Programmieraufgabe Mathe

Determinante

Name		Titel des Kurse	Datum
Juri Wiechmann	571085		
Salim Alkhodor	570766	Mathematik für Medieninformatik 2	16.01.2020

Index

Einleitung

Zunächst haben wir ein wenig in der “testDet” Klasse geändert, sodass wir beide Methoden zur Berechnung der Determinanten jeweils anwenden. Zudem lassen wir nun im Falle einer nicht quadratischen Matrix trotzdem die erste Normalform berechnen. Dafür erweiterten wir dann das verantwortliche if-Statement. Wir haben zudem die “showMatrix” Methode in unsere “Det” Klasse kopiert um mit dessen Hilfe das Debuggen für die erste Normalform zu vereinfachen und zudem diese immer in der Konsole auszugeben, wenn wir sie berechnen. Zu erwähnen sei noch, dass wir in Eclipse gearbeitet hatten und somit eine extra Klasse schreiben müssen in welcher wir in der Main-Methode unsere Tests ausführen. Die überarbeitete “testDet” Klasse sieht dabei wie folgt aus:

```
public class testDet {
    public double[][] A;
    public int nrOfDigits;

    public testDet(String filename){
        test(filename);
    }
    public void test(String filename){
        A = readMatrixFromFile(filename);
        if (A==null) return;
        if (A.length!=A[0].length){
            System.out.println("Die Matrix in "+filename+" ist nicht
quadratisch.");
            System.out.println("A:");
            det.ersteNormalform(A);
            return;
        }
        nrOfDigits = 1;
        System.out.println("A:");
        showMatrix(A, nrOfDigits);
        System.out.println();
        det.nrOfMult = 0;
        System.out.println("det(A) = "+det.calcDet(A));
        System.out.println("Anzahl der Multiplikationen: "+det.nrOfMult);
        System.out.println("det(A) rekursiv = " + det.calcDetRec(A));
    }
    //Liest die quadratische Matrix aus einer Textdatei; s.
Programmieraufgaben.pdf bezÃ1/4glich des Formats.
    public static double[][] readMatrixFromFile(String filename){...
    }
    //Schreibt die Matrix M in die Konsole; die Koeffizienten werden auf
nrOfDigits Stellen gerundet.
    public void showMatrix(double[][] M, int nrOfDigits){...
}
```

Hauptteil

public class StartTest:

Wie oben schon gesagt haben wir eine Test-Klasse geschrieben welche Objekte vom Typ “testDet” erstellt und dem jeweiligen Pfad zum File. Wir benutzen hierbei Test.txt und Text1.txt wie vorgegeben:

```
public class StartTest {

	public static void main(String[] args) {

		String filePath0 = "Pathstuff/Test.txt";
		testDet test0 = new testDet(filePath0);

		String filePath1 = "Pathstuff/Test1.txt";
		testDet test1 = new testDet(filePath1);

		String filePath2 = "Pathstuff/Test2.txt";
		testDet test4x4 = new testDet(filePath2);

		String filePath3 = "Pathstuff/Test3.txt";
		testDet testRoundErrors1= new testDet(filePath3);

		String filePath4 = "Pathstuff/Test4.txt";
		testDet test3x3= new testDet(filePath4);

	}
}
```

public static double calcDetRecursiv(double[][] A)

Um die Determinante Rekursiv zu berechnen ließen wir uns das Script an geeigneter stelle nochmal durch:

L.4.1. Berechnung von Determinanten und erste Eigenschaften

L.4.1.1. Berechnung der Determinante durch Entwicklung nach der 1. Spalte

$\det : M(n\times n,\mathbb{R}) \to \mathbb{R}$ ist rekursiv wie folgt definiert:

I. Für $n=1$ und $A=(a)\in M(1\times 1,\mathbb{R})$ ist $\det(A)=a$.

II. Sei $n\geq 2$ und $\det(B)$ bereits definiert für alle $B\in M((n-1)\times(n-1),\mathbb{R})$.

Dann ist für $A=(a_{i,j})\in M(n\times n,\mathbb{R})$

$$\det(A)=\sum_{i=1}^{n}(-1)^{i+1}\cdot a_{i,1}\cdot\det(A_{i,1}),$$

wobei $A_{i,1}$ aus A durch Streichen von Zeile i und Spalte 1 entsteht.

(Entwicklung nach der 1. Spalte)

Name: $\det(A)$ ist die **Determinante** von A.

Andere Bezeichnung: $|A|$

Hierraus kann man sich schon ganz leicht einen Pseudocode erstellen. Für unserer Rekursive Methode ist unserer Abbruchbedingung also, dass der Rank der Matrix(n) = 1 ist. Wenn n ≥ 2 dann müssen mithilfe einer Summenformel welche sich Rekursiv verhält rechen bis n=1. Wie oben schon beschrieben wird am Ende der Formel immer mit einer Determinanten Multipliziert von der Matrix Ai,1, welche entsteht, wenn wir Zeile i und die erste Spalte aus A streichen. Wir wissen, dass wir solche Summenformeln mit leichtigkeit in Code umwandeln können mithilfe einer For-schleife:

```
public static double calcDetRec(double[][] A){

  double detA = 0;
  int n = A.length;

  //Wenn ursprungsmatrix grade ist
  if(n == 2)
        return (A[0][0] * A[1][1]) - (A[0][1] * A[1][0]);
  //Wenn ursprungsmatrix ungrade ist
  if(n==1)
        return A[0][0];
  for(int i = 0; i < n ; i++) {
        //showMatrix(A, 1);
        detA += Math.pow(-1, i+1)*A[i][0]*calcDetRec(remove(i, A));
  }
    return detA; // Durch Ihren Code ersetzen!
}
```

Hier geben wir den Wert schon raus, wenn unserer Matrix einen Rank von 2 hat. So sparen wir uns einen kompletten durchgang. Wir wenden dafür folgende Formel an:

$$\det\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} = a_{1,1} \cdot a_{2,2} - a_{1,2} \cdot a_{2,1}$$

Nun kommen wir jedoch zu einem entscheidenden Problem. Wenn wir diese Formel verwenden und der Rang der Matrix ungerade ist, dann ist das Vorzeichen der ausgerechneten Matrix falsch. Wenn wir Die Abbruchbedingung mit n = 1 festlegen und a11 ausgeben, dann ist das Vorzeichen bei graden Ursprungsmatrizen falsch. Wir haben sehr lange versucht mit hilfe von Debugging den Fehler zu finden, doch leider ohne Erfolg. Wir haben uns also schlussendlich dafür entschieden, einen Weg zu finden, dass wir das if-Statement nehmen, welches das richtige Ergebnis liefert. So sind wir nach einigen Überlegungen dazu gekommen folgenden Ansatz zu wählen. Wir implementierten ein field "rang" welches den rang in gewissen Punkten speichert. Nun erweiterten wir unserer if-Statements so, dass dies jeweils richtige if-Statement gewählt wird:

Ich hoffe hierzu sind keine weiteren erklärungen Notwendig, da die Line:
`detA += Math.pow(-1, i+1)*A[i][0]*calcDetRec(remove(i, A));`
offensichtliche aus der Summenformel abgeleitet ist.

```
if(n>rang)
      rang = n;
//Wenn ursprungsmatrix grade ist
if(n == 2 && rang%2 == 0) {
      rang = 2;
      return (A[0][0] * A[1][1]) - (A[0][1] * A[1][0]);
}

//Wenn ursprungsmatrix ungrade ist
if(n==1 && rang%2 != 0) {
      rang = 1;
      return A[0][0];
}
```

Nun haben wir jedoch noch das Problem, dass der rang nicht aktualisiert wird, wenn wir nach dem ersten Test eine Matrix berechnen wollen, welche einen kleineren Rang hat, so haben wir den rang immer auf 2 oder 1 gesetzt, dass bei jeder neuen Matrix über 2 der rang aktualisiert wird. Bei Matrizen mit n < 3 brauchen wir dies nicht, da wir direkt die Determinante ausgeben ohne rekursiven aufruf.

public static double[][] remove(int line, double [][]A)

In der "calcDetRec" Methode rufen wir die "remove" Methode auf, welche als Parameter die zu entfernende Zeile und die Matrix erhält. Es wird dabei automatisch davon ausgegangen, dass die 1. Spalte ebenso entfernt wird:

```
    private static double[][] remove(int line, double[][] A){
      int newN = A.length-1;

      double[][] Ai1 = new double[newN][newN];


      for(int i = 0, newi = 0; i<A.length;i++) {
            if(i == line)
                  continue;
            //j = 1 weil j = 0 die 1. Spalte ist und weg gelassen
wird
            for(int j = 1; j<A.length;j++) {
                  Ai1[newi][j-1] = A[i][j];
            }
            newi++;
      }
            return Ai1;
    }
```

Zunächst stellen wir den neuen Rank der Matrix fest mit "newN" und erstellen daraufhin unserer noch leere newNxnewN Matrix. Nun gehen wir simple durch A durch mit zwei for-Schleifen. Die äußere ist für die Zeilen und die innere für die Spalten zuständig. Wenn wir auf die zu entfernende Zeile treffen überspringen wir diese mithilfe eines if-Statements. Die erste Spalte überspringen wir einfach, indem wir in Index für die Spalten bei 1 starten lassen und nicht bei 0. Nun müssen wir dafür Sorgen, dass unsere Werte auch in die Richtige Zeile reingeschrieben werden. Dafür erstellen wir parallel einen Zähler: "newi" welcher die Zeilen von Ai1 zählt. Da sich die Spalten alle nur einen nach links verschieben müssen wir hier einfach nur "j-1" als Angabe setzen beim reinschreiben. Am Ende geben wir unserer Ai1 Matrix zurück und unserer rekursive Berechnung der Determinanten ist vollständig.

public static double calcDet(double[][] A)

Kommen wir nun zur Berechnung der Determinanten über die 1. Normalform und des Produktes der Diagonalelemente: Die Bildung des Produktes mit Hilfe einer for-Schleife ist hierbei trivial und wir nicht weiter erläutert:

```
//Berechnung mit 1. Normalform
public static double calcDet(double[][] A){

  A = ersteNormalform(A);
  double det = A[0][0];
  for(int i = 1; i<A.length;i++) {
        det *= A[i][i];
  }

    return det; // Durch Ihren Code ersetzen!
}
```

Um die erste Normalform zu berechnen haben wir eine extra Methode erstellt: “ersteNormalform” welche eine Matrix als Parameter erhält

```
public static double[][] ersteNormalform(double[][]A){

  double factor;

  int rows = A.length-1;
  int columns = A[0].length-1;

  //Geht durch jede Spalte
  for(int j = 0;j<rows;j++) {
        //Geht durch jede Zeile, fängt mit der 2. an
        for(int i = j+1; i<=rows;i++ ) {
              factor = -A[i][j]/A[j][j];
              //Geht die Zeile durch mit dem Faktor durch.
              for(int x = j; x<=columns;x++) {
                    A[i][x] = A[i][x]+(factor*A[j][x]);
              }
              nrOfMult++;
        }
  }
  System.out.println("Erste Normalform(A):");
  showMatrix(A, 1);
  return A;
}
```

Zuerst erstellen wir einen factor. Dieser soll speichern, mit welchen Wert wir unserer Zeile Multiplizieren müssen, um dann mit addition 0 an einer bestimmten Stelle heraus zu bekommen. Dann haben wir uns zur Übersicht nochmal die Spalten und Zeilen als Integer speichern lassen. Unserer Idee ist, dass wir Spaltenweise vorgehen, Zuerst setzen wir alle Werte der Spalte 1(Index 0) unterhalb der 1. Zeile(Index 0) zu 0. Dann gehen wir Stufenweise runter, heißt in Spalte 2(Index 1) sollen alle Werte zu 0 werden, welche sich unterhalb der 2. Zeile(Index 1) befinden. Wir gehen also in der äußeren For-Schleife durch alle Spalten. Dann gehen wir in der mittleren Schleifen über die noch nötigen Zeilen. Da wir nur die Werte unterhalb der Diagonalen zu 0 setzen müssen, fällt mit jedem Spaltenwechsel die oberste Zeile weg, da diese sich bereits in der 1. Normalform befindet. Hier rechnen wir auch den factor aus, den die jeweilige Zeile braucht um den vordersten Wert, welcher nicht 0 ist, auf 0 zu setzen. Dieser vorderste Werte ist zu jedem Zeitpunkt in der Spalte i in der Zeile j. Dieser Vorderste Wert muss dann addiert werden mit mit dem Wert eine Zeile darüber, welcher zuvor mit dem factor Multipliziert wird. So wird der vorderste Werte 0. Nun müssen wir jedoch die ganze Zeile mit dieser Rechnung versehen. Dafür ist dann unserer innerste Schleife zuständig. Diese geht durch den Rest der Zeile und verrechnet jeden Wert genauso wie den vordersten Wert. Zum ende hin haben wir dann so die 1. Normalform. Zum schluss lassen wir uns dann diese nochmal in unserer Konsole ausgeben. Wir hoffen, wir konnten diesen komplizierten Vorgang einigermaßen erläutern. Nun konnten wir also die 1. Normalform bilden und haben bei dieser, wie oben schon genannt, die Diagonalelemente Multiplizieren und damit die Determinante erhalten.

```
A:
   1,0   2,0   2,0   2,0   2,0
   1,0   3,0   1,0   1,0   1,0
   1,0   3,0   2,0   0,0   0,0
   1,0   3,0   2,0   4,0   1,0
   1,0   3,0   2,0   4,0   2,0

Erste Normalform(A):
   1,0   2,0   2,0   2,0   2,0
   0,0   1,0  -1,0  -1,0  -1,0
   0,0   0,0   1,0  -1,0  -1,0
   0,0   0,0   0,0   4,0   1,0
   0,0   0,0   0,0   0,0   1,0
det(A) = 4.0
Anzahl der Multiplikationen: 14
```

```
A:
   1,0   2,0   2,0   2,0   2,0
   1,0   3,0   1,0   1,0   1,0
   1,0   3,0   2,0   0,0   0,0
   1,0   3,0   2,0   4,0   1,0
   1,0   3,0   2,0   4,0   2,0

Erste Normalform(A):
   1,0   2,0   2,0   2,0   2,0
   0,0   1,0  -1,0  -1,0  -1,0
   0,0   0,0   1,0  -1,0  -1,0
   0,0   0,0   0,0   4,0   1,0
   0,0   0,0   0,0   0,0   1,0
det(A) = 4.0
Anzahl der Multiplikationen: 14
det(A) rekursiv = 4.0
Die Matrix in f:/Alles Mögliche/Uni/Mathe
A:
Erste Normalform(A):
     1,0     2,0     3,0     1,0   128,0
     0,0    -3,0    -9,0     1,0  -192,0
     0,0     0,0     1,0    -3,0     0,0
A:
   1,0   2,0   2,0   2,0
   1,0   3,0   1,0   1,0
   1,0   3,0   2,0   0,0
   1,0   3,0   2,0   4,0

Erste Normalform(A):
   1,0   2,0   2,0   2,0
   0,0   1,0  -1,0  -1,0
   0,0   0,0   1,0  -1,0
   0,0   0,0   0,0   4,0
det(A) = 4.0
Anzahl der Multiplikationen: 9
det(A) rekursiv = 4.0
A:
   1,0   2,0   2,0
   1,0   3,0   1,0
   1,0   3,0   2,0

Erste Normalform(A):
   1,0   2,0   2,0
   0,0   1,0  -1,0
   0,0   0,0   1,0
det(A) = 1.0
Anzahl der Multiplikationen: 5
det(A) rekursiv = 1.0
```

Wie viele Multiplikationen benötigen Ihre Methoden zur Berechnung eine n x n - Determinante?

Bei genauer Betrachtung ist es irrelevant wie viele Spalten unserer Matrix im Bezug auf die Anzahl der Multiplikationen hat. Sei eine m x n Matrix gegeben. Dann ergibt sich die Anzahl der Multiplikationen aus folgender Summenformel:

$$m - 1 + \sum_{i=1}^{m-1} i;$$

Wir müssen m-1 nochmal addieren, weil wir so oft am Ende bei der Multiplikation der Diagonalelemente multiplizieren müssen.
1. Beispiel:

$$Sei\ m = 5\ \ und\ n\ aus\ \mathbb{N}$$

$$darauf\ \ folgt:$$

$$4 + \sum_{i=i}^{4} i; = 4 + 1 + 2 + 3 + 4 = 14;$$

Wie oben in der Konsole zu sehen stimmt dies.

```
A:
   1,0   2,0   2,0   2,0
   1,0   3,0   1,0   1,0
   1,0   3,0   2,0   0,0
   1,0   3,0   2,0   4,0

Erste Normalform(A):
   1,0   2,0   2,0   2,0
   0,0   1,0  -1,0  -1,0
   0,0   0,0   1,0  -1,0
   0,0   0,0   0,0   4,0
det(A) = 4.0
Anzahl der Multiplikationen: 9
```

2. Beispiel:

$$Sei\ m = 4\ \ und\ n\ aus\ \mathbb{N}$$

$$darauf\ \ folgt:$$

$$3 + \sum_{i=i}^{3} i; = 1 + 2 + 3 = 9;$$

Auch hier können wir in der Konsole sehen, dass dies stimmt.

Finden sie eine Matrix A für welche die Rundungsfehler in Ihrer Berechnung der Determinante besonders groß wird.

Unser Ursprung Ansatz war, dass wir zunächst mit einer Nachkommastelle rangehen und gucken wieviele unsere Determinante hat. Dann würden wir es mit 2 Nachkommastellen pro Wert versuchen und so weiter. So hätten wir eine Relation herstellen können. Bei einer 5 x 5 Matrix wären das 5 Nachkommastellen. Bei 2 Nachkommastellen in einer 5 x 5 Matrix wären es 7 Nachkommastellen in der Determinante. Wir testeten ein wenig weiter herum und verwarfen diesen Ansatz schnell. Zudem hatten wir diese Information nicht aus unserem Programm, sondern von einer Seite(S1). In unserem Programm nämlich gab es Problem, welches ich schon öfter beim Arbeiten mit doubles beobachten durfte.

Wie wir sehen können, rundet Java bei doubles irgendwann nach der letzten Kommastelle auf eine Zahl runter mit ganz vielen neunen. Das richtige Ergebnis wäre hier: -1.86844. Somit können wir allgemeint sagen, dass es bei Java in diesem Beispiel schon bei nur einer Nachkommastelle zu Rundungsfehlern kommt. Es ist jedoch zu erwarten, dass diese größer werden wenn die die Anzahl der Nachkommastellen der einzelnen Werte erhöhen. Dabei wird bei der rekursiven Berechnung der der Rundungsfehler erst bei der 16. Nachkommastelle auftauchen. Eine genaue Aussage zu unserer Berechnung mithilfe der 1. Normalform können wir derzeit leider nicht treffen.

```
A:
   1,1   2,3   2,3   2,5   2,5
   1,2   3,2   1,2   1,2   1,3
   1,2   3,6   2,2   0,0   0,0
   1,2   3,4   2,2   4,4   1,1
   1,4   3,4   2,2   4,1   2,1

Erste Normalform(A):
   1,1   2,3   2,3   2,5   2,5
   0,0   0,7  -1,3  -1,5  -1,4
   0,0   0,0   1,8  -0,3  -0,5
   0,0   0,0   0,0   3,9   0,6
   0,0  -0,0   0,0   0,0  -0,4
det(A) = -1.8684399999999892
Anzahl der Multiplikationen: 10
det(A) rekursiv = -1.86843999999999
```

Wir versuchen noch einen anderen Ansatz und die Anzahl der Nachkommastellen. Allgemein kann man sagen, dass bei Multiplikation mit Nachkommastellen, die Anzahl der Nachkommastellen des Produktes maximal die Summe der Anzahl der Nachkommastellen des Faktors und des Multiplikators.
Beispiel:

Lass ξ *aus* $\mathbb{N}$ *die maximale Anzahl von Nachkommastellen sein*.

Lass x, y *aus* $\mathbb{R}$ *zwei Zahlen sein f ür die gilt*:

$\xi(x) = 2$

$\xi(y) = 2$

und lass $z = x \cdot y$,

dann gilt:

$\xi(z) = \xi(x) + \xi(y) = 4$

Nun können wir anhand der Anzahl von den Multiplikationen Ungefähr abschätzen, wie viele Nachkommastellen wir max in der Determinante haben können:

Sei A eine n x n Matrix wobei n aus $\mathbb{N}$

Sei x beliebig und ein Element aus A und f ür x gilt $\xi(x) = y$

Dann ist die maximal Mö gliche Anzahl an Nachkommastellen wie f olgt zu berechnen:

$$\xi(A) = m - 1 + \sum_{i=1}^{m-1} i \cdot y;$$

Beispiel:

Sei A eine 5 x 5 Matrix und x ein beliebiges Element aus A .

Lass $\xi(x) = 2$ *sein, so gilt;*

$$\xi(A) = 5 - 1 + \sum_{i=1}^{5-1} i \cdot 2 = 2(4 + 1 + 2 + 3 + 4) = 28;$$

Wir hoffen dies reicht aus um die Aufgabenstellung zu beantworten.

Rückschau

Juri:

- Wir hatten zunächst ein paar Fragen, weil uns einige Dinge unklar waren, aber Herr Thiel hatte uns Zeitnah diese Fragen beantworten. Das Programmieren war nicht besonders viel dafür aber jedoch einiges an Kopfarbeit. Die Schleifen zum Rausstreichen der Zeilen und Spalten und das richtige Einsetzen forderten einiges an Überlegungen. Genauso wie das Umformen in die 1. Normalform.

Salim:

- Wir haben die Aufgabe der Determinanten ausgesucht , da wir das Thema super gut verstanden haben. Die war aber nicht einfach (auch nicht so schwer) , besonders für mich , aber danke an Juri, der mir dabei geholfen hat. Wir haben auf jeden Fall viel davon gelernt.

Soure:

S1:
https://matrixcalc.org/de/#Gaussian-elimination%28%7B%7B1,2,2,2%7D,%7B1,3,1,1%7D,%7B1,3,2,0%7D,%7B1,3,2,4%7D%7D%29

Code:

```java
package Determinanten;

public class StartTest {

	public static void main(String[] args) {

		String filePath0 = "f:/Alles Mögliche/Uni/Mathe/src/Determinanten/Test.txt";
		testDet test0 = new testDet(filePath0);

		String filePath1 = "f:/Alles Mögliche/Uni/Mathe/src/Determinanten/Test1.txt";
		testDet test1 = new testDet(filePath1);

		String filePath2 = "f:/Alles Mögliche/Uni/Mathe/src/Determinanten/Test2.txt";
		testDet test4x4 = new testDet(filePath2);
		/*
		String filePath3 = "f:/Alles Mögliche/Uni/Mathe/src/Determinanten/Test3.txt";
		testDet testRoundErrors1= new testDet(filePath3);
		*/
		String filePath4 = "f:/Alles Mögliche/Uni/Mathe/src/Determinanten/Test4.txt";
		testDet test3x3= new testDet(filePath4);

	}
}
package Determinanten;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class testDet {
    public double[][] A;
    public int nrOfDigits;

    public testDet(String filename){
        test(filename);
    }
    public void test(String filename){
        A = readMatrixFromFile(filename);
```

```
        if (A==null) return;
        if (A.length!=A[0].length){
            System.out.println("Die Matrix in "+filename+" ist nicht
quadratisch.");
            System.out.println("A:");
            det.ersteNormalform(A);
            return;
        }
        nrOfDigits = 1;

        System.out.println("A:");
        showMatrix(A, nrOfDigits);
        System.out.println();

        det.nrOfMult = 0;
        System.out.println("det(A) = "+det.calcDet(A));
        System.out.println("Anzahl der Multiplikationen:
"+det.nrOfMult);
        System.out.println("det(A) rekursiv = " + det.calcDetRec(A));
    }

    //Liest die quadratische Matrix aus einer Textdatei; s.
Programmieraufgaben.pdf bezÃ1/4glich des Formats.
    public static double[][] readMatrixFromFile(String filename){
        ArrayList<String> stringList = new ArrayList<String>();
        try{
            BufferedReader br = new BufferedReader(new
FileReader(filename));
            String line = br.readLine();
            while (line!=null){
                stringList.add(line);
                line = br.readLine();
            }
            br.close();

            String[] parts = stringList.get(0).split("  ");
            int m = stringList.size(), n = parts.length;
            double[][] M = new double[m][n];
            for (int i=0; i<m; i++){
                parts = stringList.get(i).split("  ");
                for (int j=0; j<n; j++) M[i][j] =
Double.valueOf(parts[j]);
            }
            return M;
        }
```

```
        catch(IOException e){
            System.out.println(""+e);
            return null;
        }
    }

    //Schreibt die Matrix M in die Konsole; die Koeffizienten werden auf
nrOfDigits Stellen gerundet.
    public void showMatrix(double[][] M, int nrOfDigits){
        int m = M.length;
        int n = M[0].length;
        //boolean hasNoNegativeEntry = true;
        double max = 0.0;
        for (int j=0; j<n; j++){
            for (int i=0; i<m; i++){
                if (M[i][j]>max) max = M[i][j];
                //if (M[i][j]<0.0) hasNoNegativeEntry = false;
            }
        }
        int l;
        if (max==0) l = 5;
        else l = (int) Math.log10(Math.abs(max))+nrOfDigits+5;//+1: log,
+1: sign, +1: point, +1
        if (nrOfDigits==0) l--;
        //if (hasNoNegativeEntry) l--;

        String f, s;
        f = "%"+l+"."+nrOfDigits+"f";
        for (int i=0; i<m; i++){
            s = "";
            for (int j=0; j<n; j++){
                s = s+String.format(f, M[i][j]);
            }
            System.out.println(s);
        }
    }
}


package Determinanten;


public class det {
    public static int nrOfMult = 0;
    private static int rang = 0;
```

```java
//Berechnung mit 1. Normalform
public static double calcDet(double[][] A){

  A = ersteNormalform(A);
  double det = A[0][0];
  for(int i = 1; i<A.length;i++) {
        det *= A[i][i];
        nrOfMult++;
  }

    return det; // Durch Ihren Code ersetzen!
}
//Rekursive Berechnung mit Def. L.4.1.1 Skript
public static double calcDetRec(double[][] A){

  double detA = 0;
  int n = A.length;

  if(n>rang)
        rang = n;
  //Wenn ursprungsmatrix grade ist
  if(n == 2 && rang%2 == 0) {
        rang = 2;
        return (A[0][0] * A[1][1]) - (A[0][1] * A[1][0]);
  }

  //Wenn ursprungsmatrix ungrade ist
  if(n==1 && rang%2 != 0) {
        rang = 1;
        return A[0][0];
  }

  for(int i = 0; i < n ; i++) {
        //showMatrix(A, 1);
        detA += Math.pow(-1, i+1)*A[i][0]*calcDetRec(remove(i, A));
  }
    return detA; // Durch Ihren Code ersetzen!
}
private static double[][] remove(int line, double[][] A){
  int newN = A.length-1;

  double[][] Ai1 = new double[newN][newN];


  for(int i = 0, newi = 0; i<A.length;i++) {
```

```
            if(i == line)
                  continue;
            //j = 1 weil j = 0 die 1. Spalte ist und weg gelassen wird
            for(int j = 1; j<A.length;j++) {
                  Ai1[newi][j-1] = A[i][j];
            }
            newi++;
      }
            return Ai1;
    }
    public static void showMatrix(double[][] M, int nrOfDigits){
        int m = M.length;
        int n = M[0].length;
        //boolean hasNoNegativeEntry = true;
        double max = 0.0;
        for (int j=0; j<n; j++){
            for (int i=0; i<m; i++){
                if (M[i][j]>max) max = M[i][j];
                //if (M[i][j]<0.0) hasNoNegativeEntry = false;
            }
        }
        int l;
        if (max==0) l = 5;
        else l = (int) Math.log10(Math.abs(max))+nrOfDigits+5;//+1: log,
+1: sign, +1: point, +1
        if (nrOfDigits==0) l--;
        //if (hasNoNegativeEntry) l--;

        String f, s;
        f = "%"+l+"."+nrOfDigits+"f";
        for (int i=0; i<m; i++){
            s = "";
            for (int j=0; j<n; j++){
                s = s+String.format(f, M[i][j]);
            }
            System.out.println(s);
        }
    }
    public static double[][] ersteNormalform(double[][]A){

      double factor;

      int rows = A.length-1;
      int columns = A[0].length-1;
```

```
      //Geht durch jede Spalte
      for(int j = 0;j<rows;j++) {
            //Geht durch jede Zeile, fängt mit der 2. an
            for(int i = j+1; i<=rows;i++ ) {
                  factor = -A[i][j]/A[j][j];
                  //Geht die Zeile durch mit dem Faktor durch.
                  for(int x = j; x<=columns;x++) {
                        A[i][x] = A[i][x]+(factor*A[j][x]);
                  }
                  nrOfMult++;
            }
      }
      System.out.println("Erste Normalform(A):");
      showMatrix(A, 1);
      return A;
    }
}
```