FH-Aachen

Gauß-Verfahren für dünn besetzte Matrizen

COBOL

Björn Lüpschen & Philipp Kohl
31.01.2017

Inhaltsverzeichnis

Hinweise 3

1. Aufgabenstellung 4

2. Aufgabenanalyse 6

2.1 System 6

2.2 Programmdokumentation 6

2.3 Tests 6

3. Möglichkeiten zur Abspeicherung dünn besetzter Matrizen (n x m) 7

3.1 Standard Speicherart 7

3.2 Compressed Row Storage (CRS)/Compressed Column Storage (CCS) 7

4. Diskussion der Speicherformen einer n x m Matrix 9

4.1 Zu speichernde Elemente 9

4.1.1 Standard Speicherart 9

4.1.2 Compressed Row Storage 9

4.2 Verwaltungsaufwand 9

4.2.1 Standard Speicherart 9

4.2.2 Compressed Row Storage 9

4.3 Fazit 10

5. Systembeschreibung 11

5.1 Eingabe 11

5.1.1 Struktogramme 12

5.2 Verarbeitung 13

5.3 Beschreibung des Gauß'schen Eliminationsalgorithmus 13

5.6 Ausgabe 17

6. Benutzeranleitung 18

7. Beschreibung der Entwicklungsumgebung 18

8. Testfallbeschreibung 19

8.1 Anweisung-/Pfadvollständig 19

8.2 Grenzwertanalyse 20

8.3 Sonstige Tests 20

9. Code Listing 21

10. Ressourcen 21

11. Hilfsmittel 21

11.1 Literaturverzeichnis 21

11.2 Programme 21

Hinweise

- Der verfassende Autor steht oben auf der Seite
- Alle Grafiken sind ebenfalls im zip Ordner enthalten, da sie im Dokument meist zu klein sind
- Der Quellcode und die ausführbare Datei befindet sich im zip Ordner

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

1. Aufgabenstellung

Hausaufgabe

Gauß-Verfahren für dünn besetzte Matrizen

Team: Philipp Kohl und Björn Lüpschen

Problemstellung

Zu Lösung von Gleichungssystemen der Form Ax = b gibt es das Gauß-Eliminationsverfahren. Selbst für heutige IT-Systeme reicht die Speicherkapazität für „große n“ (n > 10.000) nicht immer aus.

Eine Reihe von mathematischen Problemen führt auf spezielle Matrizen (z. B. tridiagonale Matrizen), für die eine volle Abspeicherung der betreffenden Matrix nicht nötig ist. Es gibt aber auch Matrizen, über deren Struktur a prior nicht ausgesagt werden kann, außer dass sie dünn besetzt sind (z. B. Adjazenzmatrizen).

Eine nxn-Matrix heißt „dünn besetzt“, falls nicht mehr als 30% der Elemente von 0 verschieden sind.

Programmsystem

Schreiben Sie ein Programm, das ein gegebenes lineares Gleichungssystem für dünnbesetzte Matrizen nach dem Gaußschen Verfahren löst.

Bitte beachten Sie die folgenden Punkte:
Beschreiben Sie zwei Verfahren zur Abspeicherung von dünnbesetzten Matrizen. Wählen Sie eines dieser Verfahren aus und begründen Sie Ihre Wahl. Berücksichtigen Sie auch die Tatsache, dass bei einem Austausschritt neue Matrixelemente hinzukommen bzw. vorhandene gelöscht werden können. Geben Sie die Anzahl der gelöschten und hinzugekommenen Elemente an.

Um die Rundungsfehler möglichst klein zu halten, wählen Sie jeweils das betragsmäßig größte Element der Matrix als Pivotelement aus.
Die Eingabe von n, A und b erfolgt über eine Datei.
Das Einlesen ist abzubrechen, wenn nicht genügend Speicherplatz zur Verfügung steht bzw. wenn die Matrix nicht dünn besetzt ist.
Die Ausgabe einer Lösung – sofern vorhanden – erfolgt ebenfalls in eine Datei. Matrixelemente, die betragsmäßig kleiner einem vorgegebenen Epsilon (= 0,00000001) sind, werden als 0 betrachtet.

Testen Sie unter anderem folgendes Beispiel: $n = 30$

$$a_{ij} = \begin{cases} v(i) & \text{für } j = v(i) \\ 0 & sonst \end{cases}$$

$v = (30,23,25,26,7,8,9,20,18,1,15,27,16,28,4,19,3,22,5,6,29,10,11,24,14,13,21,12,2,17)$

Modularisieren Sie Ihr Programmsystem nach dem EVA-Prinzip sowie nach sinnvollen Funktionseinheiten.

Entwicklerdokumentation

Die ganzheitliche Entwicklerdokumentation sollte ein Inhaltsverzeichnis aufweisen und zu jedem Kapitel eine Autorenangabe erhalten.
Dokumentieren Sie jeweils zu Ihrem Teil (s. u.), Beschreibung der Algorithmen anhand eines selbstgewählten Beispiels sowie als Struktogramm, Programmeinheiten und mathematische Hintergründe.
Begründen Sie die Auswahl Ihrer Testfälle und diskutieren Sie sie.

Teil A

- Eingaben (inklusive Fehlerprüfung), Ausgabe der Ergebnisse jeweils über Dateien
- Prüfung, ob Speicherplatz ausreichend und ob Matrix dünn besetzt ist
- Beschreibung der Möglichkeiten zur Abspeicherung dünn besetzter Matrizen und Begründung der Wahl des Verfahrens, passende Datenstruktur

Teil B

- Implementierung des Gauß-Verfahrens mit Hilfe der bei Teil A gewählten Datenstruktur, dabei
- Berechnung der Lösung bzw. Meldung, falls nicht lösbar

Abzugeben sind:

Aufgabenstellung, Programmcodes als Listing und ausführbares Programmsystem, E-/A-Dateien der Tests, die oben beschriebene ganzheitliche Entwicklerdokumentation entweder in schriftlicher Form (Word-, oder pdf-Datei) oder als Präsentation (ppt-Datei, dann auch Darbietung der Präsentation)

Unterschriebene Eigenständigkeitserkärung mit Nennung der eigenen erstellten Programmteile.

2. Aufgabenanalyse

Anforderungen an das Gauß´sche Eliminationsverfahren nach der MASTER-Anforderungsschablone:

2.1 System

- Das Programm muss Matrizen nach dem Gauß‘schen Eliminationsverfahren lösen.
- Das Programm muss eine der diskutierten Speicherverfahren anwenden.
- Das Programm muss das betragsmäßig größte Element einer Spalte der Matrix als Pivotelement auswählen.
- Die Eingabe muss mit einer Datei erfolgen.
- In der Eingabe müssen die Dimension, eine Matrix(A) und die Ergebnismatrix(b) übergeben werden.
- Das Einlesen muss abgebrochen werden, wenn die Matrix nicht dünnbesetzt ist.
- Die gelöste Matrix muss in einer Datei ausgegeben werden.
- Matrixelemente, die betragsmäßig kleiner als 0,00000001 sind, müssen als 0 betrachtet werden.
- Das Programm muss nach dem EVA-Prinzip modularisiert werden.

2.2 Programmdokumentation

- In der Programmdokumentation müssen zwei Verfahren zur Abspeicherung von dünnbesetzten Matrizen beschrieben werden.
- In der Programmdokumentation müssen die Umstrukturierungsarbeiten (löschen und hinzukommende Elemente) diskutiert werden.
- Die Programmdokumentation muss ein Inhaltsverzeichnis aufweisen.
- Die Programmdokumentation muss in jedem Kapitel eine Autorenangabe erhalten.
- In der Programmdokumentation muss eine Beschreibung des Algorithmus enthalten sein.
- Die Beschreibung des Algorithmus muss anhand eines selbstgewählten Beispiels erfolgen.
- Der Algorithmus muss mit einem Struktogramm erklärt werden.
- Die Programmdokumentation muss den vollständigen Programmcode enthalten.

2.3 Tests

- Das Programm muss anhand von verschiedenen Testfällen getestet werden.
- Die Auswahl der Testfälle muss begründet werden.
- Die Auswahl der Testfälle muss diskutiert werden.

3. Möglichkeiten zur Abspeicherung dünn besetzter Matrizen (n x m)

3.1 Standard Speicherart

Die komplette Matrix wird mit all ihren Elementen gespeichert.

Vorteile	**Nachteile**
Einfache Handhabung, da man direkt auf die einzelnen Elemente der Matrix zugreifen kann	n x m Elemente müssen gespeichert werden, obwohl 70% der Einträge 0 sind -> großer Speicherverbrauch
Höhere Performance der Algorithmen, da das Decodieren entfällt	

3.2 Compressed Row Storage (CRS)/Compressed Column Storage (CCS)[1]

Eine alternative Speicherform, um dünnbesetzte Matrizen zu speichern ist die Compressed Row Storage (CRS) oder Compressed Column Storage (CCS).
Die Vorgehensweise soll anhand des CRS-Verfahrens erläutert werden:
Gegeben sei eine Matrix (Indizes starten bei 0!):

$$A = \begin{pmatrix} 0 & 12_{(0,1)} & 3_{(0,2)} & 2_{(0,3)} \\ 4_{(1,0)} & 0 & 2_{(1,2)} & 0 \\ 0 & 1_{(2,1)} & 0 & 3_{(2,3)} \end{pmatrix}$$

Es werden die Werte, die ungleich 0 sind, in einem Array **Werte** gespeichert. Dabei wird die Reihenfolge berücksichtigt. Zeilenweise von links nach rechts.

$$Werte = (12,\ 3,\ 2,\ 4,\ 2,\ 1,\ 3)$$

Zu jedem Wert im Array **Werte** wird in einem zweiten Array **Spalten-Index** der korrespondierende Spalten-Index hinterlegt. Das heißt, dass die Arrays **Werte** und **Spalten-Index** immer die gleiche Elementanzahl aufweisen.

$$Spalten - Index = (1,\ 2,\ 3,\ 0,\ 2,\ 1,\ 3)$$

In einem dritten Array **Zeilen-Pointer** wird die Anzahl der Elemente pro Zeile codiert:
Der erste Wert ist immer die 0. Die weiteren Einträge sind die kumulierten Häufigkeiten der Nicht-null-Elemente pro Zeile. Somit ist der letzte Wert immer die Anzahl aller Nicht-null-Elemente der Matrix. Die Einträge dieses Arrays zeigen die Grenzen einer Zeile an.

$$Zeilen - Pointer = (0,\ 3,\ 5,\ 7)$$

Aus benachbarten Werten kann man auf die Indizes für **Spalten-Index** und **Werte** schließen, um eine Zeile zu rekonstruieren.

[1] Als Grundlage für diesen Abschnitt diente (https://de.wikipedia.org/wiki/Compressed_Row_Storage, 2016)

Ein Beispiel: 2. Zeile rekonstruieren
Um die zweite Zeile zu rekonstruieren, benötigt man das zweite „2-er-Tupel“.

$$Zeilen - Pointer = (0, \quad \mathbf{3}, \quad \mathbf{5}, \quad 7)$$

Die **3** gibt uns nun den Start-Index und die **5** den End-Index (exklusiv) für die Arrays **Spalten-Index** und **Werte** (sowie die Anzahl der Elemente in dieser Zeile (5-3=2)):

$$Spalten - Index = (1, \quad 2, \quad 3, \quad 0, \quad 2, \quad 1, \quad 3)$$

$$Zeilen - Pointer = (0, \quad \mathbf{3}, \quad \mathbf{5}, \quad 7)$$

$$Werte = (12, \quad 3, \quad 2, \quad 4, \quad 2, \quad 1, \quad 3)$$

Nun hat man alle Informationen, die man braucht: Zeilennummer, Spaltennummer und die dazugehörigen Werte. Anhand der 0 in **Spalten-Index** und 4 in **Werte** weiß man, dass die 4 in der ersten Spalte stehen muss. Analog dazu kann man ablesen, dass die 2 in der dritten Spalte steht. Die restlichen Elemente der zweiten Zeile mit Nullen auffüllen.

$$\begin{pmatrix} & \cdots & & \\ 4_{(1,0)} & 0 & 2_{(1,2)} & 0 \\ \vdots & \vdots & \vdots & \vdots \\ & \cdots & & \end{pmatrix}$$

Die Compressed Column Storage läuft analog zum CRS-Verfahren, nur, dass als Betrachtungsbereich nicht die Zeilen sondern die Spalten gelten. Sprich: Werte werden Spaltenweise einsortiert, es gibt ein Array um die Zeilen-Indizes zu speichern und ein Spalten-Pointer-Array.

Vorteile	**Nachteile**
Weniger Speicherverbrauch	Zeilen/Spalten müssen vor Verwendung wiederhergestellt werden
Schnelle Suche von Matrixelementen	Hinzukommen von Nicht-Null-Elementen und Entfernen der Null-Elemente erfordern ständiges umordnen -> geringe Performance

4. Diskussion der Speicherformen einer n x m Matrix

4.1 Zu speichernde Elemente

4.1.1 Standard Speicherart

Bei der Standard Speicherart müssen alle Elemente abgespeichert werden:
Anzahl der zu speichernden Elemente: $n * m$
Beispiel: $10 * 11 = 110$

4.1.2 Compressed Row Storage

Worst-Case (genau 30% der Elemente sind ungleich null):
Anzahl zu speichernden Elemente für:

a. Werte-Array: $0.3 * n * m$
b. Spalten-Index-Array: $0.3 * n * m$
c. Zeilen-Pointer-Array: $n + 1$

Insgesamt: $2 * (0.3 * n * m) + n + 1 = 0.6 * n * m + n + 1$
Beispiel: 10x11 Matrix: $0.6 * 10 * 11 + 10 + 1 = 66 + 11 = 77$

4.2 Verwaltungsaufwand

4.2.1 Standard Speicherart

Bei der Standard Speicherart ist keine weitere Verwaltung nötig. Man kann über die Indizes direkt auf die einzelnen Elemente zugreifen.

4.2.2 Compressed Row Storage

Beim CRS-Verfahren ist eine weitere Verwaltung nötig, da sich durch das Hinzukommen oder Wegfallen eines Nicht-null-Elements die Speicher-Arrays ändern.
Fällt ein Wert aus der Matrix weg (wird 0), muss dieser Eintrag aus den beiden Arrays **Werte** und **Spalten-Index** entfernt werden. Daraus folgt unter Umständen eine Neuordnung des gesamten Arrays, sofern nicht das letzte Nicht-null-Element gelöscht wurde (letztes Element im Array).

Gegeben sei folgende Matrix:

$$A = \begin{pmatrix} 0 & 12_{(0,1)} & 3_{(0,2)} & 2_{(0,3)} \\ 4_{(1,0)} & 0 & 2_{(1,2)} & 0 \\ 0 & 1_{(2,1)} & 0 & 3_{(2,3)} \end{pmatrix}$$

Daraus folgen die Verwaltungs-Arrays:

$$Werte = (12,\ 3,\ 2,\ 4,\ 2,\ 1,\ 3)$$
$$Spalten - Index = (1,\ 2,\ 3,\ 0,\ 2,\ 1,\ 3)$$
$$Zeilen - Pointer = (0,\ 3,\ 5,\ 7)$$

Würde die $2_{(1,2)}$ in der zweiten Zeile, dritten Spalte wegfallen, müssen die rot markierten Elemente in **Werte** und **Spalten-Index** entfernt werden.

$$Werte = (12, \quad 3, \quad 2, \quad 4, \quad \dots \quad 1, \quad 3)$$
$$Spalten - Index = (1, \quad 2, \quad 3, \quad 0, \quad \dots \quad 1, \quad 3)$$

Gefolgt davon müssen alle nachfolgenden Elemente nach vorne gezogen werden.

$$Werte = (12, \quad 3, \quad 2, \quad 4, \quad 1, \quad 3)$$
$$Spalten - Index = (1, \quad 2, \quad 3, \quad 0, \quad 1, \quad 3)$$

In **Zeilen-Pointer** muss der End-Index der zweiten Zeile bzw. Start-Index der dritten Zeile geändert werden. In diesem Fall von 5 auf 4, da die zweite Zeile nur noch einen Wert enthält, der nicht Null ist. Auch wenn in einer Zeile nur Nullen stehen würden, hat dieses Array immer die gleiche Länge. Es wird nichts gelöscht (Es würde nur zweimal die gleiche Zahl im Array hintereinander stehen.)

Analog dazu wird mit dem Hinzufügen eines Elementes ungleich Null verfahren, nur, dass keine Elemente entfernt, sondern eingefügt werden.

Zusammenfassung

Pro Entfernung eines Wertes müssen 2 Einträge gelöscht und einer geändert, sowie gegebenenfalls die Arrays defragmentiert werden.
Pro Hinzufügen eines Wertes müssen 2 Einträge hinzugefügt und einer geändert, sowie gegebenenfalls die Arrays defragmentieren werden.

4.3 Fazit

Das CRS-Verfahren bietet Vorteile in der Menge des benötigten Speichers, hat aber Nachteile in der Verwaltung und Performance.
Die Standard Speicherart hat Vorteile in der Verwaltung aber einen Nachteil in der Speichermenge. Da man in Cobol auf dynamische Speicherallokierung verzichten muss, muss man immer vom Worst-Case ausgehen. Auch wenn es in Cobol die Möglichkeit gibt mit „DEPENDING ON“ nicht benötigten Speicher wieder freizugeben, muss ebenfalls beachtet werden, dass auch wieder Elemente hinzugefügt werden können. Dabei muss die Obergrenze immer fest gesetzt werden. Und diese Obergrenze muss den Worst-Case abdecken.
Somit kommt man immer noch auf gut 70% des Speicherbedarfs (30% Ersparnis).
Für etwa 30% Speicherplatzersparnis muss man die Verwaltungen und Performance Einbuße in Kauf nehmen. Mit dynamischer Speicherallokierung und anderen Datentypen wie doppelt verketteten Liste, die eine Umstrukturierung eines Arrays, wenn man Werte mitten drin einfügt oder löscht, effizient gestaltet ohne alle nachfolgenden Elemente zu verschieben, könnte man das CRS-Verfahren effizient umsetzen.
Deshalb haben wir uns an dieser Stelle für die Standard Speicherart entschieden.

5. Systembeschreibung

Das Programm ist nach dem EVA (Eingabe-Verarbeitung-Ausgabe) Prinzip aufgebaut.

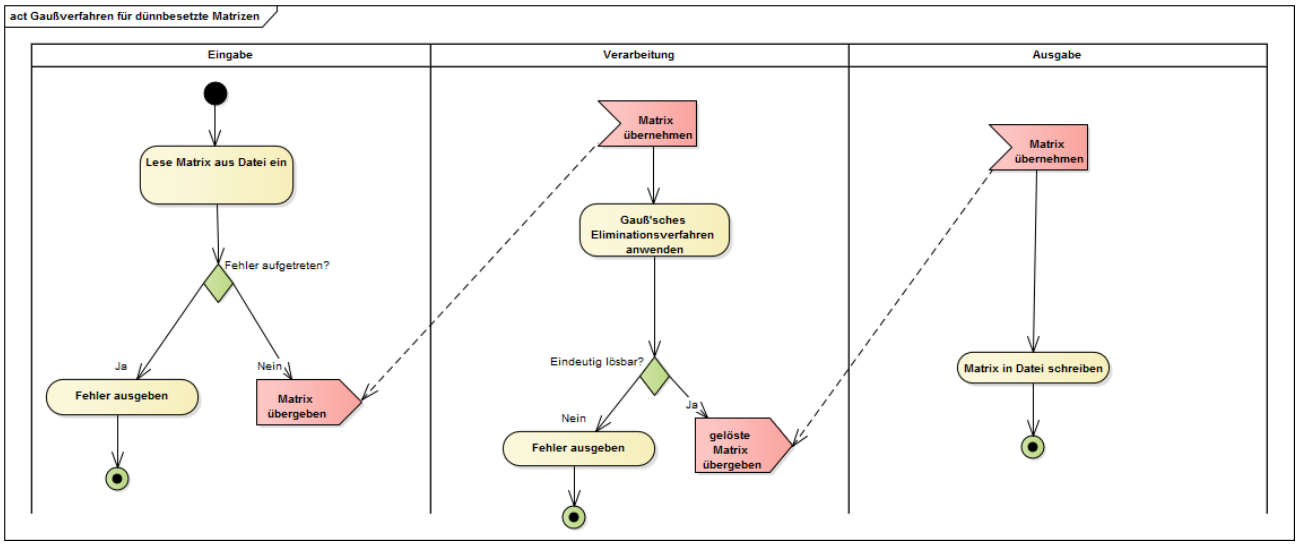


Abbildung 1:.../Aktivitätsdiagramm/AktvitätsdiagrammÜberblick.png

5.1 Eingabe

In der Eingabe wird eine Datei („eingabe.txt“), die die Dimension sowie die Matrix (Ax=b) beinhaltet, eingelesen.
Format der Datei:
Erste Zeile: maximale Spaltenanzahl (Zahlenformat: Vorzeichen|zweistellige Zahl)
Die folgenden Zeilen beinhalten die Matrix.
(Zahlenformat: Vorzeichen|dreistellige Zahl|Punkt|zwei Nachkommastellen)

Beispiel:

+05
+001.00+000.00+000.00+000.00+001.00
+000.00+001.00+000.00+000.00+001.00
+000.00+000.00+001.00+000.00+001.00
+000.00+000.00+000.00+001.00+001.00

Dabei wird die maximale Anzahl der Zeilen aus der maximalen Anzahl der Spalten berechnet. Die maximale Zeilenanzahl ist die maximale Spaltenanzahl um eins vermindert.
Dabei ist zu beachten:
Gibt man eine niedrigere Spaltenanzahl in der ersten Zeile ein, als in den folgenden Zeilen beschriebene Matrix aufweist, wird die Matrix „beschnitten“. Sei n die maximale Anzahl der Spalten, so wird auch nur die (n+1) x n gelesen.
Das Maximum beträgt eine 98x99 Matrix, da man bei Cobol eine Obergrenze braucht.
Wird beim einlesen festgestellt, dass schon mehr als 30% der Matrixelemente ungleich null sind, wird das Lesen unterbrochen und eine Fehlermeldung angezeigt.
Andernfalls wird die eingelesene Matrix weiter an die Verarbeitung geschickt.

Autor: Björn Lüpschen

5.1.1 Struktogramme

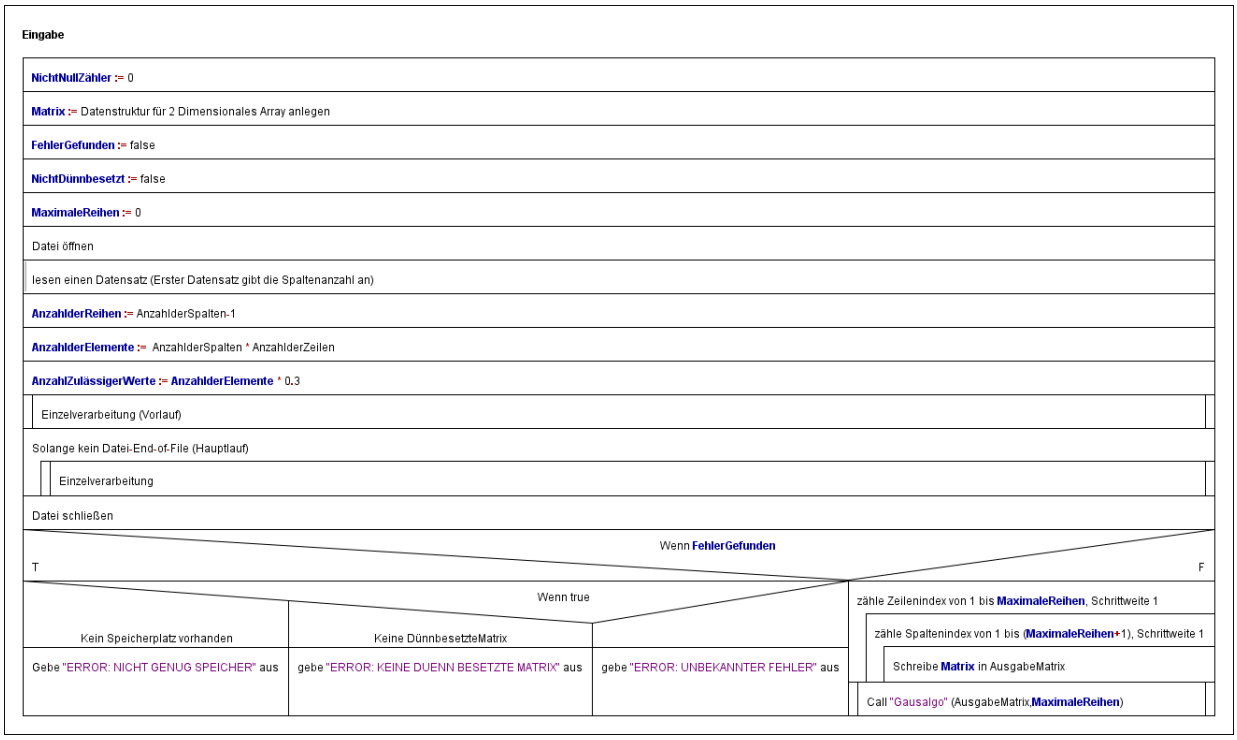


Abbildung 2: .../Struktogramme/Eingabe.png

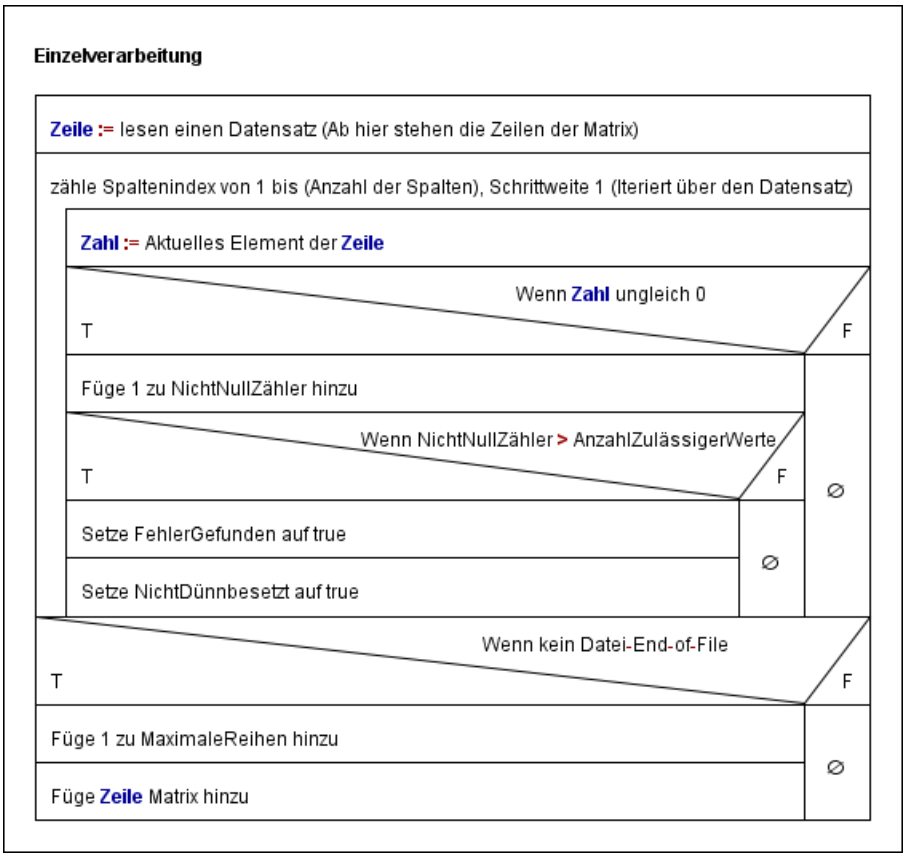


Abbildung 3:.../Struktogramme/Einzelverarbeitung.png

Autor: Philipp Kohl

5.2 Verarbeitung

In der Verarbeitung wird die Matrix aus der Eingabe entgegengenommen und mithilfe des Gauß'schen Eliminationsverfahren versucht die Matrix Ax in Diagonalform zu bringen.

5.3 Beschreibung des Gauß'schen Eliminationsalgorithmus

Es wird angestrebt in jeder Spalte nur ein Nicht-Null-Element zu haben und der Rest soll aus Nullen bestehen, abgesehen des Ergebnisvektors.
Dazu gehe man wie folgt vor:

I. Iteriere über alle Spalten bis auf die Ergebnisspalte (Im Weiteren beschreibt i den Index der Spalte, in der man sich befindet). Da in dieser nicht nur ein Wert ungleich Null sein soll.
   a. Suche in dieser Spalte den betragsmäßig größten Wert – das Pivotelement – um die Rundungsfehler zu minimieren. Da es sich hier um eine dünnbesetzte Matrix handelt, ist es sehr wahrscheinlich, dass es das einzige Nicht-null-Element in dieser Spalte ist.
      Sind nur Nullen in dieser Spalte, ist die Matrix nicht lösbar. Der Algorithmus kann abgebrochen werden.
   b. Tausche die Zeile, in der das Pivotelement steht, mit der der i-ten Zeile.
   c. Iteriere über alle Zeilen
      i. Sofern es nicht die gleiche Zeile ist, in der auch das Pivotelement steht, ziehe man die Pivotzeile von den anderen so ab, dass in der i-ten Spalte die Variable eliminiert wird.
         Wenn die Variable bereits eliminiert ist, entfällt dieser Schritt. Bei dünnbesetzten Matrizen kann das zur Performanceverbesserung führen, da eine dünnbesetzte Matrix zu 70% aus Nullen besteht.

II. Zuletzt iteriere man noch einmal über alle Zeilen um diese zu normalisieren, so dass man das Ergebnis direkt ablesen kann.

5.4 Struktogramm

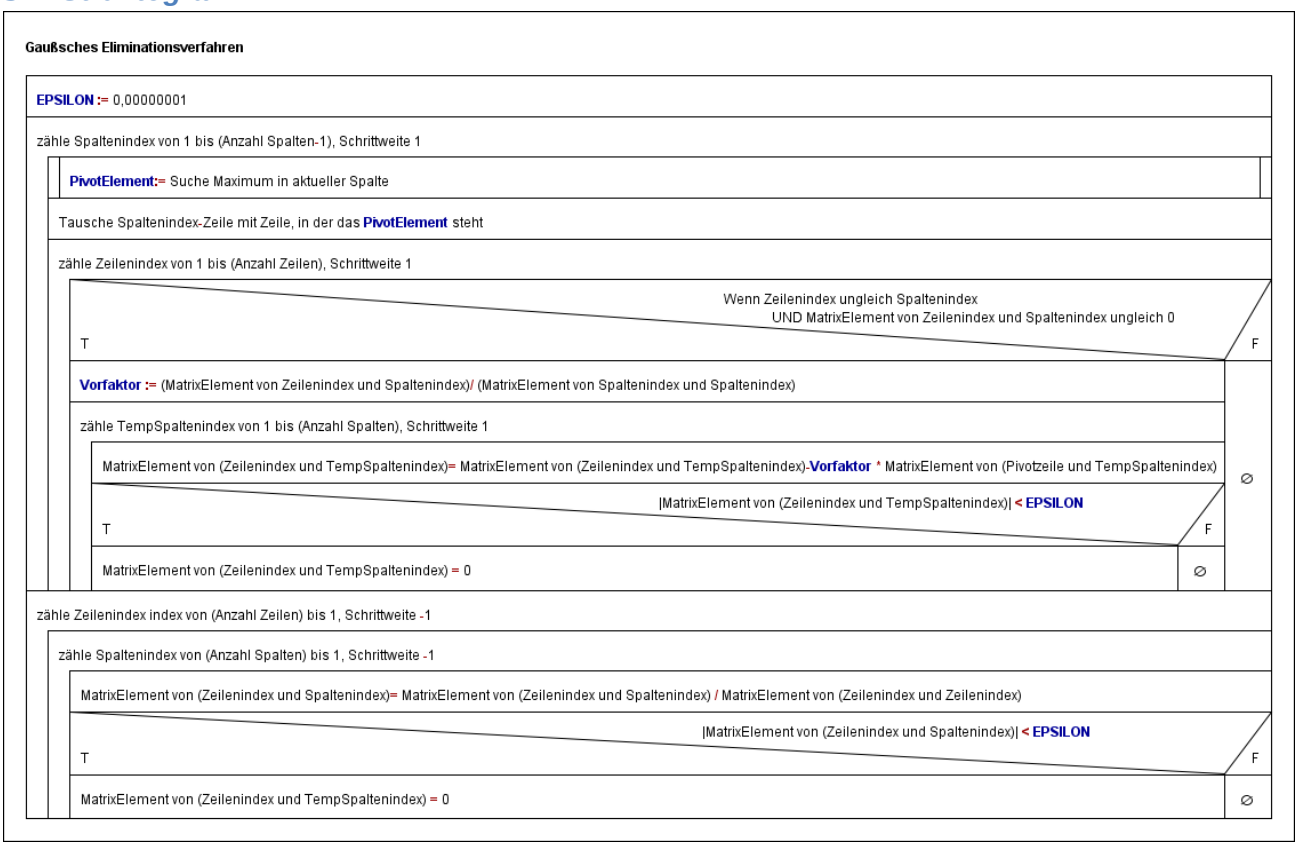


Abbildung 4:.../Struktogramme/Gaußsches_Eliminiationsverfahren.png

5.5 Beispiel

Gegeben ist eine Matrix A:
(in diesem Fall keine dünnbesetzte Matrix, da sonst nur 3 Elemente nicht Null sein dürften oder man müsste eine größere Matrix nehmen. Aus Übersichtlichkeitsgründen wird darauf verzichtet. Der Algorithmus funktioniert für „normale“ Matrizen, also auch für dünnbesetzte.)

$$A = \begin{pmatrix} -3 & -4 & 0 & -8 \\ 3 & 0 & 0 & 12 \\ 9 & 0 & -4 & 4 \end{pmatrix}$$

Man iteriert über die Spalten der Matrix A.
Erst Iteration (i=0):
In der ersten Spalte wird das betragsmäßig größte Element der Spalte gesucht – das Pivotelement: die 9 in der dritten Zeile:

$$A = \begin{pmatrix} -3 & -4 & 0 & -8 \\ 3 & 0 & 0 & 12 \\ 9 & 0 & -4 & 4 \end{pmatrix}$$

Diese Pivotzeile wird mit der ersten Zeile getauscht:

$$A = \begin{pmatrix} 9 & 0 & -4 & 4 \\ 3 & 0 & 0 & 12 \\ -3 & -4 & 0 & -8 \end{pmatrix}$$

Nun iteriert man über alle Zeilen und zieht die erste Zeile von allen anderen, nur nicht von sich selbst, so ab, dass in der ersten Spalte nur nullen stehen – die erste Variable eliminieren- bis auf die i-te Zeile:

$$A = \begin{pmatrix} 9 & 0 & -4 & 4 \\ 0 & 0 & 4/3 & 32/3 \\ 0 & -4 & -4/3 & -20/3 \end{pmatrix}$$

Ab hier fängt die Iteration wieder von vorne an.
Zweite Iteration (i=1):
Nun wird in der zweiten Spalte das Pivotelement gesucht. Die -4:

$$A = \begin{pmatrix} 9 & 0 & -4 & 4 \\ 0 & 0 & 4/3 & 32/3 \\ 0 & -4 & -4/3 & -20/3 \end{pmatrix}$$

Tausche die Pivotzeile mit der zweiten Zeile:

$$A = \begin{pmatrix} 9 & 0 & -4 & 4 \\ 0 & -4 & -4/3 & -20/3 \\ 0 & 0 & 4/3 & 32/3 \end{pmatrix}$$

Nun iteriert man wieder über alle Zeilen und versucht die zweite Variable mithilfe des Pivotelements zu eliminieren. Diese sind hier schon null. Es greift die Optimierung, so dass gar keine Rechnung stattfindet, da man schon das gewünschte Ergebnis hat.

Dritte Iteration (i=2):

In der letzten Iteration bleibt nur noch 4/3 als Pivotelement übrig und die Vertauschung der Zeilen hat keine Auswirkung, da es die selbe Zeile ist.

$$A = \begin{pmatrix} 9 & 0 & -4 & 4 \\ 0 & -4 & -4/3 & -20/3 \\ 0 & 0 & 4/3 & 32/3 \end{pmatrix}$$

Mithilfe der letzten Zeile wird die dritte Variable eliminiert:

$$A = \begin{pmatrix} 9 & 0 & 0 & 36 \\ 0 & -4 & 0 & 4 \\ 0 & 0 & 4/3 & 32/3 \end{pmatrix}$$

Zum Schluss werden alle Zeilen noch normalisiert, so dass man das Ergebnis direkt ablesen kann. Dafür wird die i-te Zeile durch den Wert, der an (i,i)-ter Stelle steht, geteilt.

$$A = \begin{pmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 8 \end{pmatrix}$$

Ob die Matrix lösbar war, wird anhand der letzten Zeile entschieden. Wenn in der letzten Zeile mehr oder weniger als ein Wert steht (den Ergebniswert außen vorgelassen), war die Matrix nicht lösbar bzw. nicht eindeutig lösbar.

5.6 Ausgabe

In die Ausgabe gelangt man nur, solange vorher kein Fehler aufgetreten ist. Somit schreibt die Ausgabe einzig und alleine die gelöste Matrix in eine Datei („ausgabe.txt“).
Beispiel einer Ausgabe:

```
1.000   0.000   0.000   0.000   1.000
0.000   1.000   0.000   0.000   1.000
0.000   0.000   1.000   0.000   1.000
0.000   0.000   0.000   1.000   1.000
```

An dieser Stelle wurde auf ein Struktogramm verzichtet, da die Ausgabe zu trivial ist.

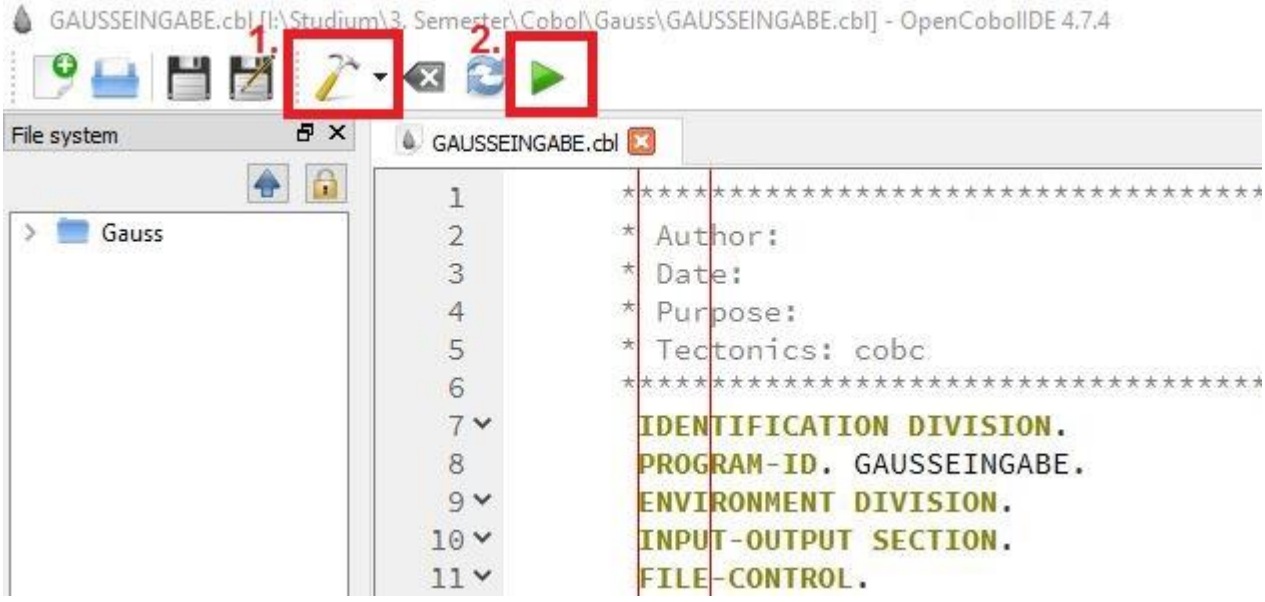
6. Benutzeranleitung

Das Programm wird allein über eine Datei gespeist und hat intern auch keine Auswahlmöglichkeiten.
Nur Fehlermeldungen werden auf dem Bildschirm ausgegeben.

1) Somit muss die Eingabedatei das richtige Format aufweisen. (Siehe Eingabe 5.1)
2) Programm ausführen: Entweder über das ausgelieferte Executable oder über die Entwicklungsumgebung (siehe Beschreibung der Entwicklungsumgebung 7)
3) Wenn es Fehler gibt, werden diese in der Konsole ausgegeben.
4) Wenn es keinen Fehler gab, steht das Ergebnis in der Datei „ausgabe.txt“

7. Beschreibung der Entwicklungsumgebung

Es wurde die bekannte, in der Universität vorgestellte, Entwicklungsumgebung OpenCobolIDE genutzt.



Um das Projekt zu bauen oder auszuführen, sollte man immer die GAUSSEINGABE.cbl öffnen.

- **Zum Bauen den Hammer klicken (1.)**
- **Zum Ausführen den Play-Button klicken (2.)**

Autor: Björn Lüpschen

8. Testfallbeschreibung

Die Testfallerstellung strebt eine Anweisungs- und Pfadabdeckung an, um jedes mögliche Szenario zu testen. Und nicht jede mögliche Matrix zu testen, da dieses Vorhaben ohnehin unmöglich ist.
Vorab: **Die Vorbedingung für alle Tests ist die, dass die Eingabedatei ein valides Format hat.**

8.1 Anweisung-/Pfadvollständig

Anweisungsvollständig:

Um jede Anweisung einmal auszuführen werden drei Testfälle benötigt (siehe Aktivitätsdiagramm), da das Programm nahezu linear ist und nur zwei Verzweigungen beinhaltet.
Im folgendem werden die Testfälle anhand der Anweisungen des Aktivitätsdiagramms beschrieben.

- Testfall : Best-Case (..src/bin/test/pflichttest.txt)
  (1) „Lese Matrix aus Datei ein“
  (2) „Matrix übergeben“
  (3) „Matrix übernehmen“
  (4) „Gauß’sches Eliminationsverfahren anwenden“
  (5) „gelöste Matrix übergeben“
  (6) „gelöste Matrix übernehmen“
  (7) „Matrix in Datei schreiben“

- Testfall: Nicht-eindeutig-loesbar (siehe Grenzwertanalyse)
  (1) „Lese Matrix aus Datei ein“
  (2) „Matrix übergeben“
  (3) „Matrix übernehmen“
  (4) „Gauß’sches Eliminationsverfahren anwenden“
  (5) „Fehler ausgeben“ (Verarbeitung)

- Test: Nicht-duennbesetzt (siehe Grenzwertanalyse)
  (1) „Lese Matrix aus Datei ein“
  (2) „Fehler ausgeben“ (Eingabe)

Diese Testfälle sind nicht nur Anweisungsvollständig (C0),
sondern direkt auch Pfadvollständig (C1).
Es gibt keinen weiteren Pfad das Programm zu durchlaufen.

8.2 Grenzwertanalyse

Die Eingabeparameter des Programms beschränken sich auf die Dimension der Matrix, die Matrix selbst sowie die Ergebnismatrix.
Eine Grenzwertanalyse bietet sich vor allem für die Entscheidung an, ob eine Matrix dünnbesetzt ist oder nicht.

Beschreibung	Status	Repräsentant(../src/bin/..)	Ergebnis(..src/bin/..)
> >30% Nicht-Null-Elemente	Ungültig	Test/NichtDünnbesetzt.txt	Konsolenausgabe: „ERROR: KEINE DUENN BESETZTE MATRIX“
30% Nicht-Null-Elemente + 1	Ungültig	Test/30,04%nichtnull.txt	Konsolenausgabe: „ERROR: KEINE DUENN BESETZTE MATRIX“
30% Nicht-Null-Elemente	Gültig	Test/30%nichtnull.txt	Konsolenausgabe „nicht (eindeutig) loesbar!“
30% Nicht-Null-Elemente – 1	Gültig	Test/29,96%nichtnull.txt	Konsolenausgabe „nicht (eindeutig) loesbar!“
<<30% Nicht-Null-Elemente	Gültig	Test/pflichttest.txt	Test/ Ergebnis/pflichttestErg.txt

8.3 Sonstige Tests

Der in der Aufgabe geforderte Test ist unter „…Test/pflichttest.txt“ zu finden. Dieser wurde nur insofern abgeändert, dass eine Ergebnisspalte hinzugefügt wurde, da der Algorithmus eine Matrix der Form Ax=b erwartet. Das Ergebnis steht wie gewohnt im Verzeichnis „…Test/Ergebnis/pflichttest.txt“

9. Code Listing

Siehe PDF „Code Listing.pdf“

10. Ressourcen

- Struktogramme (im Unterordner „Struktogramme“):
  - Eingabe
  - Einzelverarbeitung
  - Gaußsches_Eliminationsverfahren
- Source-Code (im Unterordner „src“):
  - GAUSSEINGABE.cbl
  - GAUSSALGO.cbl
  - GAUSSAUSGABE.cbl
  - MATRIX.cpy
- Aktivitätsdiagramm (im Unterordner „Aktivitätsdiagramm“):
  - AktivitätsdiagrammÜberblick.bmp
- Code-Listing (im Hauptordner)
  - Code Listing.pdf

11. Hilfsmittel

11.1 Literaturverzeichnis

- *https://de.wikipedia.org/wiki/Compressed_Row_Storage*. (12. Oktober 2016).

11.2 Programme

- STRUCTORIZER: http://structorizer.fisch.lu/
- OpenCobolIDE: http://opencobolide.readthedocs.io/en/latest/
- Enterprise Architect: http://www.sparxsystems.com/products/ea/