|  |
| --- |
| FH-Aachen |
| Gauß-Verfahren für dünn besetzte Matrizen |
| COBOL |

|  |
| --- |
| Björn Lüpschen & Philipp Kohl  31.01.2017 |

Inhaltsverzeichnis

[1. Aufgabenstellung 3](#_Toc473213094)

[2. Aufgabenanalyse 3](#_Toc473213095)

[2.1 System 3](#_Toc473213096)

[2.2 Programmdokumentation 3](#_Toc473213097)

[2.3 Tests 4](#_Toc473213098)

[3. Möglichkeiten zur Abspeicherung dünn besetzter Matrizen (n x m) 4](#_Toc473213099)

[3.1 Standard Speicherart 4](#_Toc473213100)

[3.2 Compressed Row Storage (CRS)/Compressed Column Storage (CCS) 4](#_Toc473213101)

[4. Diskussion der Speicherformen einer n x m Matrix 5](#_Toc473213102)

[4.1 Zu speichernde Elemente 5](#_Toc473213103)

[4.1.1 Standard Speicherart 5](#_Toc473213104)

[4.1.2 Compressed Row Storage 5](#_Toc473213105)

[4.2 Verwaltungsaufwand 6](#_Toc473213106)

[4.2.1 Standard Speicherart 6](#_Toc473213107)

[4.2.2 Compressed Row Storage 6](#_Toc473213108)

[4.2.3 Zusammenfassung 7](#_Toc473213109)

[4.3 Fazit 7](#_Toc473213110)

[5. Systembeschreibung 7](#_Toc473213111)

[5.1 Eingabe 7](#_Toc473213112)

[5.2 Verarbeitung 7](#_Toc473213113)

[5.3 Textuelle Beschreibung des Gauß’schen Eliminationsalgorithmus mit Beispiel 8](#_Toc473213114)

[5.4 Struktogramm 9](#_Toc473213115)

[5.5 Ausgabe 10](#_Toc473213116)

[6. Testfallbeschreibung 10](#_Toc473213117)

[6.1 Anweisung-/Pfadvollständig 10](#_Toc473213118)

[6.2 Grenzwertanalyse 11](#_Toc473213119)

[6.3 Sonstige Tests 11](#_Toc473213120)

[7. Anhang 11](#_Toc473213121)

# Aufgabenstellung

# Aufgabenanalyse

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

* 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.
  1. 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.
  1. Tests
  + Das Programm muss anhand von verschiedenen Testfällen getestet werden.
  + Die Auswahl der Testfälle muss begründet werden.
  + Die Auswahl der Testfälle muss diskutiert werden.

# Möglichkeiten zur Abspeicherung dünn besetzter Matrizen (n x m)

## 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 |  |

## Compressed Row Storage (CRS)/Compressed Column Storage (CCS)

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

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.   
  
  
  
Aus benachbarten Werten kann man auf die Indizes für **Spalten-Index** und **Werte** schließen, um eine Zeile zu rekonstruieren.  
Ein Beispiel: 2. Zeile rekonstruieren  
Um die zweite Zeile zu rekonstruieren, benötigt man das zweite „2-er-Tupel“.  
  
  
  
Die **3** gibt uns nun den Start-Index und die **5** den End-Index für die Arrays **Spalten-Index** und **Werte** (sowie die Anzahl der Elemente in dieser Zeile (5-3=2)):

Nun hat man alle Informationen, die man braucht: Zeilennummer, Spaltennummer und die dazugehörigen Werte. Anhand der 0 in **Spalten-Index** und4 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.

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 |

# Diskussion der Speicherformen einer n x m Matrix

## Zu speichernde Elemente

* + 1. Standard Speicherart  
       Bei der Standard Speicherart müssen alle Elemente abgespeichert werden:

Anzahl der zu speichernden Elemente:   
Beispiel:

* + 1. Compressed Row Storage  
       Worst-Case (genau 30% der Elemente sind ungleich null):   
       Anzahl zu speichernden Elemente für:
  1. Werte-Array:
  2. Spalten-Index-Array:
  3. Zeilen-Pointer-Array:

Insgesamt:   
Beispiel: 10x10 Matrix:

## Verwaltungsaufwand

### Standard Speicherart

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

### 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.  
  
Gegeben sei folgende Matrix:

Daraus folgen die Verwaltungs-Arrays:

Würde die in der zweiten Zeile, dritten Spalte wegfallen, müssen die rot markierten Elemente in **Werte** und **Spalten-Index** entfernt werden.

Gefolgt davon müssen alle nachfolgenden Elemente nach vorne gezogen werden.

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.

## 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.

# Systembeschreibung

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

## Eingabe

In der Eingabe wird eine Datei, die die Dimension sowie die Matrix (Ax=b) beinhaltet, eingelesen.  
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.

## Verarbeitung

In der Verarbeitung wird die Matrix aus der Eingabe entgegengenommen und mithilfe des Gauß’schen Eliminationsverfahren versucht in Diagonalform zu bringen (wenn man b [Ergebnisvektor] ausser Acht lässt).

### Textuelle Beschreibung des Gauß’schen Eliminationsalgorithmus mit Beispiel

Es wird angestrebt in jeder Spalte nur ein Nicht-null-Element zu haben und der Rest soll aus nullen bestehen, abgesehen des Ergebnisvektors.   
Gegeben ist eine Matrix A (in diesem Fall keine dünnbesetzte Matrix, da sonst nur 3 Elemente nicht null sein dürften. Algorithmus funktioniert für „normale“ Matrizen, also auch für dünnbesetzte.):

Dazu iteriert man über die Spalten der Matrix A.  
In der i-ten Spalte wird das betragsmäßig größte Element der Spalte gesucht – das Pivotelement.  
In der ersten Iteration wäre das die 9 in der dritten Zeile.

Diese Pivotzeile wird mit der i-ten Zeile getauscht. In diesem Fall mit der ersten (i=0).

Nun iteriert man über alle Zeilen und zieht die i-te Spalte von allen anderen, nur nicht von sich selbst, so ab, dass in der ersten Spalte nur nullen stehen, bis auf die i-te Zeile.   
TODO: Faktor beschreiben? Ja? Nein?

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

Tausche die Pivotzeile mit der i-ten Zeile.

Nun iteriert man wieder über alle Zeilen und versucht die Werte in der 2.Spalte mithilfe des Pivotelements zu eliminieren. Diese sind hier schon null, deshalb hat der Algorithmus keinen Effekt, da man 0\*2.Zeile von den anderen Zeilen abzieht.

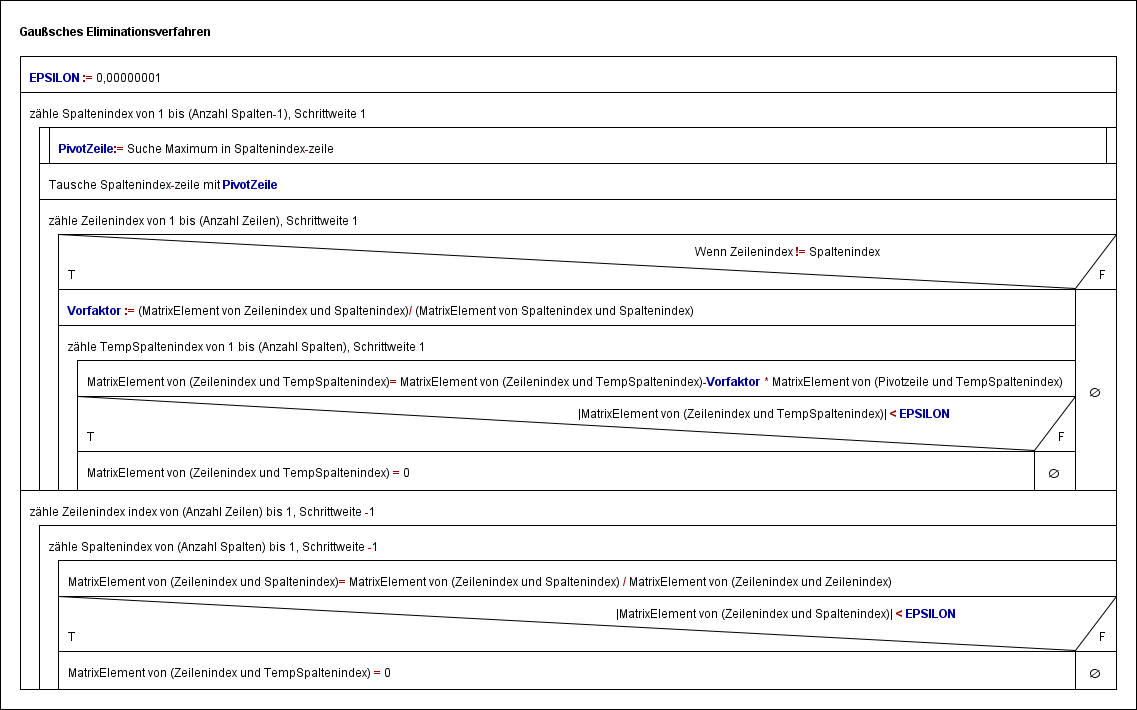
DARF MAN VON VORNE ITERIEREN? DANN KÖNNTE DOCH AUCH IN DER ERSTE ZEILE WIEDER DAS PIVOT ELEMENT STEHEN.  
  
In der letzten Iteration bleibt nur noch 4/3 als Pivotelement übrig.

Mithilfe der letzten Zeile werden nun die anderen Zeilen so verrechnet, dass die Spalte aus nullen besteht, bis auf das Pivotelement.

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.

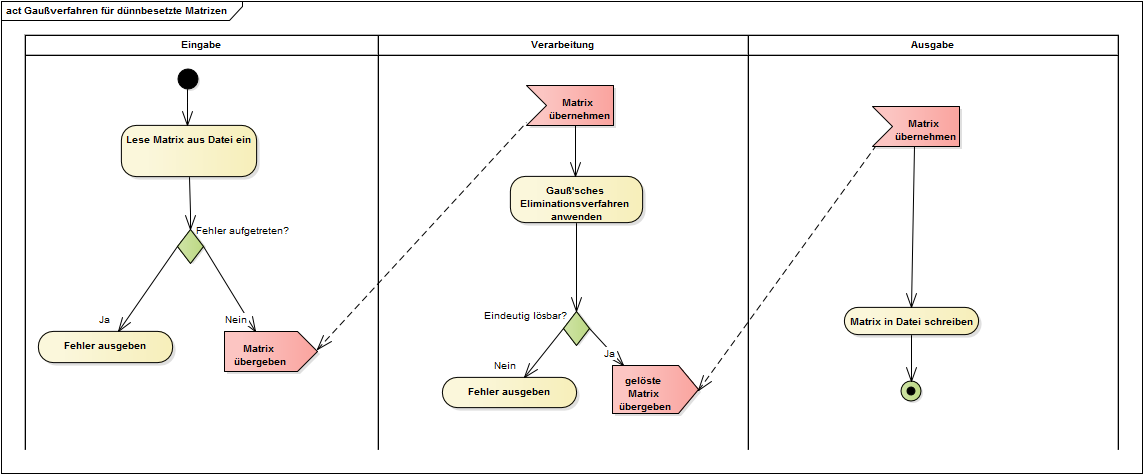
Ob die Matrix lösbar war, wird anhand der letzten Zeile entschieden. Wenn in der letzten Zeile mehr als ein Wert steht (den Ergebniswert außen vorgelassen), war die Matrix nicht lösbar.

### Struktogramm



## 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.



# 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.**

## 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
  1. „Lese Matrix aus Datei ein“
  2. „Matrix übergeben“
  3. „Matrix übernehmen“
  4. „Gauß’sches Eliminationsverfahren anwedenen“
  5. „gelöste Matrix übergeben“
  6. „gelöste Matrix übernehmen“
  7. „Matrix in Datei schreiben“
* Testfall: Nicht-eindeutig-loesbar
  1. „Lese Matrix aus Datei ein“
  2. „Matrix übergeben“
  3. „Matrix übernehmen“
  4. „Gauß’sches Eliminationsverfahren anwenden“
  5. „Fehler ausgeben“ (Verarbeitung)
* Test: Nicht-duennbesetzt
  1. „Lese Matrix aus Datei ein“
  2. „Fehler ausgeben“ (Eingabe)

Diese Testfälle sind nicht nur anweisungsüberdeckend, sondern direkt auch pfadüberdeckend.  
Es gibt keinen weiteren Pfad das Programm zu durchlaufen.

## 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 (Dateiname)** |
| > >30% Nicht-Null-Elemente | Ungültig |  |
| 30% Nicht-Null-Elemente + 1 | Ungültig |  |
| 30% Nicht-Null-Elemente | Gültig |  |
| 30% Nicht-Null-Elemente – 1 | Gültig |  |
| <<30% Nicht-Null-Elemente | Gültig |  |

## Sonstige Tests

Epsilon testen?

# Anhang

Bilder einfügen? Listing vom Quellcode?