dots$$

mit $H = H_1$ wird als einfacher **QR-Algorithmus von Francis** bezeichnet.

Für diese Folge gilt folgende Konvergenzaussage:

Sei λ_i die Eigenwerte von H mit $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_n|$. Seien $\vec{x}_i$ die Eigenvektoren von H, welche als Spalten die reguläre Matrix $X \in \mathbb{R}^{n,n}$ bilden. Falls für X^{-1} die LR-Zerlegung existiert, dann konvergieren die Matrizen H_k für $k \to \infty$ gegen eine Rechtsdreiecksmatrix, und es gilt $\lim_{k\to\infty} h_{ii}^{(k)} = \lambda_i$, $i = 1, \dots, n$. Hat H Paare von konjugiert komplexen Eigenwerte, derart dass ihre Beträge und die Beträge der reellen Eigenwerte paarweise verschieden sind, und existiert die (komplexe) LR-Zerlegung von X^{-1}, dann konvergieren die Matrizen H_k gegen eine Quasidreiecksmatrix.

Da in unserem Fall keine komplexen Eigenwerte auftauchen, wird hier noch nicht näher auf die Quasidreiecksmatrix eingegangen.
Bei der Matrix A konvergieren also alle Elemente $h_{i+1,i}^{(k)}$ gegen Null, wobei die Konvergenz jedoch sehr langsam sein kann. Eine Analyse des Konvergenzverhaltens dieser Elemente ergibt für hinreichend großes k:

$$|h_{i+1,i}^{(k)}| \approx |\frac{\lambda_{i+1}}{\lambda_i}|^k, \qquad i = 1, 2, \dots, n-1$$

(Quelle: Hans-Rudolf Schwarz, Norbert Köckler; Numerische Mathematik, 5.Auflage, Seite 246)

Die langsamere Konvergenz dieses Verfahrens gegenüber dem Jacobiverfahren, wird auch durch die Anzahl an Iterationen deutlich, welche im C++ Code mitgezählt werden.
Um die Konvergenz zu beschleunigen, kann jedoch vor jedem Schritt des *QR-Algorithmus* eine sogenannte *Spektralverschiebung* vorgenommen werden. Diese wird jedoch erst im Kapitel **QR-Algorithmus 2. Teil** behandelt.

Es gibt mehrere Arten den *QR-Algorithmus* mittels C++ zu implementieren. Die leichteste verwendet die Matrizenmultiplikation.

Achtung: Der Befehl `C.mm(A,U);` aus der hdnum-Bibliothek, welcher die Rechnung: C=A*U implementiert, kann nicht ohne Weiteres auf Windowssystemen verwendet werden!
Zur Ausgabe einer Fehlermeldung bei z.B. falscher Dimension der zu multiplizierenden Matrizen, verwendet `hdnum` vordefinierte Variablen. Das Kommentar an dieser Stelle lautet:

```
// the "format" the exception-type gets printed.  __FILE__ and
  // __LINE__ are standard C-defines, the GNU cpp-infofile claims that
  // C99 defines __func__ as well. __FUNCTION__ is a GNU-extension
```

Da der *Microsoft Visual C++*-Kompiler den Ausdruck `__func__` nicht kennt, kommt es zur Fehlermeldung. Des weitern heißt es im `hdnum`-Code:

```
#ifdef HDNUM_DEVEL_MODE
# define THROWSPEC(E) #E << " [" << __func__ << ":" << __FILE__ << ":"
<< __LINE__ << "]: "
#else
# define THROWSPEC(E) #E << ": "
#endif
```

Um den Fehler zu umgehen, habe ich mich beim folgenden C++ Code dafür entschieden, die Variable `HDNUM_DEVEL_MODE` nicht definieren zu lassen, was man am (wieder mal) veränderten Kopf der Quellcode Datei erkennen kann.

C++-Code (QR-Algorithmus)

```
#define TIMER_USE_STD_CLOCK
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
#include <math.h>
#include <iomanip>
//#include "hdnum.hh"
#include "src/exceptions.hh"
#include "src/vector.hh"
#include "src/densematrix.hh"
#include "src/timer.hh"
#include <fstream>

using namespace hdnum;

void drehung(DenseMatrix<double> &A, DenseMatrix<double> &U,int p, int q,int n)
// Matrix wird um phi gedreht, A(q,p) = 0
// U speichert die Rotationsmatrizen
{
//size_t N=A.colsize();
double w;
if (A(p,p)<0)
{
w=-sqrt(A(q,p)*A(q,p)+A(p,p)*A(p,p));
}
else w=sqrt(A(q,p)*A(q,p)+A(p,p)*A(p,p));
double c=A(p,p)/w; //c=cos(phi)
double s=-A(q,p)/w; //s=sin(phi)
double r=s/(1+c);

for (int j=p; j<n; j++)
{
double h=A[p][j];
A[p][j]=A[p][j]-s*(A[q][j]+r*A[p][j]);
A[q][j]=A[q][j]+s*(h-r*A[q][j]);
}
A(q,p)=0;

// matrix A jetzt gedreht, so dass A(q,p)=0

for (int j=0; j<n; j++) // U wird aufgebaut
{
double h=U[j][p];
```

```
U[j][p]=U[j][p]-s*(U[j][q]+r*U[j][p]);
U[j][q]=U[j][q]+s*(h-r*U[j][q]);
}



}

void zerlegung(DenseMatrix<double> &A,DenseMatrix<double> &U,int n)
{
hdnum::identity(U);
//size_t N=A.colsize();
for (int i=0; i<(n-1); i++)
{
drehung(A,U,i,i+1,n);
}
}

double maxfinden(DenseMatrix<double> &A,int n)
{
//size_t N=A.colsize();
double max=0;
for (int i=0; i<(n-1); i++)
{
if (abs(A(i+1,i))>max)
{
max=abs(A(1+i,i));
}
}
return max;
}


int main()
{
//ausgabe in file
  std::ofstream fout;
  fout.open("Matrixqralg.txt");
// Eingabe der Matrix durch den Benutzer
  int n,k;
  double tol,max;
  std::cout<< "Dimension Matrix: \n"; std::cin>> n;
  std::cout<< "Toleranz: \n"; std::cin>> tol;
  hdnum::DenseMatrix<double> A(n,n);
  hdnum::DenseMatrix<double> U(n,n);
  hdnum::DenseMatrix<double> C(n,n);
  hdnum::identity(U);
```

```
hdnum::identity(A);
A*=2;

for(int i=0; i<n; i++)
{
         for(int j=0; j<n; j++)
         {
         if(i==(j+1) || j==(i+1) )
            {A[i][j]=-1; }
         } //Matrix fertig erstellt
}

A.iwidth(2);
A.scientific(false);
A.width(10);
A.precision(5);
std::cout<< "Eingabe Matrix A: \n" << A << "\n";
fout<< "Eingabe Matrix A: \n" << A<< "\n";

max=abs(A(1,0));
k=0;
//for (int l=1; l<2; l++)
//{
while (max>tol)
{
zerlegung(A,U,n);
//std::cout<< "R: \n" << A << "\n Q: \n"<< U<<"\n";
C.mm(A,U);
A=C;
k=k+1;
max=maxfinden(A,n);
//}
}

A.iwidth(2);
U.iwidth(2);
A.scientific(false);
U.scientific(false);
A.width(10);
U.width(10);
A.precision(5);
U.precision(5);
std::cout << "A: \n" << A << "\n";
std::cout << "Iterationsschritte: \n"<<k<<"\n";
//std::cout<<" Q: \n" << U <<"\n";
//std::cout<< "maximum: \n "<< max;
 fout <<"A: \n" << A << "\n";
```

```
   fout << "Iterationsschritte: \n" <<k<<"\n";
   //fout <<" Q: \n"<< U<< "\n";

  fout.close();
    return 0;
}
```

Beispielausgabe für Dimension $n = 6$, Toleranz 0.001:

```
Eingabe Matrix A:

             0          1          2          3          4          5
  0    2.00000   -1.00000    0.00000    0.00000    0.00000    0.00000
  1   -1.00000    2.00000   -1.00000    0.00000    0.00000    0.00000
  2    0.00000   -1.00000    2.00000   -1.00000    0.00000    0.00000
  3    0.00000    0.00000   -1.00000    2.00000   -1.00000    0.00000
  4    0.00000    0.00000    0.00000   -1.00000    2.00000   -1.00000
  5    0.00000    0.00000    0.00000    0.00000   -1.00000    2.00000

A:

             0          1          2          3          4          5
  0    3.80194   -0.00097    0.00000    0.00000    0.00000    0.00000
  1   -0.00097    3.24698   -0.00001   -0.00000   -0.00000    0.00000
  2    0.00000   -0.00001    2.44504   -0.00000    0.00000   -0.00000
  3    0.00000    0.00000   -0.00000    1.55496   -0.00000   -0.00000
  4    0.00000    0.00000    0.00000   -0.00000    0.75302    0.00000
  5    0.00000    0.00000    0.00000    0.00000   -0.00000    0.19806

Iterationsschritte:
44
```

Wie bereits im erwähnt, weist ein Subdiagonalelement $h^{(k)}_{i+1,i}$ folgendes lineares Konvergenzverhalten auf:

$$|h^{(k)}_{i+1,i}| \approx \left|\frac{\lambda_{i+1}}{\lambda_i}\right|^k, \qquad i = 1, 2, \dots n-1.$$

wobei $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$ gelte.
Allerdings kann der Term $|\frac{\lambda_{i+1}}{\lambda_i}|$ beliebig nahe bei eins sein, sodass die Konvergenz sehr langsam ist. Treten zusätzlich auch komplexe Eigenwerte aus $\lambda_i, \lambda_{i+1} = \bar{\lambda}_i$ so ist $|\frac{\lambda_{i+1}}{\lambda_i}| = 1$ und das Verfahren konvergiert überhaupt nicht.
(Beachte: Ist $\lambda \in \mathbb{C}$ Eigenwert einer Matrix A und gilt $\mathrm{Im}(\lambda) \neq 0$, so ist $\bar{\lambda}$ auch Eigenwert von A.)
Im Folgenden, wird jedoch zunächst nur der Fall besprochen, dass die Matrix H nur reelle Eigenwerte hat. Bei symmetrischen Matrizen, insbesondere unsere Beispiel Matrix A, ist dies stets der Fall!

Sei also $\left|\frac{\lambda_{i+1}}{\lambda_i}\right|$ nahe bei eins. Um die Konvergenz des Verfahrens zu beschleunigen, führt man eine sogenannte **Spektralverschiebung** durch. Sei $\sigma \in \mathbb{R}$. Die Matrix $H - \sigma I$, wobei I die Einheitsmatrix ist, hat nun die um σ verschobenen Eigenwerte $\lambda_i - \sigma$, $i = 1, \dots, n$. Ist σ eine gute Schätzung für λ_i so gilt:

$$\left|\frac{\lambda_{i+1}}{\lambda_i}\right| > \left|\frac{\lambda_{i+1} - \sigma}{\lambda_i - \sigma}\right|$$

und $\tilde{h}^{(k)}_{i,i-1}$ konvergiert wesentlich schneller gegen null.
Diese *Spektralverschiebung* wird nun vor jeder Ausführung eines Schrittes des *QR-Algorithmus* durchgeführt und es ergibt sich eine Folge von ähnlichen Matrizen:

$$H_k - \sigma_k I = Q_k R_k, \qquad H_{k+1} = R_k Q_k + \sigma_k I, \qquad k = 1, 2, \dots$$

welche **QR-Algorithmus mit expliziter Spektralverschiebung** genannt wird.
Da bei der Bildung von H_{k+1} die zuvor durhcgeführte Spektralveschiebung rückgängig gemacht wird, haben die Matrizen H_k und H_{k+1} dieselben Eigenwerte.

Bestimmung von σ_k:

Sei

$$C_k := \begin{pmatrix} h^{(k)}_{n-1,n-1} & h^{(k)}_{n-1,n} \\ h^{(k)}_{n,n-1} & h^{(k)}_{n,n} \end{pmatrix}$$

eine Untermatrix von H_k. Da C_k Dimensions 2 hat, lassen sich die Eigenwerte $\mu^{(k)}_1$ und $\mu^{(k)}_2$ von C_k leicht analytisch berechnen. Sei o.B.d.A. $\mu^{(k)}_1$ der Eigenwert von C_k welcher näher bei $h^{(k)}_{n,n}$ liegt. Dann setzt man σ_k gleich diesem Eigenwert. D. h.:

$$\sigma_k = \mu^{(k)}_1 \quad \text{mit} \quad |\mu^{(k)}_1 - h^{(k)}_{n,n}| \leq |\mu^{(k)}_2 - h^{(k)}_{n,n}|, \ k = 1, 2, \dots$$

(Siehe Hans-Rudolf Schwarz, Norbert Köckler, *Numerische Mathematik*, 5. Auflage, Seite 250)

Bei dieser Wahl von σ_k ist es am wahrscheinlichsten, dass das Subdiagonalelement $h^{(k)}_{n,n-1}$ zuerst null wird oder zumindest als erstes die geforderte Toleranzgrenze erreicht. Die Matrix H_k zerfällt dann zu:

$$H_k = \begin{pmatrix} \times & \times & \times & \times & \times & \vdots & \times \\ \times & \times & \times & \times & \times & \vdots & \times \\ 0 & \times & \times & \times & \times & \vdots & \times \\ 0 & 0 & \times & \times & \times & \vdots & \times \\ 0 & 0 & 0 & \times & \times & \vdots & \times \\ 0 & 0 & 0 & 0 & 0 & \vdots & \lambda \end{pmatrix} = \begin{pmatrix} \hat{H} & \vdots & \hat{h} \\ 0^T & \vdots & \lambda \end{pmatrix} \qquad \text{(o.B.d.A.: } n = 6\text{)}$$

Tritt diese Situation ein, so hat man bereits den ersten Eigenwert λ gefunden und kann den $QR - Algorithmus$ nun auf die Untermatrix $\hat{H}$ anwenden. Hier zeigt sich ein Vorteil des $QR - Algorithmus$; bei jedem gefunden Eigenwert, reduziert sich die weiter zu bearbeitene Matrix um eine Dimension!

Bei der Implementierung des *QR-Algorithmus mit Spektralverschiebung* wollen wir nun zusätzlich auf den einfacher zu programmierenden Code, welcher Matrizenmultiplikation verwendet, verzichten und zeit- und speicherplatzsparsamer programmieren. Dazu betrachten wir zunächst die Gleichungen:

$$\begin{aligned} U^T_{n-1} \ \ldots U^T_3 U^T_2 U^T_1 (H_k - \sigma_k I) &= Q^T_k (H_k - \sigma_k I) = R_k \\ H_{k+1} &= R_k Q_k + \sigma_k I = R_k U_1 U_2 U_3 \ldots U_{n-1} + \sigma_k I \\ &= U^T_{n-1} \ldots U^T_3 U^T_2 U^T_1 (H_k - \sigma_k I) U_1 U_2 U_3 \ldots U_{n-1} + \sigma_k I \end{aligned}$$

wobei $U_i,\ i = 1, 2, \ldots, n-1$ die bereits aus den vorherigen Verfahren bekannten Drehmatrizen sind. Da bei der Bildung von R aus H nur die Subdiagonalelemente auf null gedreht werden müssen, gibt es $n-1$ Drehmatrizen U pro $QR - Zerlegung$ (Iterationsschritt).
Nun kann man sich die Assoziativität der Matrizenmultiplikation zunutze machen und bei der Bildung von H_{k+1} eine geeignete Reihenfolge wählen. Startmatrix ist $H - k - \sigma_k I$. Nach Multiplikation der ersten beiden Drehmatrizen, hat $U^T_2 U^T_1 (H_k - \sigma_k I)$ folgende allgemeine Gestalt:

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{pmatrix} \qquad \text{(sei wieder o.B.d.A.: } n = 6\text{)}$$

Die weiteren Multiplikationen von links, lassen die ersten beiden Spalten unverändert! Die Multiplikation von recht mit U_1, welche ja nur eine Linearkombination der ersten und zweiten Spalte bewirkt, operiert also bereits mit den endgültigen Werten von R_k der $QR - Zerlegung$. Aus analogem Grund kann dann nach der Multiplikation mit U^T_3 von links, die Multiplikation mit U_2 von rechts ausgeführt werden.
Durch diese gestaffelte Ausführung des Algorithmus, müssen die (n-1)-Wertepaare $c_i = \cos\varphi_i$ und $s_i = \sin\varphi_i$ nicht alle abgespeichert werden.
Im i-ten Schritt ($2 \leq i \leq n-1$) der $QR - Transformation$ wird also in der i-ten und

$(i+1)$-ten Zeile die QR-Zerlegung weitergeführt, während in der $(i-1)$-ten und i-ten Spalte im Wesentlichen H_{k+1} aufgebaut wird. Subtraktion und die spätere Addition von σ_k zu den Diagonalelementen, kann fortlaufend in den Prozess eingebunden werden.
Der Vorteil dieser *Spektralverschiebung* zeigt sich deutlich an den benötigten Iterationsschritten.
(Beachte Iterationsschritte bei den beiden Beispielausgaben der jeweiligen C++ -Codes. Im ersteren, ohne Spektralverschiebung sind 44 Iterationsschritte nötig, mit Spektralverschiebung nur 9. Wählt man z.B. $n = 10$ und Toleranz 0.0001, so wird der Unterschied noch deutlicher. Ohne Spektralverschiebung 136 Iterationsschritte, mit nur noch 16.)

Rechenaufwand

Der Rechenaufwand des $QR-Verfahrens$ setzt sich aus 2 Teilen zusammen:
1. Bildung der Rechtsdreiecksmatrix. Im j-ten Schritt werden $4+4(n-j)$ Multiplikationen und eine Quadratwurzel benötigt (siehe dazu Kapitel **QR-Zerlegung**, Formeln (1) bis (3)). Hinzu kommt die Berechnung von σ mit 8 Multiplikationen und 2 Quadratwurzeln.
2. Bildung der neuen Matrix H': Im j-ten Schritt werden $4j+2$ multiplikative Operationen benötigt (Unterhalb der Diagonalen stehen nur Nullen.)
Summiert man nun über j von 1 bis $(n-1)$ so erhält man einen Rechenaufwand von $4n^2+10n-14$ Multiplikationen und $3(n-1)$ Quadratwurzeln.
Allerdings muss man beachten, dass im Laufe des $QR-Verfahrens$, sobald ein Eigenwert gefunden wurde, der Algorithmus nur noch auf die $(n-1)$ dimensionale Untermatrix angewandt wird. Nachdem der zweite Eigenwert gefunden wurde nur noch auf die $(n-2)$ dimensionale Untermatrix, usw. Hieran erkennt man einen Vorteil dieses Verfahrens.

C++ -Code des QR-Algorithmus mit expliziter Spektralverschiebung:

```
#define TIMER_USE_STD_CLOCK
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
#include <math.h>
#include <iomanip>
#include "hdnum.hh"
#include <fstream>

using namespace hdnum;




void qralg(DenseMatrix<double> &A,double tol,int n,double shift,int z)
{
//shift=0;
A(0,0)=A(0,0)-shift;
```

```
double ce;
double se;
double re;
double c;
double s;
for (int i=0; i<n; i++)
{
if (i<(n-1)) // für i=n-1 nur noch multiplikation von rechts
{
double w;
if (abs(A(i,i))<(1.11022e-16*abs(A(i+1,i)))) //1.11022e-16 ist Maschinengenauigkeit
{
w=abs(A(i+1,i));
c=0;
s=(A(i+1,i))/(abs(A(i+1,i)));
}

else if (A(i,i)<0)
{
w=-sqrt(A(i+1,i)*A(i+1,i)+A(i,i)*A(i,i));
c=A(i,i)/w;
s=-A(i+1,i)/w;
}

else
{
w=sqrt(A(i,i)*A(i,i)+A(i+1,i)*A(i+1,i));
c=A(i,i)/w; //c=cos(phi)
s=-A(i+1,i)/w; //s=sin(phi)
}

double r=s/(1+c);
A(i,i)=w;
A(i+1,i)=0;
A(i+1,i+1)=A(i+1,i+1)-shift;

for (int j=(i+1); j<n; j++)
{
double h=A[i][j];
A[i][j]=A[i][j]-s*(A[i+1][j]+r*A[i][j]);
A[i+1][j]=A[i+1][j]+s*(h-r*A[i+1][j]);
}

}
if (i>0)
{
for (int j=0; j<(i+1); j++)
```

```
{
double h=A[j][i-1];
A[j][i-1]=A[j][i-1]-se*(A[j][i]+re*A[j][i-1]);
A[j][i]=A[j][i]+se*(h-re*A[j][i]);
}
A(i-1,i-1)=A(i-1,i-1)+shift;

}
ce=c;
se=s;
re=se/(1+ce);
}
A(n-1,n-1)=A(n-1,n-1)+shift;
}



int main()
{
//ausgabe in file
  std::ofstream fout;
  fout.open("MatrizenQRalg.txt");

  int n;
  int k=0;
  int z=0;
  double tol;
  std::cout<< "Dimension Matrix: \n"; std::cin>> n;
  int dim=n;  // n wird im alg geändert, deshalb einführung dim
  std::cout<< "Toleranz: \n"; std::cin>> tol;
  hdnum::DenseMatrix<double> A(n,n);
  hdnum::identity(A);
  A *=2; // Matrix hat jetzt 2er auf der diagonalen

  for(int i=0; i<(n-1); i++)
  {
       A(i,i+1)=A(i+1,i)=-1; //Matrix fertig erstellt
  }

  A.iwidth(2);
  A.scientific(false);
  A.width(10);
  A.precision(5);

  std::cout<< "eingegebene Matrix: \n" << A<< "\n";
  fout<< "eingegebene Matrix: \n" << A<< "\n";
```

```
  double *eigenwerte = new double [n];

  while (n>1)
  {
  if (abs(A(n-1,n-2))>tol) //check ob A in unter matrix zerfallen ist
  {
  double shift; // Bestimmung des shifts nach
  double a=A(n-2,n-2); // Seite 250, numerische mathematik,
  double b=A(n-2,n-1); // Schwarz und Köckler
  double c=A(n-1,n-2);
  double d=A(n-1,n-1);
  double eig1=0.5*(a+d+sqrt((a+d)*(a+d)-4*(d*a-b*c)));
  double eig2=0.5*(a+d-sqrt((a+d)*(a+d)-4*(d*a-b*c)));
  if (abs(eig1-A(n-1,n-1))<abs(eig2-A(n-1,n-1)))
  {
  shift=eig1;
  }
  else
  {
  shift=eig2;
  }
  //double shift=A(n-1,n-1); //Spektralverschiebung shift
  qralg(A,tol,n,shift,z);
  k=k+1;
 }
  else
  {
  //if (n==6)
  //{ std::cout<< "hallo \n";
  //}
   eigenwerte[n-1]=A(n-1,n-1); //letztes element konvergiert gegen EW nach Satz
n=n-1; // algorithmus wird auf unter matrix angewandt
  }
  }

  std::cout << "neue Matrix A: " << A<< "\n";
  eigenwerte[0]=A(0,0);

  //A.iwidth(2);
  //A.scientific(false);
  //A.width(10);
  //A.precision(5);
  std::cout << "Eigenwerte von A: \n";
  for (int i=0; i<dim; i++)
  {
std::cout<<eigenwerte[i] <<"\n";
  }
```

```
  std::cout<< "Anzahl Iterationsschritte: \n"<< k<<"\n";
   fout <<"Eigenwerte von A: \n";
   for (int i=0; i<dim; i++)
   {
   fout <<eigenwerte[i]<< "\n";
   }
   fout<<"Anzahl Iterationsschritte: \n"<<k<<"\n";

  fout.close();
    return 0;
}
```

Beispielausgabe, n=6, Toleranz =0.001:

```
eingegebene Matrix:

             0          1          2          3          4          5
  0    2.00000   -1.00000    0.00000    0.00000    0.00000    0.00000
  1   -1.00000    2.00000   -1.00000    0.00000    0.00000    0.00000
  2    0.00000   -1.00000    2.00000   -1.00000    0.00000    0.00000
  3    0.00000    0.00000   -1.00000    2.00000   -1.00000    0.00000
  4    0.00000    0.00000    0.00000   -1.00000    2.00000   -1.00000
  5    0.00000    0.00000    0.00000    0.00000   -1.00000    2.00000

Eigenwerte von A:
3.80194
3.24698
2.44504
0.19806
1.55496
0.75302
Anzahl Iterationsschritte:
9
```

Transformation auf Hessenberg-Form **Zeitraum: 12.01. – 19.01.2012**

In den beiden vorherigen Kapiteln über den **QR-Algorithmus** wird erwähnt, dass dieser Algorithmus zur Berechnung der Eigenwerte nur angewandt werden kann, wenn sich die Matrix in $Hessenberg - Form$ befindet. In diesem Kapitel wird ein Verfahren zur Transformation einer beliebigen, quadratischen Matrix A in $Hessenberg - Form$ vorgestellt.

Wiederholung: $Hessenberg - Form$ bedeutet:

$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} & \dots & h_{1n} \\ h_{21} & h_{22} & h_{23} & h_{24} & \dots & h_{2n} \\ 0 & h_{32} & h_{33} & h_{34} & \dots & h_{3n} \\ 0 & 0 & h_{43} & h_{44} & \dots & h_{4n} \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & h_{n,n-1} & h_{nn} \end{pmatrix}$$

also $h_{ij} = 0 \quad$ für alle $i > j + 1$.

Um eine beliebige quadratische Matrix auf diese Form zu bringen, wird wieder mit $Jacobi-Rotationsmatrizen$ gearbeitet. Anders als beim $Jacobi - Verfahren$ soll diesmal ein bereits auf Null gedrehtes Element seinen Wert beibehalten. Aus diesem Grund werden die Rotationsindexpaare nach einer anderen Strategie gewählt. Man spricht deshalb auch statt von Jacobi-Rotationen von $Givens - Rotationen$. Eine einfache und naheliegende Strategie zur Durchführung dieser $Givens-Rotationen$, ist die spaltenweise Eliminierung der Matrixelemente unterhalb der Nebendiagonalen. Dies geschieht in der Reihenfolge:

$$a_{31}, a_{41}, \dots, a_{n1}, a_{42}, a_{52}, \dots, a_{n2}, \dots, a_{n,n-2}$$

Die dazugehörigen Rotationsindexpaare sind:

$$(2,3), (2,4), \dots, (2,n), (3,4), (3,5), \dots, (3,n), \dots, (n-1,n)$$

Allgemein gesagt: Zur Eliminierung des Elements $a_{ij} \neq 0$ mit $i \geq j+2$ wird eine $(j+1, i)$-Drehung angewandt. Durch Wahl $(j+1, i)$ als Rotationsindexpaar, ist a_{ij} nur durch die Zeilenoperation betroffen (siehe Kapitel **Einarbeitung in die Theorie**). Die Matrixeinträge ändern sich für $A' = U^T A$ folgendermaßen ($U(p, q, \varphi)$):

$$\left.\begin{aligned} a'_{pk} &= a_{pk} \cos\varphi - a_{qk} \sin\varphi \\ a'_{qk} &= a_{pk} \sin\varphi + a_{qk} \cos\varphi \\ a'_{lk} &= a_{lk} \text{ für } l \neq p, q \end{aligned}\right\} \; k = 1, \dots, n \qquad (A)$$

und für $A'' = A'U$:

$$\left.\begin{aligned} a''_{lp} &= a'_{lp} \cos\varphi - a'_{lq} \sin\varphi \\ a''_{lq} &= a'_{lp} \sin\varphi + a'_{lq} \cos\varphi \\ a''_{lk} &= a'_{lk} \text{ für } k \neq p, q \end{aligned}\right\} \; l = 1, \dots, n \qquad (B)$$

Aufgrund der obigen Reihenfolge der Rotationsindexpaare, wird ein einmal auf Null gedrehtes Element nicht mehr verändert, da in den nachfolgenden Schritten in diesen Spalten, durch die Zeilenoperation, nur Nullelemente linear kombiniert werden. Insgesamt werden somit höchstens $N = (n-1(n-2)/2$ Givens-Rotationen durchgeführt.

Anders als beim $Jacobi - Verfahren$ soll diesmal kein Außendiagonalelement auf null gedreht werden, sondern a_{ij}, woraus sich die Bedingung:

$$a'_{ij} = a_{j+1,j} \sin\varphi + a_{ij} \cos\varphi = 0$$

ergibt, wobei $p = j + 1 < i = q$ gilt.
Zusammen mit der Bedingung

$$\cos^2\varphi + \sin^2\varphi = 1$$

lassen sich die Werte $s = \sin\varphi$ und $c = \cos\varphi$ dann wie folgt bestimmen:

$$\begin{aligned}
&\text{Falls} \quad a_{j+1,j} \neq 0: \quad w := \operatorname{sgn}(a_{j+1,j})\sqrt{a_{j+1,j}^2 + a_{ij}^2}, && (C)\\
&\qquad \cos\varphi = \frac{a_{j+1,j}}{w}, \quad \sin\varphi = \frac{-a_{ij}}{w} && (D)\\
&\text{Falls} \quad a_{j+1,j} = 0: \quad w := -a_{ij},\\
&\qquad \cos\varphi = 0, \qquad \sin\varphi = 1,
\end{aligned}$$

Es gilt nun $a'_{ij} = 0$. Wie man an den Gleichungen (A) sieht gilt $a'_{j+1,j} = w$. Durch die Einführung von w lässt sich also eine Rechenoperation sparen.
Anhand der Gleichungen (B) lässt sich auch nochmal sehen, dass a'_{ij} nicht noch einmal verändert wird und den Wert Null beibehält.

Rechenaufwand

Der Rechenaufwand zur Elimination des Elements a_{ij} ergibt sich aus $4 + 4n + 4(n - j)$ Multiplikationen und einer Quadratwurzel. Man beachte die Formeln (A), (B), (C) und (D), wobei bei (A) der Laufindex bereits bei $j = q = p + 1$ beginnen kann, da davor nur Nullen miteinander kombiniert werden (siehe hierzu den C++Code).
Da die j-te Spalte $(n-j-1)$ solcherElemente besitzt und es $(n-2)$ zu bearbeitene Spalten gibt, ergibt sich ein gesamt Rechenaufwand von $\frac{10}{3}n^3 - 8n^2 + \frac{2}{3}n + 4$ Multiplikationen und $(n-1)(n-2)/2$ Quadratwurzeln.

C++ Code der Hessenbergtransformation:

```
#define TIMER_USE_STD_CLOCK
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
#include <math.h>
#include <iomanip>
//#include "hdnum.hh"
```

```
#include "src/exceptions.hh"
#include "src/vector.hh"
#include "src/densematrix.hh"
#include "src/timer.hh"
#include <fstream>

using namespace hdnum;


void drehung(DenseMatrix<double> &A,DenseMatrix<double> &U, int j, int i,int n)
{ //A(i,j) wird auf null gedreht

double c;
double s;
double w;
if (abs(A(j+1,j))<(1.11022e-16*abs(A(i,j))))
//if (A(j+1,j)==0)
{
w=-A(i,j);
c=0;
s=1;
}
else
{
w=(A(j+1,j))/(abs(A(j+1,j)))*sqrt(A(j+1,j)*A(j+1,j)+A(i,j)*A(i,j));
c=(A(j+1,j))/w;
s=(-A(i,j))/w;

}
A(j+1,j)=w;
A(i,j)=0;

for (int k=j+1; k<n; k++)
{
double h=c*A(j+1,k)-s*A(i,k);   //rest der zeile (k<(j+1))
A(i,k)=s*A(j+1,k)+c*A(i,k);     // uninteressant, da nur nullen
A(j+1,k)=h; // miteinander kombiniert werden.
}
for (int k=0; k<n; k++)
{
double h=c*A(k,j+1)-s*A(k,i);
A(k,i)=s*A(k,j+1)+c*A(k,i);
A(k,j+1)=h;
}

}
```

```
int main()
{
//ausgabe in file
  std::ofstream fout;
  fout.open("hesse.txt");
// Eingabe der Matrix durch den Benutzer
  int n;
  std::cout<< "Dimension Matrix: \n"; std::cin>> n;
  hdnum::DenseMatrix<double> A(n,n);
  hdnum::identity(A);




  for (int j=0; j<n; j++) //  beliebige matrix durch benutzer
  { //   definiert
  for (int i=0; i<n; i++)
  {
  std::cout<< "A("<<j <<","<<i <<" )= ";
  std::cin>> A(j,i); std::cout<< "\n";
  }
  }

  A.iwidth(2);
  A.scientific(false);
  A.width(10);
  A.precision(5);

  std::cout<< "eingegebene Matrix: \n" << A<< "\n";
  fout<< "eingegebene Matrix: \n" << A<< "\n";


  for (int j=0; j<(n-2); j++) //für hesseberg muss in den letzten beiden spalten
  {   // keine null auftreten
  for (int i=j+2; i<n; i++)
  {
  if (A(i,j) != 0)
  {
  drehung(A,U,j,i,n);
  }
  }
  }

  A.iwidth(2);
  A.scientific(false);
  A.width(10);
  A.precision(5);
```

```
  std::cout<< "Hessenbergform: \n"<<A<< "\n";

  fout<< "Hessenbergform: \n"<<A<< "\n";


  fout.close();
   return 0;
}
```

Beispielausgabe Dimension $n = 6$, Matrix beliebig, nicht symmetrisch:

```
eingegebene Matrix:

             0          1          2          3          4          5
  0    1.00000    3.00000    5.00000    7.00000    9.00000   11.00000
  1   -2.00000    4.00000   -6.00000    8.00000  -10.00000   12.00000
  2    0.00000    1.00000    0.00000    1.00000    0.00000    1.00000
  3   12.00000  -12.00000    3.00000   -3.00000    6.00000   -6.00000
  4    8.00000   -8.00000   -1.00000    1.00000    0.00000   10.00000
  5   -4.00000    0.00000    7.00000   -2.00000   -2.00000    0.00000

Hessenbergform:

             0          1          2          3          4          5
  0    1.00000   -7.02002  -10.17944   -6.40428    4.80321    8.24698
  1  -15.09967    3.78947   -2.95678  -10.13979  -11.37826   -0.73963
  2    0.00000   -4.03162   -0.59309   14.60563   -8.00438   -2.20323
  3    0.00000    0.00000   -3.69612   -3.17186    1.80876   -2.90016
  4    0.00000    0.00000    0.00000   13.68378   -1.17706    6.04499
  5    0.00000    0.00000    0.00000    0.00000    0.06901    2.15253
```

Vergleich: QR-Algorithmus – Jacobiverfahren **Zeitraum: 19.01. – 26.01.2012**

Im abschließenden Kapitel werden die zwei vorgestellten numerischen Algorithmen zur Bestimmung der Eigenwerte einer Matrix bezüglich des Rechenaufwandes mit einander verglichen.
Dies geschieht mit der bereits eingeführten symmetrischen Matrix A

$$A = \begin{array}{c} \begin{matrix} 1 & 2 & \cdots & m \end{matrix} \\ \begin{pmatrix} B_n & -I_n & & \\ -I_n & B_n & \ddots & \\ & \ddots & \ddots & -I_n \\ & & -I_n & B_n \end{pmatrix} \end{array} \in \mathbb{R}^{N\times N}, \qquad B_n = \begin{pmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{n\times n}$$

Wenn $m = n$ und damit insbesondere $N = n^2$ gilt, lassen sich die Eigenwerte dieser Matrix mit der Formel

$$4 - 2[\cos(\gamma h\pi) + \cos(\mu h\pi)], \quad h := \frac{1}{n+1}, \quad 1 \le \gamma, \mu \le n$$

analytisch bestimmen. (siehe hierzu *Einführung in die Numerische Mathematik*, Vorlesungsskiptum SS 2005, `Rolf Rannacher`)
Aus diesem Grund werden weiter unter nur Quadratzahlen als Dimensionen der Matrizen verwendet. Im folgenden Vergleich der zwei Algorithmen, sei n die Dimension der Matrix. Es wird mit Genauigkeit 10^{-7} gerechnet und bei der Angabe der Toleranz handelt es sich um eine relative Toleranz. Es wird also gefordert:

$$\max_{i>j}|a_{ij}| < \max_i|a_{ii}| \cdot TOL \qquad 1 \le i, j \le n$$

Wiederholung: Rechenaufwand von Jacobiverfahren bzw. QR-Algorithmus

Beide Verfahren verwenden als Grundbausteine die bereits bekannten Rotationsmatrizen $U(p, q, \varphi)$. Der QR-Algorithmus verwendet pro Iterationsschritt $(n-1)$ Rotationsmatrizen, das Jacobiverfahren nur eine Rotation pro Iterationsschritt, dafür müssen pro Schritt $(n^2-n)/2-1$ Vergleichsoperationen verwendet werden. Außerdem verringert sich die Dimension beim $QR-Algorithmus$ um 1 je gefundenen Eigenwert.
Pro Iteration werden beim $QR-Algorithmus$ $4n^2+10n-14$ Multiplikationen und $3(n-1)$ Quadratwurzeln verwendet. Hinzu kommen einmalig $\frac{10}{3}n3 - 8n^2 + \frac{2}{3}n + 4$ Multiplikationen und $(n-1)(n-2)/2$ Quadratwurzeln aus der $Hessenbergtransformation$, da obige Matrix erst in Hessenbergform transformiert werden muss, bevor der $QR-Algorithmus$ anwendbar ist.
Beim $Jacobiverfahren$ werden pro Iteration $4n+5$ Multiplikationen und 2 Quadratwurzeln verwendet.
DIe Maximale Abweichung gibt den größten Unterschied von korrektem Eigenwert und numerisch bestimmten Eigenwert an. Da hier mit einer sehr dünn besetzten Matrix gearbeitet wird, reduziert sich bei der Transformation auf Hessenberggestalt die Anzahl der Multiplikationen auf $6n^2 - 6n - 12$ und die Anzahl der Quadratwurzeln auf $n-2$.

QR-Algorithmus (+ Hessenbergtransformation)

	Multiplikationen	Quadratwurzeln	Maximale Abweichung
n=4			
TOL=0.01	110 (+60)	11 (+2)	$8.89 \cdot 10^{-4}$
=0.001	154 (+60)	15 (+2)	0
=0.0001	154 (+60)	15 (+2)	0
=0.00001	154 (+60)	15 (+2)	0
n=9			
TOL=0.01	1100 (+420)	42 (+7)	$8.86 \cdot 10^{-5}$
=0.001	1144 (+420)	46 (+7)	0
=0.0001	1144 (+420)	46 (+7)	0
=0.00001	1144 (+420)	46 (+7)	0
n=16			
TOL=0.01	4006 (+1428)	109 (+14)	$2.4018 \cdot 10^{-3}$
=0.001	4756 (+1428)	118 (+14)	$4.9 \cdot 10^{-6}$
=0.0001	5174 (+1428)	133 (+14)	0
=0.00001	6608 (+1428)	166 (+14)	0
n=16			
TOL=0.01	16366 (+3588)	249 (+23)	$5.9582 \cdot 10^{-3}$
=0.001	19370 (+3588)	305 (+23)	$2.69 \cdot 10^{-5}$
=0.0001	24400 (+3588)	380 (+23)	10^{-7}
=0.00001	26774 (+3588)	433 (+23)	0
n=100			
TOL=0.01	ca $1.2 \cdot 10^6$	4732	$2.7716 \cdot 10^{-2}$
=0.001	ca $1.5 \cdot 10^6$	6070	$9.196 \cdot 10^{-4}$
=0.0001	ca $1.7 \cdot 10^6$	6701	$4.4 \cdot 10^{-6}$
=0.00001	ca $1.8 \cdot 10^6$	7096	0

Jacobi-Verfahren

	Multiplikationen	Quadratwurzeln	Maximale Abweichung
n=4			
TOL=0.01	80	8	0
=0.001	80	8	0
=0.0001	80	8	0
=0.00001	80	8	0
n=9			
TOL=0.01	1440	72	$1.0760 \cdot 10^{-3}$
=0.001	1960	98	$1.49 \cdot 10^{-5}$
=0.0001	2360	118	$2 \cdot 10^{-7}$
=0.00001	2600	130	0
n=16			
TOL=0.01	4216	124	$3.0218 \cdot 10^{-3}$
=0.001	5984	176	$2.72 \cdot 10^{-5}$
=0.0001	7344	216	$2 \cdot 10^{-7}$
=0.00001	7888	232	0
n=25			
TOL=0.01	17056	328	$1.14667 \cdot 10^{-2}$
=0.001	36088	694	$2.609 \cdot 10^{-4}$
=0.0001	47944	922	$5 \cdot 10^{-7}$
=0.00001	58032	1116	0
n=100			
TOL=0.01	487628	2414	$6.56928 \cdot 10^{-2}$
=0.001	ca $2.0 \cdot 10^6$	9886	$9.828 \cdot 10^{-4}$
=0.0001	ca $3.1 \cdot 10^6$	15388	$2.48 \cdot 10^{-5}$
=0.00001	ca $3.9 \cdot 10^6$	19264	0

Beide Verfahren haben bei TOL=0.01 eine große Abweichung, so dass die Ergebnisse erst ab TOL=0.0001 guten Gewissens verwendet werden können. Je höher die Dimension der verwendeten Matrix desto deutlicher zeigen sich die Vorzüge des $QR-Algorithmus$. Trotz hoher zusätzlicher Operationen aufgrund der Hessenbergtransformation, ist die Anzahl wesentlich geringer als bem $Jacobiverfahren$. Ab $n = 25$ sogar weniger als die Hälfte. Hinzu kommt, dass der $QR - Algorithmus$ (fast) immer eine kleinere Abweichung vom korrekten Wert aufweist.
Was man leider anhand dieser Tabelle nicht sieht, ist die Rechenzeit, welche beim $QR - Algorithmus$ ebenfalls deutlich geringer ist als beim $Jacobi-Verfahren$. Für $n = 25$ und TOL=0.00001 müssen bereits 166 842 Vergleichsoperationen durchgeführt werden. Ein erheblicher Zeitunterschied bei der Rechenzeit entsteht schließlich bei $n = 100$. Zwischen 6 bis 47.5 Millionen (!!) Vergleichsoperationen werden durchgeführt.
(Bei mir benötigte der $QR - Algorithmus$ ($n = 100$, TOL=0.00001) eine Rechenzeit von ca 11 Sekunden. Das $Jacobi - Verfahren$ brauchte etwa 204 Sekunden!)