Facharbeit im Grundkurs Informatik

Implementation des Gauß-Algorithmus

Georg-Büchner-Gymnasium

Joel Mantik

Pablo Sonnauer

März 2023

Inhaltsverzeichnis

Kapitel 1

Einleitung

> "The simplest model in applied mathematics is a system of linear equations. It is also by far the most important."
> - GILBERT STRANG

Die lineare Algebra stellt eines der wichtigsten Felder der Mathematik dar. Wie aus dem Zitat des Mathematikers Gilbert Strang hervorgeht, sind die linearen Gleichungssysteme trotz ihrer Simplizität eines der wichtigsten Konzepte in der angewandten Mathematik. Eines der faszinierendsten Lösungsverfahren dieser Gleichungssysteme ist der Gauß-Algorithmus, weil er das Lösen auf einfachste Weise möglich macht. Seit der Entdeckung im frühen 19. Jahrhundert, durch den Mathematiker Carl Friedrich Gauß, stellt er eines der elementarsten Lösungsverfahren linearer Gleichungssysteme dar. Auch in der Informatik spielt der Algorithmus eine große Rolle, beispielsweise in der Computergrafik. Aus diesen Gründen wird im Folgenden eine Implementationsmöglichkeit vorgestellt. Zuerst wird der mathematische Hintergrund erläutert, anschließend der Quellcode des Algorithmus vorgestellt und Vor- bzw. Nachteile abgewägt. Weiter werden spezifische Anwendungsmöglichkeiten in der Informatik aufgezeigt und zuletzt ein Fazit gezogen.

Kapitel 2

Lineare Gleichungssysteme

Ein lineares Gleichungssystem ist eine Sammlung von Gleichungen, in denen jede Unbekannte mit höchstens dem ersten Grad vorkommt. Es kann in der Form $A_x = b$ geschrieben werden, wobei A eine $m * n$ Matrix ist, x ein n-dimensionaler Vektor von Unbekannten und b ein m-dimensionaler Vektor von Konstanten ist. Ein allgemeines lineares Gleichungssystem lässt sich wie folgt definieren.: [Gra21]

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = &\quad b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = &\quad b_2 \\
&\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = &\quad b_m \qquad (2.1)
\end{aligned}$$

Das Ziel eines solchen linearen Gleichungssystems ist es, eine Lösung für x zu finden, die alle Gleichungen erfüllt. Hierbei gibt es drei Arten von Lösungen:

1. Das Gleichungssystem hat genau *eine* Lösung; es gibt genau eine Lösung, welche alle Gleichungen im System erfüllt. Die Lösungsmenge ist z. B. : $\mathbb{L} = \{(x, y, z)|(1, 2, 3)\}$.

2. Das Gleichungssystem hat *keine* Lösung, wenn es keine Lösung gibt, die alle Gleichungen erfüllt. Die Lösungsmenge ist eine leere Menge: $\mathbb{L} = \emptyset$.

3. Das Gleichungssystem hat *unendlich viele* Lösungen, wenn es mehrere Lösungen gibt die alle Gleichungen im System erfüllen. Hierbei sind die verschiedenen

Variablen möglicherweise voneinander abhängig. Die Lösungsmenge sieht beispielsweise wie folgt aus:
$\mathbb{L} = \{(x, y, z)|(x = ay + z, y \in \mathbb{R}, z \in \mathbb{R})\}$.

Kapitel 3

Gauß'sches Eliminationsverfahren

Gegeben sei das Allgemeine lineare Gleichungssystem 2.1. Gesucht ist nun die Menge der $(x_1, \ldots, x_n) \in \mathbb{R}^n$, die alle Gleichungen erfüllen. Um diese Menge zu finden, geht man folgendermaßen vor:

1. Das Gleichungssystem in Die erweiterte Koeffizientenmatrix (A, b) bringen. [Fis14]:

$$(A, b) := \begin{pmatrix} a_{11} & \ldots & a_{1n} & b_1 \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & \ldots & a_{mn} & b_m \end{pmatrix} \tag{3.1}$$

2. (A, b) durch elementare Zeilentransformationen, also Vertauschen von Zeilen, Multiplikation einer Zeile mit einer Zahl $\neq 0$ oder Addition des Vielfachen von einer Zeile zu einer anderen Zeile auf Zeilenstufenform bringen.

$$\begin{pmatrix} 1 & a_{12} & a_{13} & \cdots & a_{1,n-1} & a_{1n} \\ 0 & 1 & a_{23} & \cdots & a_{2,n-1} & a_{2n} \\ 0 & 0 & 1 & \cdots & a_{3,n-1} & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & a_{n-1,n} \end{pmatrix} \tag{3.2}$$

Eine $m * n$-Matrix (A, b), wie in 3.2 wird *Zeilenstufenform* genannt. Das a steht für eine beliebige reelle Zahl, alle anderen Plätze ohne ein a sind von Nullen besetzt. Der erste von null verschiedene Eintrag in jeder Zeile ist 1. Dieser Eintrag wird das Pivot-Element der Zeile genannt.
Das Pivot-Element der $(i + 1)$-ten Zeile steht immer rechts des Pivot-Elements der I-ten Zeile, und alle Einträge oberhalb eines Pivot-Elements sind gleich null. Vgl. [Zwe22].

3. Nun lassen sich durch Rücksubstitution die Werte für die Variablen ermitteln. Man teilt die letzte Spalte durch den Wert des Koeffizienten, sodass die Variable alleine steht. Der gefundene Wert wird dann in die nächst höhere Zeile eingesetzt, dann wird analog zum ersten Schritt vorgegangen.

Kapitel 4

Erläuterung des Quellcodes

Für die im Folgenden dargelegte Implementation und Analyse des Algorithmus wurde die Programmiersprache "Java" verwendet.

Das Implementationsdiagramm zeigt den Aufbau:

Gauß
- datei: String - countSpalten: int - NORMALFALL: boolean - koeff: double[][] - solMatrix: double[][] - zeilenElemente: String[] - countZeilen: int - SONDERFALL: boolean
+ main(args: String[]): void + einlesen(): void + ausgabe(): void + gaussAlgo(): void + multiplyAndAdd(lineOne: int, lineTwo: int, factor: double):void - checkWievielNullZeilen(): boolean - checkObNull(): boolean

Es wurden aussagefähige Methodennamen gewählt. In der Main Methode des Programms wird ein Parameter übernommen, welcher die Datei mit der Koeffizientenmatrix annimmt. Außerdem werden in dieser die Methoden `einlesen()`, `ausgabe() und gaussAlgo()` ausgeführt. Die Methode `ausgabe()` gibt in der Main Methode die Eingangsmatrix, die triangularisierte Matrix und die Lösungsmatrix aus.
Die Methode `einlesen()` liest die Koeffizientenmatrix aus der Datei ein, dabei wird eine Liste erstellt, in welcher alle Zeilen der eingelesenen Datei gespeichert werden. Die Datei kann Kommentare und Leerzeilen enthalten, daher werden durch eine For-Schleife alle Leerzeichen und Kommentare entfernt. Des Weiteren werden die Kommata zu Punkten gemacht, um das Funktionieren des Algorithmus auch auf anderen Systemen zu gewährleisten. Weiterhin wird die Anzahl der Spalten in der Variable `countSpalten`, und die Anzahl der

Zeilen in `countZeilen` gespeichert und im Zuge dessen auch die *Eingangsmatrix* `koeff[][]`und die *Lösungsmatrix* `solMatrix[][]`initialisiert. Die Methode `ausgabe()` nimmt zwei Parameter an, das zweidimensionale Array `double mx` und den String `matrixName`. Beide werden durch eine For-Schleife auf der Konsole ausgegeben. Die Methode `multiplyAndAdd()`, welche die Parameter `lineOne, lineTwo (Integer)` und den `double factor` annimmt, wird verwendet, um die Koeffizienten zu einer Null zu machen. Dabei wird durch eine For-Schleife über die Spalten der Matrix iteriert und dabei der Koeffizient an der Stelle von `lineTwo` und `spalten` auf den Wert Null gebracht. Dies geschieht, indem die Matrix an der Stelle `lineOne` und `spalten` mit dem Parameter `factor` multipliziert und zu dem zu verändernden Koeffizienten addiert wird. Weiter gibt es die Methode `checkObNull()`, welche den Parameter `zeile (Integer)` annimmt und über die Eingangsmartix iteriert und prüft, ob ein Koeffizient den Wert Null hat. Zu diesem Fall wird `true` zurückgegeben, andernfalls `false`. Diese Methode wird später für die Sonderbehandlung von Matrizes verwendet, die entweder eine Null-Zeile haben oder gleich null sind. Anknüpfend gibt es noch die Methode `checkWievielNullZeilen()`. Diese iteriert, ausgehend von der letzten Zeile, über alle Zeilen und prüft, ob alle Koeffizienten in der Zeile gleich null sind. Für jede gefundene Null-Zeile wird ein Counter `nullCounter` erhöht. Wenn diese Zählvariable am Ende der Iteration ungleich Null ist, wird die Konstante `SONDERFALL` zurückgegeben und die Anzahl der Null-Zeilen auf der Konsole ausgegeben, andernfalls wird die Konstante `NORMALFALL` zurückgegeben. Wenn die Matrix an jeder Stelle eine Null hat, wird ebenfalls interveniert und der Sonderfall auf der Konsole ausgegeben. Die Methode arbeitet sehr effektiv, denn durch die Pivotisierung kann nach erfolgter triangularisierung von der letzten Zeile aus iteriert werden, denn wenn es Null-Zeilen gibt so sind diese am Ende.

Die Methode `gaussAlgo()`, führt den eigentlichen Algorithmus aus. Als Erstes wird die Eingangsmartix, also `koeff[][]`, durch eine For-Schleife, in die Lösungsmatrix `solMatrix[][]` kopiert, damit am Ende beide ausgegeben werden können. Anschließend findet die Pivotisierung der Matrix statt. Dies geschieht, indem zuerst, eine Variable `maxZeile (Integer)` erzeugt und mit der Zählvariable `zeile` initialisiert wird. Diese wird verwendet, um die Zeile mit dem größten Koeffizienten zu finden. In der darauf folgenden For-Schleife findet die eigentliche Pivotisierung statt. Sie sucht in der aktuellen Spalte nach dem Element mit dem größten, absoluten Wert und speichert diesen, wenn der größere Wert in der Zeile unter der aktuellen Zeile ist in der Variable `maxZeile`. Anschließend wird geprüft ob die zuvor initialisierte maxZeile ungleich dem Wert der Variable `zeile` ist. Wenn dem so ist, werden die Zeilen mithilfe einer temporären Variable getauscht, sodass am Ende die Zahl mit dem größten Koeffizienten oben steht. Als nächstes wird der Faktor berechnet, mit dem die Koeffizienten gleich null werden. Dies geschieht durch die nächste For-Schleife, welche den Faktor `factor` berechnet, indem die Lösungsmatrix mit negativem Vorzeichen an der Stelle der Zählvariable i und `zeile` (die Zählvariable der ersten for-Schleife), durch die Lösungsmatrix an der Stelle `zeile, zeile` geteilt wird. Dieser Faktor wird dann an die Methode `multiplyAndAdd` zusammen mit `zeile, i` als Parameter übergeben. Nachdem die For-Schleife durchgelaufen ist, wird die entstandene triangularisierte Matrix ausgegeben.

Anschließend prüft der Algorithmus die triangularisierte Matrix auf Sonderfälle, also ob die Matrix Null-Zeilen hat oder komplett gleich Null ist, indem durch eine Wenn-dann prüfung abgefragt wird, ob die Methode `checkWievielNullZeilen` einen Sonderfall zurückgibt. Wenn sie dies tut, bricht der Algorithmus ab.

Der letzte Schritt, wird verwendet, um die Matrix durch Rücksubstitution zu lösen. Eine For-Schleife läuft von der letzten Spalte bis zur ersten. Zuerst wird in dieser Schleife eine Variable `double diagonale` erstellt, in welcher das aktuelle Diagonalelement gespeichert wird. In der nächsten Zeile iteriert eine weitere For-Schleife über die Matrix und teilt bei jeder Iteration den Wert des aktuellen Koeffizienten durch den Wert in `diagonale`. Dadurch entsteht die Dreiecksform, bei welcher jedes Element in dieser den Wert Eins haben soll. Weitergehend wird der Wert des Tupels in der Variable `double result` gespeichert. Durch eine weitere For-Schleife wird die über alle Zeilen unter der aktuellen iteriert, bei jeder

Iteration wird der Wert der rechten Zeile um das Produkt aus dem Wert von `result` und dem Wert des Elements aus der aktuellen Spalte der Zeile subtrahiert. Als Letztes wird noch der Wert der aktuellen Zeile auf null gesetzt, damit das Gleichungssystem in der Dreiecksform bleibt.

Kapitel 5

Abwägungen

5.1 Vorteile des Algorithmus

Der Gauß-Algorithmushat klar den Vorteil das er bei Gleichungssystemen mit wenig Koeffizienten sehr effektiv ist, d.h er kann die Lösung schnell und einfach berechnen. Auch ist der Algorithmus in der Programmierung relativ leicht umzusetzen, da er nach einfachen Schemata funktioniert, bei welchem wenige Schritte benötigt werden.

5.2 Nachteile des Algorithmus

Zum einen kann es durch die Verwendung vom Datentyp Double und die mehrfache Rechnung mit denselben Werten zu Rundungsfehlern kommen. Zum anderen hat der Algorithmus bei komplizierten Martizen mit vielen Zeilen und Spalten eine große Laufzeit, die Worstcase Laufzeit würde aufgrund von doppelten For-Schleifen $O(n^2)$ betragen. Nachteile des Algorithmus sind zum einen, dass bei schlecht konditionierten Gleichungssystemen Rundungsfehler auftreten können. Dies sieht man auch in der vorliegenden Implementation, wobei durch die Verwendung vom Datentyp double und dem rechnen mit denselben gerundeten Zahlen noch größere Rundungsfehler entstehen können. Außerdem kann der Algorithmus bei vielen Koeffizienten eine sehr hohe Rechenleistung in Anspruch nehmen.

Kapitel 6

Anwendungsbereiche des Gauß'schen Eliminationsverfahrens in der Informatik

Der Algorithmus spielt in der Informatik aufgrund seiner Simplizität eine tragende Rolle in vielen Teilbereichen, wie der Kryptografie, der Computergrafik oder der Datenanalyse. Eine Auswahl von Anwendungsbereichen wird im Folgenden umrissen.

6.1 Kryptografie

Das Gauß'sches Eliminationsverfahrenkann verwendet werden, um modulare Gleichungen zu lösen, die in der Kryptografie häufig auftreten. Zum Beispiel wird das Verfahren bei der Berechnung von Schlüsseln in asymmetrischen Kryptosystemen wie RSA eingesetzt. Hierbei wird mit dem Gauß-Algorithmus eine Primfaktorzerlegung durchgeführt, die dann verwendet, um Schlüssel in asymmetrischen Kryptosystemen zu verwenden. Mehr Informationen in [KK10] vgl. Kapitel 11.4.3.

6.2 Computergrafik

In der 3D-Computergrafik werden 4x4-Matrizen verwendet, um Objekte im Raum zu transformieren. Diese Matrizen enthalten Informationen über Translationen, Rotationen und Skalierungen von Objekten. Das Gauß'sche Eliminationsverfahren wird verwendet, um die inverse Matrix zu berechnen, die dann verwendet wird, um die Transformationsoperationen umzukehren. Die inverse Matrix wird auch dazu verwendet, um Normale von 3D-Objekten zu transformieren, um sie in eine konsistente Richtung zu bringen, was wichtig ist für Beleuchtungs- und Schattierungsberechnungen. Vgl. [scr].

Kapitel 7

Fazit

Die lineare Algebra und das Lösen linearer Gleichungssysteme sind fundamentale Konzepte in der angewandten Mathematik und finden auch in der Informatik weitreichende Anwendungen. Der Gaußsche Algorithmus ist eines der elementarsten und faszinierendsten Verfahren zur Lösung linearer Gleichungssysteme und hat seit seiner Entdeckung im frühen 19. Jahrhundert große Bedeutung erlangt. Der mathematische Hintergrund und die Definitionen linearer Gleichungssysteme sowie des Gaußschen Algorithmus wurden erläutert, und eine Implementierungsmöglichkeit des Algorithmus wurde vorgestellt. Der Algorithmus hat den Vorteil, dass er das Lösen linearer Gleichungssysteme auf einfachste Weise ermöglicht, aber es gibt auch Nachteile wie die Notwendigkeit einer hohen Rechenleistung bei großen Matrizen. Dennoch sind die Anwendungsmöglichkeiten in der Informatik zahlreich. Zusammenfassend lässt sich sagen, dass der Gauß-Algorithmusein wichtiges Werkzeug für Mathematiker und Informatiker ist, um komplexe Probleme zu lösen und praktische Anwendungen zu ermöglichen.

Anhang

```java
/**
 * @author Joel Mantik
 */
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.List;


public class Gauss {
    static String datei; // der Parameter der Datei
    static double[][] koeff; // Initialierung der erweiterten Koeffizienten Matrix
    static double[][] solMatrix; // Initialierung der Loesungsmatrix
    static String[] zeilenElemente;
    static int countSpalten = 0;
    static int countZeilen;
    static final boolean SONDERFALL = true;
    static final boolean NORMALFALL = false;

    /**
     * main Methode in welcher alle Methoden ausgefuehrt werden
     * @param args liest die Datei mit der Matrix ein
     */
    public static void main(String[] args) throws IOException {
        /*
         * Hier wird der Dateiname als Parameter eingelesen
         */
        if (args.length < 1) {
            System.out.println("Bitte Dateiname als Parameter uebergeben!");
            return;
        } else {
            datei = args[0];
        }

        einlesen();
        ausgabe(koeff, "Eingangsmatrix");
        gaussAlgo1();
        ausgabe(solMatrix, "Loesungsmatrix");
    }

    /**
     * Diese Methode liest die Koeffizienten aus der Datei ein
     */
    private static void einlesen() throws IOException {

        List<String> f = Files.readAllLines(Paths.get(datei));
```

```java
        int zeilenIndex = 0;

        for (int i = 0; i < f.size(); i++) {
            String zeile = f.get(i);
             // Kommentare und Whitespaces der input Datei werden ignoriert
            if (zeile.isEmpty() || zeile.charAt(0) == '#') {
                continue;
            }

            zeilenElemente = zeile.replace(',', '.').split("\\s+"); // Kommata werden zu Punkten
            // Speichert Spaltenanzahl
            if (countSpalten == 0){
                countSpalten = zeilenElemente.length;
                koeff     = new double[countSpalten - 1][countSpalten];
                solMatrix = new double[countSpalten - 1][countSpalten];
            }
            for (int spaltenIndex = 0; spaltenIndex < countSpalten; spaltenIndex++) {
                String element = zeilenElemente[spaltenIndex];
                double wert = Double.parseDouble(element);
                // Der Koeffizientenmatrix werden die double Werte zugewiesen, welche in wert gespeichert wurden
                koeff[zeilenIndex][spaltenIndex] = wert;

            }

            zeilenIndex++;
         }
        /*double[][] hilfMat = koeff;
        if(hilfMat.length != countSpalten){
            System.out.println("Falsche Matrix!");
            return;
        }*/

        // Speichert Zeilenanzahl
        countZeilen = koeff.length;
        System.out.println(countSpalten);
        System.out.println(countZeilen);
    }

    /**
     * Methode ausgabe gibt die Eingangsmatrix und die Loesungsmatrix aus
     * @param mx nimmt die Eingangsmatrix oder die Loesungsmatrix an
     * @param matrixName nimmt den Namen der Matrix an
     */
    private static void ausgabe(double[][] mx, String matrixName){ // Ausgabe der Matrizen

        System.out.println("Die " + matrixName + ":");
        for (int j = 0; j < mx.length; j++) {
            for (int k = 0; k < mx[j].length; k++) {
                System.out.print(Math.round(mx[j][k] * 10000d) / 10000d + " "); // TODO: GILT NUR FUER NETTE ZAHLEN
            }
            System.out.println();
        }
        System.out.println();
    }


    /**
     * Die Methode gaussAlgo fuehrt den eigentlichen Algorithmus aus
     */
    private static void gaussAlgo1() {
```

```
        for (int i = 0; i < countZeilen; i++) {
            for (int j = 0; j < countSpalten; j++) {
                solMatrix[i][j] = koeff[i][j];
            }
        }
       // copyLine(0);// Erste Zeile der Originalmatrix wird in Loesungs Matrix kopiert
        //copyLine(1);
        //copyLine(2);
        /**
         * Diese for-Schleife triangularisiert die Matrix
         */
        for (int zeile = 0; zeile < countZeilen - 1; zeile++) {
            int maxZeile = zeile;

            /*
            *  Diese Schleife sucht in der aktuellen Spalte (durch die Variable zeile indiziert)
            *  nach dem Element mit dem groessten absoluten Wert und merkt sich die Zeilennummer dieses Elements in der Variable maxZeile.
            */
            for (int i = zeile + 1; i < countZeilen; i++) {
                // Der spaltenIndex der solMatrix rueckt nacht jeder Vertauschung einen weiter deswegen kann als spaltenIndex zeile benutzt werden
                if (Math.abs(solMatrix[i][zeile]) > Math.abs(solMatrix[maxZeile][zeile])) {
                    maxZeile = i;
                }
            }
            /*
            * Wenn das Element mit dem groessten absoluten Wert in einer anderen Zeile als in der aktuellen Zeile (maxZeile) gefunden wurde
            * werden die maxZeile und die oberste Zeile vertauscht vertauscht. Dies ist die Pivotisierung.
            */
            if (maxZeile != zeile) {
                double[] temp = solMatrix[zeile];
                solMatrix[zeile] = solMatrix[maxZeile];
                solMatrix[maxZeile] = temp;
            }

            // Die Koeffizienten werden durch x = -b/a 0
            for (int i = zeile + 1; i < countZeilen; i++) {
                // TODO: Vielleicht wird durch 0 geteilt
                double factor = -solMatrix[i][zeile] / solMatrix[zeile][zeile];
                multiplyAndAdd(zeile, i, factor);
            }
        }

        ausgabe(solMatrix, "Triangularisierte Matrix");

        /*
         * Untersuchung der Triangularisierten Matrix: 1.: wvl 0 Zeilen gibt es am Ende
         */
        if(checkWievielNullZeilen() == SONDERFALL){
            System.exit(-1);
        }

        /*
         *  Durch ruecksubstitution werden die gefundenen Werte RUECKWAERTS in die uebere Zeile eingesetzt
         *  und so x und y errechnet
         */
        for(int zeilen = countZeilen-1; zeilen >= 0; zeilen--){
            double diagonale = solMatrix[zeilen][zeilen];
            // TODO: Teilen durch Diagonale
            for(int spalte = 0; spalte < countSpalten; spalte++){
```

```java
                solMatrix[zeilen][spalte] /= diagonale;
            }
            double result = solMatrix[zeilen][solMatrix[zeilen].length - 1];
            for(int rueckZeilen = zeilen - 1; rueckZeilen >= 0; rueckZeilen--){
                solMatrix[rueckZeilen][countSpalten - 1] -= result * solMatrix[rueckZeilen][zeilen];
                solMatrix[rueckZeilen][zeilen] = 0.0;
            }
        }
    }



    /**
     * multiplyAndAdd bringt die Koeffizienten auf 0
     * @param lineOne  nimmt die 1. Zeile welche benutzt werden soll an
     * @param lineTwo  nimmt die 2. Zeile welche benutzt werden soll an
     * @param factor   der Wert mit dem multiplizert wird, so dass der Koeffzient 0 wird
     */
    private static void multiplyAndAdd(int lineOne, int lineTwo, double factor) { // x = -b/a wird durchgefuehrt
        for(int spalten = 0; spalten < solMatrix[lineOne].length; spalten++){
            solMatrix[lineTwo][spalten] = (solMatrix[lineOne][spalten] * factor) + solMatrix[lineTwo][spalten];
        }

    }

    /**
     * Die Methode prueft wieviele Nullzeilen die Matrix hat.
     * @return SONDERFALL wird zurckgegeben wenn die Matrix ein Sonderfall ist.
     * @return NORMALFALL wird sonst zurueckgegeben
     *
     *
     */

    private static boolean checkWievielNullZeilen() {
        int nullCounter = 0;
        for(int i = countZeilen - 1; i >= 0; i--){
            if(checkObNull(i) == true){
                nullCounter++;
            } else {
                if(nullCounter != 0){
                    System.out.println("Sonderfall gefunden: Es gibt " + nullCounter + " Nullzeilen " );
                    return SONDERFALL;
                }else{
                    return NORMALFALL;
                }
            }
        }
        System.out.println("Sonderfall gefunden, Matrix besteht aus Nullen!");
        return SONDERFALL;
    }

    /**
     * @param zeile Die Zeile welchhe geprueft wird
     * Die Methode prueft fuer jedes Element ob es den Wert Null hat
     */
    private static boolean checkObNull(int zeile){
        for(int i= 0; i < countSpalten; i++){
            if(solMatrix[zeile][i] != 0){
                return false;
            }
```

```
        }
        return true;
    }

}
```

Listing 7.1: Quellcode des Gauß-Algorithmus

Literatur

[Fis14] Gerd Fischer. *Lineare Algebra: Eine Einführung für Studien Anfänger*. Springer Spektrum, 2014.

[Gra21] Günther Gramlich. *Lineare Algebra: Eine Einführung*. Carl Hanser Verlag GmbH Co KG, 2021.

[KK10] Christian Karpfinger und Hubert Kiechle. *Kryptologie, Algebraische Methoden und Algorithmen*. Vieweg+Teubner, 2010.

[scr] scratchapixel. *The Perspective and Orthographic Projection Matrix*. Letzter Zugriff: 2023-03-26. URL: `https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/projection-matrix-introduction.html`.

[Zwe22] Prof. Dr. Sander Zwegers. *Lineare Algebra, Notizen zur Vorlesung*. Universitaet zu Koeln, Juli 2022.