Kapitel 5

Lineare Gleichungssysteme und der Gauß-Algorithmus

Verständnisfragen

Sachfragen

1. Was ist ein lineares Gleichungssystem?
2. Was ist die erweiterte Koeffizientenmatrix eines linearen Gleichungssystems?
3. Was ist eine obere bzw. untere Dreiecksmatrix?
4. Was ist ein gestaffeltes lineares Gleichungssystem?
5. Wie verläuft die Rückwärts– bzw. Vorwärtssubstitution?
6. Den Verlauf der Gauß-Elimination erläutern können.
7. Die Matrix-Darstellung der Gauß-Elimination herleiten können.
8. Bedingungen für die Invertierbarkeit einer quadratischen Matrix kennen.
9. Die LU-Zerlegung und ihre Anwendung bei der Lösung von linearen Gleichungssystemen erläutern können.
10. Was versteht man unter Scheinskalierung?
11. Was versteht man unter der Pivotwahl? Welche Strategien kennen Sie?
12. Was ist eine Determinante?
13. Wie wird eine 2×2 Determinante berechnet?
14. Welche Eigenschaften haben Determinanten?
15. Wie wird eine 3×3 Determinanten berechnet?
16. Erläutern Sie den Laplace'schen Entwicklungssatz zur Berechnung allgemeiner Determinanten!
17. Was ist ein Eigenwert einer quadratischen Matrix?

18. Was ist ein Eigenvektor einer quadratischen Matrix?
19. Wie werden Eigenwerte und Eigenvektoren einer quadratischen Matrix berechnet?
20. Was versteht man unter der Ähnlichkeit von Matrizen?

Methodenfragen

1. Die erweiterte Koeffizientenmatrix eines linearen Gleichungssystems aufstellen können.
2. Ein gestaffeltes lineares Gleichungssystem erkennen und mit Rückwärts- oder Vorwärtssubstitution lösen können.
3. Elementare Zeilenumformungen durchführen können.
4. Gauß-Elimination durchführen können.
5. Elementarmatrizen zu elementaren Zeilenumformungen aufstellen können.
6. Feststellen können, ob eine gegebenen quadratische Matrix invertierbar ist.
7. Die Inverse einer invertierbaren quadratischen Matrix mit Gauß-Jordan bestimmen können.
8. Die Inverse einer invertierbaren quadratischen Matrix mit linearen Gleichungssystemen bestimmen können.
9. Mit einer gegebenen LU-Zerlegung der Koeffizientenmatrix ein lineares Gleichungssystem lösen können.
10. Die LU-Zerlegung mit und ohne Zeilenvertauschungen durchführen können.
11. Die Gauß-Elimination und die LU-Zerlegung mit Scheinskalierung und Spaltenpivotsuche implementieren können.
12. 2×2, 3×3 und beliebige $n \times n$-Determinanten berechnen können.
13. Die Eigenschaften der Determinante anwenden können.
14. Der Laplace'schen Entwicklungssatz anwenden können.
15. Eigenwerte und Eigenvektoren einer quadratischen Matrix berechnen können.
16. Entscheiden können, ob eine Matrix zu einer Diagonalmatrix ähnlich ist.

Übungsaufgaben

1. Bestimmen Sie in der erweiterten Koeffizientenmatrix $\begin{pmatrix} a & b & c & -2 \\ c & a & b & 8 \\ b & c & a & 0 \end{pmatrix}$ die Koeffizienten a, b und c so, dass das zugehörige Gleichungssystem genau die Lösung $x_1 = 1, x_2 = -1$ und $x_3 = 2$ besitzt!

   Lösung:
   a, b, c ergeben sich als Lösung des Gleichungssystems mit der Koeffizientenmatrix

   $$\begin{pmatrix} 1 & -1 & 2 \\ -1 & 2 & 1 \\ 2 & b & -1 \end{pmatrix}.$$

   Das Gleichungssystem erhalten Sie, indem Sie die gegebene Lösung in die Gleichungen einsetzen und nach a, b und c sortieren. Die Lösung dieses Gleichungssystems ist gegeben durch $a = -1, b = 3, c = 1$.

2. Bestimmen Sie die Koeffizienten des Polynoms $p(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ so, dass $p(-1) = 0, p(1) = 2, p(2) = 3, p(3) = 12$ erfüllt sind!

   Lösung:
   Die Koeffizienten des kubischen Polynoms ergeben sich als Lösung eines linearen Gleichungssystems. Die Gleichungen für das System erhalten Sie, indem sie die gegebenen Punkte in die Polynome einsetzen; die rechte Seite ist durch die vorgegebenen Funktionswerte gegeben. Als Lösung erhält man $a_3 = 1, a_2 = -2, a_1 = 0, a_0 = 3$.

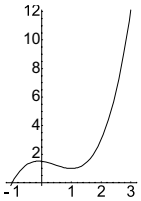

Abbildung 5.1: Das gesuchte Polynom aus Aufgabe 2

3. Bestimmen Sie die Elementarmatrizen E_1, E_2, E_3 und E_4 mit $E_1A = B, E_2B = A, E_3A = C, E_4C = A$ für die Matrizen $A = \begin{pmatrix} 3 & 4 & 1 \\ 2 & -7 & -1 \\ 8 & 1 & 5 \end{pmatrix}$, $B = \begin{pmatrix} 8 & 1 & 5 \\ 2 & -7 & -1 \\ 3 & 4 & 1 \end{pmatrix}$, $C = \begin{pmatrix} 3 & 4 & 1 \\ 2 & -7 & -1 \\ 2 & -7 & 3 \end{pmatrix}$.

   Lösung:
   Die Elementarmatrix E_1 ist die Matrix, die die erste mit der dritten Zeile vertauscht. Auch E_2 ist durch diese Matrix gegeben. E_3 ist die Matrix, die von der dritten Zeile das zweifache der ersten Zeile subtrahiert; E_4 addiert zur dritten Zeile das zweifache der ersten Zeile hinzu.

   Es ist

$$E_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} E_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}, E_4 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}$$

4. Lösen Sie die linearen Gleichungssysteme $A\mathbf{x} = \mathbf{b}$ mit Hilfe der Gauß-Elimination für die erweiterten Koeffizientenmatrizen $\begin{pmatrix} 1 & 0 & 1 & 3 \\ 2 & 2 & 1 & 10 \\ 0 & 2 & 1 & 4 \end{pmatrix}$ und $\begin{pmatrix} 9 & -11 & 10 & 4 & 9 \\ 2 & -2 & 2 & 1 & 3 \\ 7 & -16 & 11 & 2 & -7 \\ 1 & 0 & 0 & 2 & 8 \end{pmatrix}$.

   Lösung:
   Die Lösung für das 3×3 Gleichungssystem ist $\mathbf{x} = (3, 2, 0)^T$; für das 4×4 System erhält man die Lösung $\mathbf{x} = (2, 1, -1, 3)^T$.

5. Bestimmen Sie die Inversen der Matrizen $A = \begin{pmatrix} 1 & 6 & 4 \\ 2 & 4 & -1 \\ -1 & 2 & 5 \end{pmatrix}$ und $B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 1 & 3 & 5 & 0 \\ 1 & 3 & 5 & 7 \end{pmatrix}$!

   Lösung:
   A ist nicht invertierbar, der Gauß-Jordan Algorithmus bricht ab. B ist invertierbar und eine untere Dreiecksmatrix, also muss nur noch das untere Dreieck eliminiert werden. Das Ergebnis des Gauß-Jordan-Algorithmus ist

$$B^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{3} & \frac{1}{3} & 0 & 0 \\ 0 & -\frac{1}{5} & \frac{1}{5} & 0 \\ 0 & 0 & -\frac{1}{7} & \frac{1}{7} \end{pmatrix}$$

6. Schreiben Sie eine Funktion in der Programmiersprache Ihrer Wahl, die die Rückwärtssubstitution für eine obere Dreiecks-Matrix `A` und eine rechte Seite `b` durchführt. Testen Sie diese

Funktion an selbst gewählten linearen Gleichungssystemen!

Lösung:
Hier ein Java-Code mit Hilfe der Ninja-Klassen (Ninja steht für *Numerical Intensive Java*). Auf *www.matheinfo.de* finden Sie Links zum Download dieser frei zur Verfügung stehenden Klassen für die Arbeit mit Vektoren, Matrizen in Java. Auch die *BLAS*, die *Basic Linear Algebra Subroutines*, ist in *Ninja* enthalten.

Hier der Java-Code:

```
import com.ibm.math.array.*;    // import the Array package

public class backsub {
  private static void backSubstitution(doubleArray2D A, doubleArray1D b)
  {
    // Rücksubstitution für eine gegebene obere Dreiecksmatrix
    // und eine rechte Seite. Das Ergebnis steht anschließend auf der
    // übergebenen rechten Seite!
    int i, j, n = b.size();
    double summe;

    for (i=n-1; i>=0; i--) {
         summe = b.get(i);
         for (j=i+1; j<n; j++) {
            summe -= A.get(i,j)*b.get(j);
         }
         b.set(i,summe/A.get(i,i));
    }
  }

  public static void main(String[] args) {
    int n = 3;                 // problem size
    doubleArray1D b = new doubleArray1D(n);
    doubleArray1D x = new doubleArray1D(n);
    doubleArray2D A = new doubleArray2D(n,n);

    initArrays(A,b);

    // b auf x kopieren, damit später die Probe durchgeführt werden kann
    x.assign(b);

    backSubstitution(A, x);

    System.out.println("Die berechnete Lösung x");
    System.out.println("x[0] = "+x.get(0));
    System.out.println("x[1] = "+x.get(1));
    System.out.println("x[2] = "+x.get(2));

    if (validate(A,x,b))
        System.out.println("Die Probe war erfolgreich!");
  }
}
```

Mit Hilfe der Funktion `initArrays` werden die Koeffizientenmatrix und die rechte Seite besetzt. Achten Sie darauf, dass die Funktion `backSubstitution` den übergebenen Vektor `b` überschreibt!

```
private static void initArrays(doubleArray2D A, doubleArray1D b) {
    b.set(0, 1.0); b.set(1, 1.0); b.set(2, 0.6);
```

```
    A.set(0,0, 1.0); A.set(0, 1, -1.0); A.set(0, 2, 0.0);
    A.set(1,0, 0.0); A.set(1, 1, 5.0);  A.set(1, 2, -3.0);
    A.set(2,0, 0.0); A.set(2, 1, 0.0);  A.set(2, 2, 1.8);
}
```

Für diese Werte, also

$$A = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 5 & -3 \\ 0 & 0 & 1,8 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 0,6 \end{pmatrix}$$

erhält man die Lösung

$$\mathbf{x} = \begin{pmatrix} 1,4 \\ 0,4 \\ 0,3333 \end{pmatrix}.$$

Mit der folgenden Funktion `validate` wird die Probe durchgeführt:

```
  private static boolean validate(doubleArray2D A, doubleArray1D x,
                                  doubleArray1D b) {
    // Probe der berechneten Lösung
    boolean ok = true;
    double eps = 1e-09, error;
    int n = x.size();
    doubleArray1D bb = new doubleArray1D(n);

    // Berechne Ax mit Hilfe von dgemv der BLAS!
    Blas.dgemv(Blas.NoTranspose,1.0,A,x,0.0,bb);

    for (int i=0; i<n; i++) {
         error = Math.abs(b.get(i)-bb.get(i));
            if (error > eps) {
               ok = false;
            }
    }
    return ok;
    }
```

7. Schreiben Sie eine Funktion in der Programmiersprache Ihrer Wahl, die die Gauß-Elimination für eine Matrix `A` und eine rechte Seite `b` durchführt. Testen Sie diese Funktion an selbst gewählten linearen Gleichungssystemen und mit der Funktion aus Aufgabe 6!

   Lösung:
   Mit der Rückwärtssubstitution aus der Lösung zu Aufgabe 6 und dem folgenden Quellcode können lineare Gleichungssysteme wie

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 6 & -3 \\ 0 & -3 & 3,6 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

   gelöst werden. Die obere Dreiecksmatrix aus der Lösung von Aufgabe 6 ist das Ergebnis der Gauß-Elimination für diese erweiterte Koeffizientenmatrix. Die Besetzung der Matrix muss entsprechend angepasst werden. Ganz wichtig: die Funktion `gausselimination` überschreibt das obere Dreieck; also muss für die Probe die Originalmatrix gesichert werden!

```
public class gauss {

  public static void main(String[] args) {
    int n = 3;                  // problem size
    doubleArray1D b = new doubleArray1D(n);
```

```
  doubleArray1D x = new doubleArray1D(n);
  doubleArray2D A = new doubleArray2D(n,n);
  doubleArray2D Aalt = new doubleArray2D(n,n);

  initArrays(A,b);

  // b auf x kopieren, damit später die Probe durchgeführt werden kann
  x.assign(b);
  // Das gleiche für A
  Aalt.assign(A);

  // Die Gauss-Zerlegung aufrufen
  gaussElimination(A,x);

  // Rückwärtsauflösung durchführen
  backSubstitution(A, x);

  System.out.println("Die berechnete Lösung x");
  System.out.println("x[0] = "+x.get(0));
  System.out.println("x[1] = "+x.get(1));
  System.out.println("x[2] = "+x.get(2));

  if (validate(Aalt,x,b))
      System.out.println("Die Probe war erfolgreich!");
  else
      System.out.println("Die Probe hat nicht gepasst!");
}

/*!
 * Gauß-Elimination für eine quadratische Matrix und einer
 * gegebenen rechten Seite. Beide Argumente werden überschrieben!
 */
private static void gaussEelimination(doubleArray2D A, doubleArray1D b)
{
  int i, j, k, n = b.size();

  for (j=0; j<n; j++)
      for (i=j+1; i<n; i++) {
          for (k=j+1; k<n; k++)
              A.set(i,k, A.get(i,k) - (A.get(i,j)/A.get(j,j))*A.get(j,k));
          b.set(i,b.get(i) - (A.get(i,j)/A.get(j,j))*b.get(j));
      }
}

private static void initArrays(doubleArray2D A, doubleArray1D b) {

  b.set(0, 1.0); b.set(1, 0.0); b.set(2, 0.0);

  A.set(0,0, 1.0);  A.set(0, 1, -1.0); A.set(0, 2, 0.0);
  A.set(1,0, -1.0); A.set(1, 1, 6.0);  A.set(1, 2, -3.0);
  A.set(2,0, 0.0);  A.set(2, 1, -3.0);  A.set(2, 2, 3.6);
}
```

Die obere Dreiecksmatrix kann man sich auch ausgeben lassen; insgesamt erhält man eine Ausgabe wie:

```
Der erste Spalte nach Gauß
A[0,0] = 1.0
Der zweite Spalte nach Gauß
A[0,1] = -1.0
A[1,1] = 5.0
Der dritte Spalte nach Gauß
A[0,2] = 0.0
A[1,2] = -3.0
A[2,2] = 1.8000000000000003
Die berechnete Lösung x
x[0] = 1.4
x[1] = 0.39999999999999997
x[2] = 0.33333333333333326
Die Probe war erfolgreich!
```

8. Die *Hilbert-Matrix* ist eine quadratische Matrix mit $h_{ij} = \frac{1}{i+j-1}, 1 \leq i, j \leq n$. Invertieren Sie H_n mit Hilfe der Gauß-Elimination und Rücksubstitution aus Aufgabe 7. Vergleichen Sie Ihre Ergebnisse mit der exakten Inversen H_n^{-1}, die durch

$$h_{ij}^{-1} = (-1)^{i+j}(i+j-1)\binom{n+i-j}{n-j}\binom{n+j-1}{n-i}\binom{i+j-2}{i+1}^2.$$

gegeben ist!

Lösung:
Die Inverse der Hilbertmatrix H_n kann mit Hilfe der Gauss-Elimination in den letzten beiden Aufgaben gelöst werden. Als rechte Seiten verwendet man die n kanonischen Einheitsvektoren; dann erhalten Sie die jeweiligen Spalten der Matrix.

Wenn Sie zur Klasse der letzten Aufgabe noch eine Funktion beifügen, die die Hilbertmatrix besetzt wie

```
  private static doubleArray2D hilbertmatrix(int n) {
    doubleArray2D hn = new doubleArray2D(n,n);
    int i,j;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            hn.set(i, j, 1.0/(i+j+1.0));
    return hn;
  }
```

dann erhält man beispielsweise für $n = 3$ die erste Spalte

```
Die berechnete erste Spalte der Inversen für n=3:
x[0] = 9.000000000000043
x[1] = -36.000000000000234
x[2] = 30.000000000000224
Die Probe war erfolgreich!
```

Die Probe-Funktion ist offensichtlich zufrieden.

Das Ganze geht gut, bis Sie in den Bereich von $n = 11$ und $n = 12$ kommen. Das liegt daran, dass die Hilbertmatrix ein Beispiel einer schlecht-konditionierten Matrix ist. Die Rundungsfehler, die bei der Gauß-Elimination unweigerlich auftreten, werden mit einem immer größeren Faktor, der Kondition der Matrix, multipliziert. Ist diese Kondition größer wie die

Maschinengenauigkeit, kann das Ergebnis nicht mehr sinnvoll sein. Die Kondition der Hilbertmatrix steigt exponentiell an; bei $n = 11$ ergibt sich auf Cygwin und JDK1.4 noch die Ausgabe

```
Die berechnete erste Spalte der Inversen für n=11:
x[0] = 120.89701771433465
x[1] = -7249.145595417011
x[2] = 141286.79323580535
x[3] = -1318138.0520180147
x[4] = 6917893.199034784
x[5] = -2.2130997809192467E7
x[6] = 4.477813681939939E7
x[7] = -5.755993554357505E7
x[8] = 4.5560038581912935E7
x[9] = -2.024568544551592E7
x[10] = 3864540.6361000347
Die Probe war erfolgreich!
```

Dabei vergleicht die Probe-Funktion das Ergebnis von $A\mathbf{x}$ mit $\mathbf{b}$ mit einer Genauigkeit von 10^{-9}. Bei $n = 12$ ergibt sich auf meinem Rechner dann

```
Die berechnete erste Spalte der Inversen für n=12:
x[0] = 141.5584489814937
x[1] = -9986.581068394009
x[2] = 230509.64239294577
x[3] = -2570134.73345685
x[4] = 1.6325579265833031E7
x[5] = -6.4343784039349094E7
x[6] = 1.6454066371450692E8
x[7] = -2.777840891763152E8
x[8] = 3.0733657354688287E8
x[9] = -2.1432348592910305E8
x[10] = 8.544011827110982E7
x[11] = -1.4842115901764408E7
Die Probe hat nicht gepasst!
```

Für noch größere Werte von n werden die Ergebnisse immer unsinniger ...

9. In vielen Anwendungen tauchen lineare Gleichungssysteme mit *tridiagonaler* Koeffizientenmatrix auf. Für $n = 5$ hat die tridiagonale Koeffizientenmatrix die Form

$$A = \begin{pmatrix} a_1 & b_1 & 0 & 0 & 0 \\ c_1 & a_2 & b_2 & 0 & 0 \\ 0 & c_2 & a_3 & b_3 & 0 \\ 0 & 0 & c_3 & a_4 & b_4 \\ 0 & 0 & 0 & c_4 & a_5 \end{pmatrix}.$$

Wenn die Gauß-Elimination für eine solche Koeffizientenmatrix ohne Zeilenvertauschung durchführbar ist, dann muss eine LU-Zerlegung existieren, bei der L eine bidiagonale untere und U eine bidiagonale obere Dreiecksmatrix ist. Stellen Sie einen Algorithmus auf, der für eine gegebene $n \times n$-Tridiagonalmatrix diese LU-Zerlegung berechnet wenn keine Zeilenvertauschungen nötig sind. Implementieren Sie den Algorithmus in der Programmiersprache Ihrer Wahl und testen Sie die Funktion an selbst gewählten Beispielen!

Lösung:
Die LU-Zerlegung einer Tridiagonalmatrix muss aus zwei Matrizen bestehen, die jeweils eine Haupt- und eine Nebendiagonale besitzen. Das kann man leicht verifizieren; nehmen Sie an, es wäre falsch, dann sieht man schnell, dass das Produkt LU keine Tridiagonalmatrix ist.

Es ist also

$$\begin{pmatrix} a_1 & b_1 & 0 & 0 & 0 \\ c_1 & a_2 & b_2 & 0 & 0 \\ 0 & c_2 & a_3 & b_3 & 0 \\ 0 & 0 & c_3 & a_4 & b_4 \\ 0 & 0 & 0 & c_4 & a_5 \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & l_2 & 1 & & \\ & & l_3 & 1 & \\ & & & l_4 & 1 \end{pmatrix} \begin{pmatrix} m_1 & r_1 & & & \\ & m_2 & r_2 & & \\ & & m_3 & r_3 & \\ & & & m_4 & r_4 \\ & & & & m_5 \end{pmatrix}.$$

Die Unbekannten Elemente l_i, m_i und r_i sind dann durch Koeffizientenvergleich bestimmbar:

$$\begin{aligned} a_1 = m_1, \quad & b_1 = r_1 \\ c_1 = l_1 m_1, a_2 = l_1 r_1 + m_2, & b_2 = r_2 \\ c_2 = l_2 m_2, a_3 = l_2 r_2 + m_3, & b_3 = r_3 \\ c_3 = l_3 m_3, a_3 = l_3 r_3 + m_4, & b_4 = r_4 \\ c_4 = l_4 m_4, a_4 = l_4 r_4 + m_5. & \end{aligned}$$

Also bestimmen sich die Unbekannten immer in der Reihenfolge $m_1, r_1, l_1, m_2, r_2, l_2$ und so weiter. Dabei ist einfach $r_i = b_i$. Dann kann das in einer Schleife formuliert werden:

$$\begin{aligned} m_1 &= a_1, \\ l_1 = \frac{c_1}{m_1}, m_2 &= a_2 - l_1 b_1, \\ l_2 = \frac{c_2}{m_2}, m_3 &= a_3 - l_2 b_2, \ldots \end{aligned}$$

Auch die Auflösung der beiden bidiagonalen Gleichungssysteme kann einfach formuliert werden. Die Vorwärtsauflösung für das Gleichungssystem $L\mathbf{y} = \mathbf{d}$ ist gegeben durch

$$y_1 = d_1, y_i = d_i - l_{i-1} y_{i-1}, i = 2, \ldots, n.$$

Zur Berechnung von y_2 benötigt man also nur die erste Komponente der rechten Seite, d_1 und l_1. Das bedeutet, dass Sie die Vorwärtsauflösung in die Schleife für die Berechung der beiden Faktoren L und U integrieren können.

Anschließend wird noch mit einer Rückwärtsauflösung das Gleichungssytem $U\mathbf{x} = \mathbf{y}$ gelöst:

$$x_n = \frac{y_n}{m_n}, x_i = \frac{1}{m_i}\left(y_i - r_i x_{m+1}\right), i = n-1, \ldots, 1.$$

```
/*!
  * LU-Zerlegung und Auflösung für ein tridiagonales Gleichungssystem.
  * Die Matrix ist durch die Vektoren a, b und c gegeben.
  *
  * Der Vektor a wird überschrieben!
  */
 private static void LUandSolve(doubleArray1D c, doubleArray1D a,
                                doubleArray1D b, doubleArray1D x,
                                doubleArray1D d)
 {
   int j, n = d.size();
   doubleArray1D l = new doubleArray1D(n);

   x.set(0,d.get(0));
   for (j=1; j<n; j++) {
       l.set(j-1, c.get(j-1)/a.get(j-1));
       a.set(j, a.get(j)-l.get(j-1)*b.get(j-1));
       x.set(j, d.get(j)-l.get(j-1)*x.get(j-1));
```

```
    }

    // Es wird nicht überprüft, ob die auftretenden Diagonalelemente
    // ungleich Null sind!#
    x.set(n-1, x.get(n-1)/a.get(n-1));
    for (j=n-2; j>=0; j--) {
        x.set(j, (x.get(j)-b.get(j)*x.get(j+1))/a.get(j));
    }
  }
```

Für das tridiagonale Gleichungssystem

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 2 & 1 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}$$

ist die exakte Lösung gegeben durch

$$\mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Dies wird auch durch den Code berechnet:

```
Die berechnete Lösung x
x[0] = 1.0
x[1] = 1.0
x[2] = 1.0
```

10. Implementieren Sie die *LU*-Zerlegung mit Scheinskalierung und Spaltenpivotwahl in der Programmiersprache Ihrer Wahl. Lösen Sie damit das lineare Gleichungssystem mit der 20×20-Matrix A mit

$$a_{ij} = \begin{cases} 2, & i + j = 21, \\ \cos(0,1 \cdot i + 0,05 \cdot j + 0,04 \cdot i \cdot j), & \text{sonst} \end{cases}$$

und der rechten Seite $b_i = 1, 1 \leq i \leq 20$.

Lösung:
Für diese Aufgabe müssen *drei* Funktionen implementiert werden. Neben der *LU*-Zerlegung eine Vorwärtssubstitution für das Gleichungssytem $L\mathbf{y} = P\mathbf{b}$ mit der Permutationsmatrix P und einer Rückwärtssubstition für $U\mathbf{x} = \mathbf{y}$.

Die Rückwärtssubstitution kann aus den bisherigen Aufgaben übernommen werden. Die Vorwärtssubstitution ist analog zu programmieren. Die Permutationsmatrix wird als Vektor gespeichert. Ist $\mathbf{p}_i = k$, dann müssen die Zeilen i und k vertauscht werden. Insgesamt erhält man dann folgenden Quellcode für diese beiden Auflösungen:

```
private static void forwardSubstitution(doubleArray2D A, doubleArray1D b,
                                        intArray1D p)
  {
    // Vorwärtssubstitution für eine gegebene untere Dreiecksmatrix
    // und eine rechte Seite. Das Ergebnis steht anschließend auf der
    // übergebenen rechten Seite! Im Feld p steht eine
    // eventuell durchgeführte Zeilenvertauschung.
    //
    // Die Diagonale von A wird implizit auf 1 gesetzt!
    int i, j, n = b.size();
    double h, c;
```

```
  // Zeilenvertauschung in der rechten Seite
  for (i=0; i<n-1; i++) {
      if (p.get(i) != i) {
         h = b.get(i);
         b.set(i, b.get(p.get(i)));
         b.set(p.get(i),h);
      }
  }

  for (i=1; i<n; i++) {
      c = b.get(i);
      for (j=0; j<i; j++)
          c -= A.get(i,j)*b.get(j);
      b.set(i,c);
  }

}

private static void backSubstitution(doubleArray2D A, doubleArray1D b)
{
  // Rücksubstitution für eine gegebene obere Dreiecksmatrix
  // und eine rechte Seite. Das Ergebnis steht anschließend auf der
  // übergebenen rechten Seite!

  int i, j, n = b.size();
  double summe;

  for (i=n-1; i>=0; i--) {
       summe = b.get(i);
       for (j=i+1; j<n; j++) {
          summe -= A.get(i,j)*b.get(j);
       }
       b.set(i,summe/A.get(i,i));
  }
}
```

Wie bisher wird auch hier die rechte Seite überschrieben; also müssen Sie diesen Vektor vorher sichern, wenn Sie nochmal die Probe durchführen möchten.

Die *LU*-Zerlegung hat die Matrix *A* und die Permutationsmatrix *P* als Argument. Auch hier steht nach dem Aufruf der Funktion auf *A* das Ergebnis der *LU*-Zerlegung. Die Diagonale von *L* wird implizit auf 1 gesetzt.

```
/*!
 * LU für eine quadratische Matrix und einer
 * gegebenen rechten Seite. Beide Argumente werden überschrieben!
 *
 * Die Zerlegung wird mit Spaltenpivotsuche und Scheinskalierung
 * durchgeführt. Die Zeilenvertauschungen werden auf dem Integer-Feld
 * p abgelegt und werden der Vorwärts-Auflösung übergeben.
 */
private static void LUzerlegung(doubleArray2D A, intArray1D p)
{
  int i, j, k, n = p.size();
  double q, s, h, max, summe;

  // Den Permutationsvektor initialisieren
  for (i=0; i<n; i++) p.set(i,i);
```

```
    for (k=0; k<n-1; k++) {

        // Die Spaltensumme berechnen
        max = 0.0;
        for (i=k; i<n; i++) {
            summe = 0.0;
            for (j=k; j<n; j++) {
                summe += Math.abs(A.get(i,j));
            }

            // Spaltenpivotsuche mit Scheinskalierung
            q = Math.abs(A.get(i,k))/summe;
            if (q > max) {
                max = q;
                p.set(k, i);
            }
        }
        if (max == 0) {
            // Throw exeption!
            return;
        }

        // Vertauschen der Zeilen, falls erforderlich
        if (p.get(k) != k) {
            for (j=0; j<n; j++) {
                h = A.get(k, j);
                A.set(k, j, A.get(p.get(k),j));
                A.set(p.get(k), j, h);
            }
        }

        // Elimination
        for (i=k+1; i<n; i++) {
            A.set(i, k, A.get(i,k)/A.get(k,k));
            for (j=k+1; j<n; j++) {
                A.set(i, j, A.get(i,j)-A.get(i,k)*A.get(k,j));
            }
        }
    }
}
```

Wenn Sie diese Funktionen auf die Matrix aus Aufgabe 7 anwenden, dann wird keine Zeilenvertauschung durchgeführt.

Im folgenden Code-Fragment sehen Sie die Besetzung der vorgegebenen Matrix und das Hauptprogramm:

```
private static void initArrays(doubleArray2D A, doubleArray1D b) {
  int i, j, n=b.size();

  b.assign(1.0);
  for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
      if (i+j==21)
          A.set(i, j, 2.0);
      else
          A.set(i, j,
          Math.cos(0.1*(double)i+0.05*(double)j+
                    0.04*(double)i*(double)j));
  }
```

```
  }

  public static void main(String[] args) {
    int i, n = 20;                // problem size
    doubleArray1D b = new doubleArray1D(n);
    doubleArray1D x = new doubleArray1D(n);
    doubleArray2D A = new doubleArray2D(n,n);
    doubleArray2D Aalt = new doubleArray2D(n,n);
    intArray1D p = new intArray1D(n);

    initArrays(A,b);

    // b auf x kopieren, damit später die Probe durchgeführt werden kann
    x.assign(b);
    // Das gleiche für A
    Aalt.assign(A);

    // Die Gauss-Zerlegung aufrufen
    LUzerlegung(A, p);

    // Ausgabe des Permutationsvektors
    System.out.println("Der Permutationsvektor der LU-Zerlegung");
    for (i=0; i<n; i++)
        System.out.println("p[" + i + "] = "+p.get(i));

    // Die Vorwärtssubstitution für Ly = Pb
    forwardSubstitution(A, x, p);

    // Rückwärtsauflösung durchführen
    backSubstitution(A, x);

    System.out.println("\nDie berechnete Lösung x");
    for (i=0; i<n; i++)
       System.out.println("x[" + i + "] = "+x.get(i));

    if (validate(Aalt,x,b))
        System.out.println("\nDie Probe war erfolgreich!");
    else
        System.out.println("\nDie Probe hat nicht gepasst!");

  }
```

Damit erhalten Sie die folgenden Ausgaben der Permutationsmatrix und der erhaltenen Lösung; die Probefunktion wird dabei erfolgreich durchgeführt:

```
Der Permutationsvektor der LU-Zerlegung
p[0] = 1
p[1] = 16
p[2] = 19
p[3] = 18
p[4] = 17
p[5] = 15
p[6] = 14
p[7] = 13
p[8] = 12
p[9] = 11
p[10] = 10
p[11] = 11
p[12] = 12
```

```
p[13] = 13
p[14] = 14
p[15] = 15
p[16] = 17
p[17] = 17
p[18] = 18
p[19] = 19

Die berechnete Lösung x
x[0] = 2.536362184292622
x[1] = -1.7691770080297828
x[2] = 0.0871597654482613
x[3] = 0.06404845762858202
x[4] = 0.045846727606265444
x[5] = 0.02852318961741817
x[6] = 0.012378367951613147
x[7] = 6.161281447016294E-4
x[8] = -0.004428505321647933
x[9] = -0.003952624773600349
x[10] = -0.0012861512578206155
x[11] = 7.37151462158858E-4
x[12] = 0.0010712592221248283
x[13] = 3.816974115166879E-4
x[14] = -6.382871244859799E-5
x[15] = 2.3828117914932964E-4
x[16] = 8.403897600632952E-4
x[17] = 0.0011846824049227677
x[18] = 0.0010839925860701616
x[19] = 6.192427701304958E-4

Die Probe war erfolgreich!
```

11. Invertieren Sie mit der *LU*-Zerlegung mit Scheinskalierung und Spaltenpivotwahl die Hilbertmatrix für verschiedene Werte von n. Vergleichen Sie Ihre Ergebnisse mit den Ergebnissen aus Aufgabe 8!

    Lösung:
    Hier wird jetzt die *LU*-Zerlegung aus der Aufgabe 10 auf die Hilbertmatrix angewandt. Wieder ist für $n = 3$ die Invertierung (immer am Beispiel der ersten Spalte der inversen Hilbertmatrix durchgeführt) ohne Zeilenvertauschung möglich:

```
Der Permutationsvektor der LU-Zerlegung
p[0] = 0
p[1] = 1
p[2] = 2

Die berechnete Lösung x
x[0] = 9.000000000000043
x[1] = -36.000000000000234
x[2] = 30.000000000000224

Die Probe war erfolgreich!
```

    Auch für $n = 11$ ist alles noch in Ordnung:

```
Der Permutationsvektor der LU-Zerlegung
p[0] = 0
p[1] = 1
p[2] = 2
```

```
p[3] = 3
p[4] = 4
p[5] = 5
p[6] = 6
p[7] = 7
p[8] = 8
p[9] = 9
p[10] = 10

Die berechnete Lösung x
x[0] = 120.89701771573164
x[1] = -7249.14559544495
x[2] = 141286.79323595553
x[3] = -1318138.0520184098
x[4] = 6917893.199035297
x[5] = -2.2130997809192814E7
x[6] = 4.477813681939951E7
x[7] = -5.755993554357506E7
x[8] = 4.556003858191293E7
x[9] = -2.024568544551592E7
x[10] = 3864540.6361000347

Die Probe war erfolgreich!
```

Es wird keine Zeilenvertauschung durchgeführt; die Ergebnisse sind identisch! Auch für größere Werte verändert sich nichts mehr.

12. Beweisen Sie durch Nachrechnen, dass alle für 2×2-Determinanten aufgestellten Regeln auch für den Fall 3×3 gelten!

Lösung:
Die Regeln können durch Nachrechnen nachgewiesen werden.

Eine Determinante wird mit einem Faktor λ multipliziert, indem eine beliebige Zeile oder Spalte der Determinante mit λ multipliziert wird. Sinnvoller Weise multiplizieren wir die Zeile, nach der entwickelt wird, beispielsweise die erste Zeile:

$$\begin{vmatrix} \lambda a_{11} & \lambda a_{12} & \lambda a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = \lambda a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - \lambda a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + \lambda a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} = \lambda det(A).$$

Werden beispielsweise zwei Zeilen vertauscht, dann ändert die Determinante ihr Vorzeichen:

$$\begin{vmatrix} a_{21} & a_{22} & a_{23} \\ a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = -a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} - a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} = -det(A).$$

Auch alle weiteren Regeln ergeben sich auf diese Weise.

13. Berechnen Sie $D_1 = \begin{vmatrix} 1 & -2 & 7 \\ 0 & 3 & 2 \\ 5 & -1 & 4 \end{vmatrix}$, $D_2 = \begin{vmatrix} 1 & 2 & -3 & 5 \\ 0 & 12 & 0 & 1 \\ 1 & 0 & -1 & 2 \\ -1 & 2 & 2 & 1 \end{vmatrix}$ und $D_3 = \begin{vmatrix} 1 & 5 & 5 & 0 \\ -2 & 1 & -2 & 3 \\ 0 & 1 & 1 & 0 \\ 1 & 2 & 4 & 1 \end{vmatrix}$.

Lösung:
D_1 kann mit der Regel von Sarrus oder mit dem Entwicklungssatz berechnet werden. Es ist $D_1 = 1 \cdot 3 \cdot 4 + (-2) \cdot 2 \cdot 5 + 7 \cdot 0 \cdot (-1) - 7 \cdot 3 \cdot 5 - 1 \cdot 2 \cdot (-1) - (-2) \cdot 0 \cdot 4 = 12 - 20 - 105 + 2 = -111$.

D_2 ist (Entwicklung nach der ersten Spalte)

$$D_2 = \begin{vmatrix} 12 & 0 & 1 \\ 0 & -1 & 2 \\ 2 & 2 & 1 \end{vmatrix} + \begin{vmatrix} 2 & -3 & 5 \\ 12 & 0 & 1 \\ 2 & 2 & 1 \end{vmatrix} + \begin{vmatrix} 2 & -3 & 5 \\ 12 & 0 & 1 \\ 0 & -1 & 2 \end{vmatrix} = 102.$$

D_3 ist (Entwicklung nach der vierten Zeile)

$$D_3 = 3 \cdot \begin{vmatrix} 1 & 5 & 5 \\ 0 & 1 & 1 \\ 1 & 2 & 4 \end{vmatrix} = 3 \cdot 2 = 6.$$

14. Berechnen Sie die Determinante von A^{-1} für $A = \begin{pmatrix} 1 & 0 & 3 & 4 \\ -2 & 1 & 0 & 3 \\ 1 & 4 & 1 & 5 \\ 0 & 2 & 2 & 0 \end{pmatrix}$.

Lösung:
Um die Determinante einer Inversen zu berechnen wird die $det(A)$ berechnet; $det(A^{-1}) = \frac{1}{det(A)} = \frac{1}{142}$.

Bei der Berechnung der Determinante von A kann vorteilhaft nach der vierten Zeile entwickelt werden, es ist

$$det(A) = 2 \cdot \begin{vmatrix} 1 & 3 & 4 \\ -2 & 0 & 3 \\ 1 & 1 & 5 \end{vmatrix} - 2 \cdot \begin{vmatrix} 1 & 0 & 4 \\ -2 & 1 & 3 \\ 1 & 4 & 5 \end{vmatrix} = 2 \cdot 28 - 2 \cdot (-43) = 142.$$

Möglich wäre natürlich auch, die Inverse mit Gauß-Jordan und dann die Determinante zu berechnen.

15. Berechnen Sie Eigenwerte und Eigenvektoren der Matrizen $A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$ und $B = \begin{pmatrix} 4 & 0 & 1 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}$.

Lösung:
Die Eigenwerte von A sind gegeben durch die Nullstellen des charakteristischen Polynoms $(1-\lambda)^2 - 4 = 1 - 2\lambda + \lambda^2 - 4 = \lambda^2 - 2\lambda - 3$. Als Nullstellen erhält man $\lambda_1 = -1$ und $\lambda_2 = 3$. Die Eigenvektoren sind dann als Lösung der zugehörigen Gleichungssysteme gegeben durch $\mathbf{x}_1 = (-1,1)^T, \mathbf{x}_2 = (1,1)^T$.

Das charakteristische Polynom von B ist gegeben durch $(1-\lambda)((4-\lambda)(1-\lambda)^2 + 2)$; möglich ist beispielsweise die Entwicklung nach der ersten Zeile. Dann sind die Nullstellen gegeben durch $\lambda_1 = 1, \lambda_2 = 2, \lambda_3 = 3$. Die zugehörigen Eigenvektoren sind $\mathbf{x}_1 = (0,1,0)^T, \mathbf{x}_2 = (-1,2,2)^T, \mathbf{x}_3 = (-1,1,1)^T$.

16. Bestimmen Sie die Eigenwerte von A^{25} für $A = \begin{pmatrix} -1 & -2 & -2 \\ 1 & 2 & 1 \\ -1 & -1 & 0 \end{pmatrix}$.

Lösung:
Die Eigenwerte von A^{25} sind die 25-te Potenz der Eigenwerte von A. Es gilt $\lambda_1 = -1, \lambda_2 = 1, \lambda_3 = 1$.

17. Beweisen Sie, dass eine 2×2-Matrix A genau dann zu einer Diagonalmatrix ähnlich ist, wenn $(a_{11} - a_{22})^2 + 4a_{12}a_{21} > 0$ erfüllt ist.

Lösung:
Die Matrix A muss zwei reelle Eigenwerte haben, dann kann die Konstruktion wie für die Beispielmatrix im Buch durchgeführt werden. Mit der Formel von Vieta folgt nach Aufstellen

des charakteristischen Polynoms die Behauptung:

$$det(A-\lambda I) = \begin{vmatrix} a_{11}-\lambda & a_{12} \\ a_{21} & a_{22}-\lambda \end{vmatrix} \\ = (a_{11}-\lambda)(a_{22}-\lambda) - a_{12}a_{21} = (a_{11}a_{12} - a_{12}a_{21}) - \lambda(a_{11}+a_{22}) + \lambda^2.$$

Die Diskrimante ist dann größer Null, wenn

$$D = b^2 - 4ac = (a_{11}+a_{22})^2 - 4(a_{11}a_{22} - a_{12}a_{21}) = (a_{11}-a_{22})^2 + 4a_{12}a_{21} > 0$$

erfüllt ist.

18. Berechnen sie auf der Basis der LU-Zerlegung der Hilbertmatrix die Determinante $det(H_n)$ und vergleichen Sie die Werte mit dem exakten Wert

$$det(H_n) = \frac{(\prod_{i=2}^{n-1} i!)^4}{\prod_{i=2}^{2n-1} i!} \approx 2^{-2n^2}.$$

Lösung:
Die Determinante ergibt sich als Produkt der Diagonalelemente der Matri U; denn es ist $det(A) = det(LU) = det(L)det(U) = det(U)$.

Man könnte die Berechnung der Determinanten direkt in die LU-Zerlegung integrieren; hier ist eine Lösung, die einfach das Ergebnis der bereits verwendeten Funktion für die LU-Zerlegung aufgreift. Wichtig ist, dass Sie daran denken, dass sich bei jeder bei der Zerlegung durchgeführten Zeilenvertauschung sich das Vorzeichen der Determinante umkehrt!

```
/*!
 * Berechnung der Determinanten einer Matrix auf der Basis
 * einer LU-Zerlegung
 */
private static double LUdeterminante(doubleArray2D A, intArray1D p) {
  int i, n = p.size();
  double det;

  det = 1.0d;

  // Für jede Zeilenvertauschung bei der LU-Zerlegung
  // bändert die Determinante einmal das Vorzeichen.
  for (i = 0; i<n; i++) {
      det *= A.get(i,i);
      if (p.get(i) != i) det = -det;
  }
  return det;
}
```

Für kleine n ist die Schätzung der Determinanten nicht sehr gut, aber beispielsweise für $n = 5$ erhält man die folgende Ausgabe, wenn der Vergleich auf Gleichheit mit einer Genauigkeit von 10^{-9} durchgeführt wird:

```
Die berechnete Determinante für n=5 ist 3.749295132517303E-12
Die Schätzung für n=5 ist 8.881784197001244E-16
Die Probe war erfolgreich!
```

Beachten Sie, dass die Determinanten der Hilbertmatrizen exponentiell gegen Null konvergieren!

Hier nochmals das Hauptprogramm für diese Aufgabe:

```
public static void main(String[] args) {
  int i, n = 5;
  double det, estimate, eps = 1e-09d;
  doubleArray2D A = new doubleArray2D(n,n);
  intArray1D p = new intArray1D(n);

  A = hilbertmatrix(n);
  LUzerlegung(A, p);
  det = LUdeterminante(A, p);

  System.out.println("Die berechnete Determinante für n="
       + n +" ist " + det);

  // Die Schätzung für die Determinante der Hilbertmatrix berechnen
  estimate = Math.exp(-2.0*n*n*Math.log(2.0));
  System.out.println("Die Schätzung für n=" + n +" ist " + estimate);

  if (Math.abs(estimate-det) < eps)
          System.out.println("Die Probe war erfolgreich!");
  else
          System.out.println("Die Probe hat nicht gepasst!");

}
```