Friedrich-Schiller-Universität Jena

seit 1558

WS 2007 / 2008

Hausarbeit - Computational Physics

Anwendung des Gauß-Jordan-Verfahrens auf einfache elektrische Schaltungen

Erstellt von: Christian Vetter (89114)
Wöllnitzer Straße 34
07749 Jena
Christian.Vetter@Uni-Jena.de
Tel. 0173 / 40 52 692

Betreuer: Prof. Thomas Pertsch
Thomas.Pertsch@Uni-Jena.de

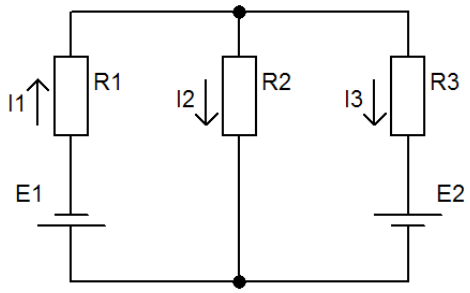
Abgabedatum: 29.02.2008

Inhaltsverzeichnis

1 Aufgabenstellung

1.1 Eigentliche Aufgabenstellung

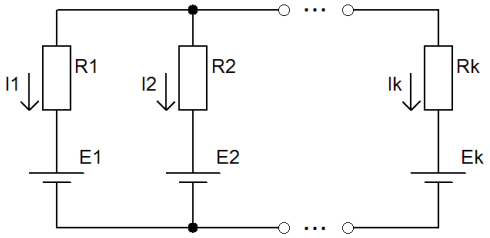
Berechnen Sie die drei Ströme I_1, I_2 und I_3 der dargestellten Schaltung.
Stellen Sie dafür die Strom - Spannungszusammenhänge in Matrixform als lineares Gleichungssystem auf und lösen Sie dieses mit dem Gauß-Jordan-Verfahren in eigener Implementierung mit vollständiger Pivotisierung.
($R_1 = 5\Omega$, $R_2 = 10\Omega$, $R_3 = 15\Omega$, $E_1 = E_2 = 1.5V$)



[1]

1.2 Erweiterte Zielsetzung

Um den Praxisbezug zu steigern und eine gewissen Variabilität zu erlauben möchte ich die Aufgabenstellung ein wenig erweitern.
Ich werde versuchen ein Programm zu entwickeln welches alle I_k der dargestellten Schaltung berechnet. Sowohl Spannungs- als auch Widerstandswerte sollen frei vom Benutzer einzugeben sein.
Ein solches Programm könnte genutzt werden um die Belastung parallel geschalteter realer Spannungsquellen unter einer gewissen Außenlast zu kalkulieren. Dabei entsprechen die Widerstände R_k den Innenwiderständen der Spannungsquellen.



[1] Darstellungen von Schaltungen, auch im Weiteren, erstellt mit ABACOM sPlan 5.0

2 Einleitung

Für die hier vorliegenden Belegarbeit habe ich mir die Zielstellung gesetzt einen Überblick über die grundlegendsten Berechnungen bei elektrischen Schaltkreisen zu geben, diese auf ein mit dem Computer ausführbares Schema zurückzuführen (kirchhoffsche Regeln) und schließlich ein Programm zu entwickeln welches dieses Schema abarbeitet.
Dazu wird die Problematik idealer Bauelemente angesprochen und einiges zu den Kirchhoffschen Regeln ausgeführt. Anschließend werden die Berechnungen der gegebenen Aufgabenstellung auf zwei Weisen demonstrieren und eine entsprechend zu genanntem Schema entwickeln.
Im zweiten, größeren Teil dieser Arbeit soll es um das Lösen von Gleichungssystemen insbesondere mithilfe des Gauß-Jordan-Algorithmus gehen. Hierzu werde ich ein Programm (in C++) entwickeln und die entsprechenden Abläufe anhand des Quelltextes kommentieren.
Der dritte und letzte Teil der Arbeit befasst sich mit der abschließenden Zusammenführung des elektronischen und des mathematischen Problems und enthält den Quelltext des vollständigen Programms, welches sich ebenfalls (für Windows compiliert) auf CD im Anhang befindet.

Christian Vetter

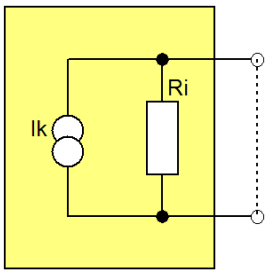
3 Grundlagen zur Berechnung linearer Netzwerke

3.1 Ideale Bauelemente

Zunächst wollen wir uns mit einfachen Bauelementen auseinandersetzen. Wichtig für die hier gestellte Aufgabe sind, die Spannungsquelle als Lieferant elektrischer Energie und der ohmschen Widerstand als deren Verbraucher. Weitere einfache Bauelemente wie Spule und Kondensator werden hier nicht näher erläutert.

3.1.1 Die ideale Stromquelle

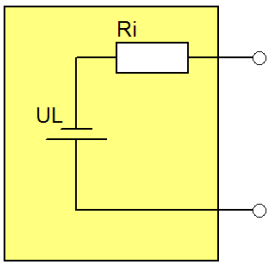
Als ideale Stromquelle wird eine Quelle elektrischer Energie bezeichnet, die in der Lage ist, unabhängig von der Außenlast einen konstanten Strom zu liefern. Eine solche ideale Quelle besitzt den Innenwiderstand $R_i = \infty$. Demnach sind für Schaltbilder alle in Reihe zu einer idealen Stromquelle befindlichen Widerstände vernachlässigbar.
Für jede lineare Schaltung lässt sich das sogenannte Kurzschlussstromersatzschaltbild zeichnen. Hierbei bezeichnet R_i den Innenwiderstand der gesamten Schaltung.



I_K ist der Kurzschlussstrom der fließt, wenn keine weitere Außenlast an die Schaltung angeschlossen wird. (ideales Ampermeter mit $R_i = 0$)

3.1.2 Die ideale Spannungsquelle

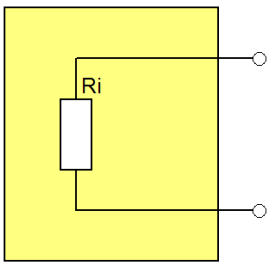
Die ideale Spannungsquelle ist im Vergleich zur idealen Stromquelle dazu in der Lage unabhängig von der Außenlast eine konstante Spannung zu generieren. Eine solche Quelle besitzt den Innenwiderstand $R_i = 0$. Demnach sind für Schaltbilder alle parallel zu einer idealen Spannungsquelle befindlichen Widerstände vernachlässigbar.
Für jede lineare Schaltung lässt sich das sogenannte Leerlaufspannungsersatzschaltbild zeichnen. Hierbei bezeichnet R_i den Innenwiderstand der gesamten Schaltung.



U_L ist die Leerlaufspannung die anliegt, wenn eine unendlich hohe Außenlast an die Schaltung angeschlossen wird. (ideales Voltmeter mit $R_i = \infty$)

3.1.3 Der ideale ohmsche Widerstand

Zunächst soll ein idealer ohmscher Widerstand elektrische Leistung umsetzen. Er soll im idealen Fall keinerlei kapazitive oder induktive Eigenschaften besitzen und somit vollkommen frequenzunabhängig sein.
Jeder Schaltung lässt sich ein idealer ohmscher Widerstand in Form des Innenwiderstandes zuordnen. Dazu werden alle Quellen in der Schaltung durch ihre idealen Innenwiderstände ersetzt und der entsprechende Widerstand zwischen den Klemmen berechnet.
Es ergibt sich das Innenwiderstandsersatzschaltbild für die Schaltung.



Abschließend gilt folgender Zusammenhang:
$\boxed{U_L = I_K \cdot R_i}$

3.2 Die Kirchhoffschen Regeln

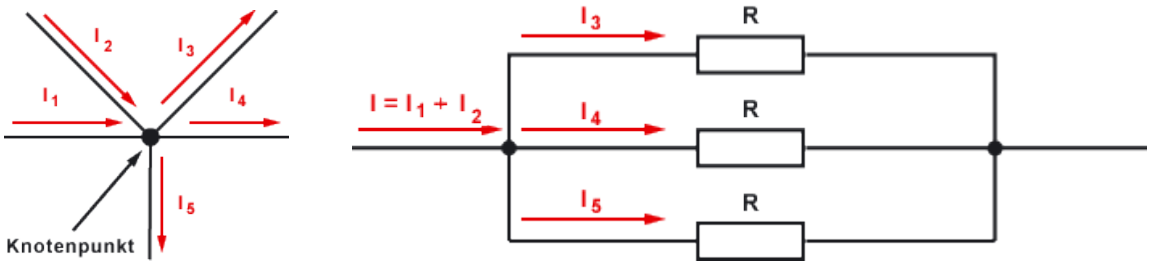
Die beiden nach Gustav Robert Kirchhoff benannten Regeln wurden 1845 von ihm entdeckt und stellen die wohl grundlegendsten Zusammenhänge zur Berechnung elektrischer Netzwerke dar. Der Vorteil der kirchhoffschen Regeln liegt darin, dass sie unter der Annahme idealer Leiter usw. für alle elektrischen Schaltungen und alle Bauelemente gültig sind. Somit stellen sie eine ideale Grundlage zur computerge-

stützten Simulation von Schaltungen dar.
Hat man alle, sich aus den Regeln ergebenden Gleichungen aufgestellt, so lassen sich die einzelnen, gesuchten Größen als lineares Gleichungssystem berechnen.

3.2.1 Knotenregel - 1. Kirchhoffsches Gesetz

Die Knotenregel beruht auf dem physikalischen Prinzip der Ladungserhaltung und besagt, dass die Summe der zufließenden Ströme zu einem elektrischen Knotenpunkt gleich der Summe der abfließenden Ströme ist.

$$\sum_{k=1}^{n} I_K = 0$$

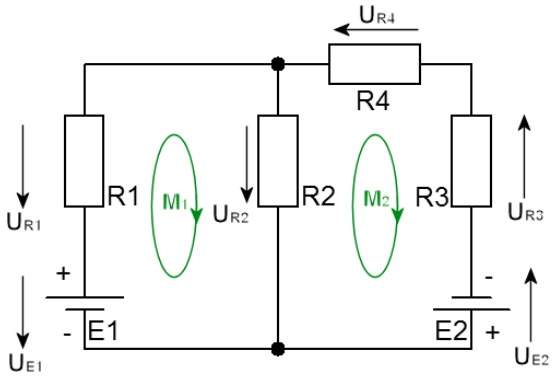


Für den hier betrachteten Fall bedeutet dies: $I_1 + I_2 - I_3 - I_4 - I_5 = 0$

3.2.2 Maschenregel - 2. Kirchhoffsches Gesetz

Die Maschenregel beruht auf dem physikalischen Prinzip der Energieerhaltung und besagt, dass die Summe aller Teilspannungen eines Umlaufs (einer Masche) verschwindet (Null wird).

$$\sum_{k=1}^{n} U_K = 0$$



Bei der Maschenregel ist besonders darauf zu achten, dass alle Maschen in der zu berechnenden Schaltung den gleichen Umlaufsinn haben!
Für den hier betrachteten Fall bedeutet dies:
M1: $-U_{E1} - U_{R1} + U_{R2} = 0$
M2: $-U_{R2} - U_{R4} - U_{R3} - U_{E2} = 0$

3.2.3 Herleitung von Strom- und Spannungsteilerregel

Aus den kirchhoffschen Regeln lassen sich auf einfache Weise Formeln zur Berechnung von Strömen und Spannungen in einfachen Netzwerken herleiten welche das Lösen eines linearen Gleichungssystems umgehen.

Spannungsteilerregel:
Für eine Reihenschaltung zweier beliebiger Widerstände gilt laut der kirchhoffschen Regeln: $U_1 + U_2 = U_0$ und $I_1 = I_2 = I_0$.
Aus $R_{ges} = R_1 + R_2$ und dem ohmschen Gesetz folgt $I_0 = \frac{U_0}{R_{ges}} = \frac{U_0}{R_1+R_2}$.
Da $R_1 = \frac{U_1}{I_1} = \frac{U_1}{I_0} = \frac{U_1}{\frac{U_0}{R_1+R_2}}$ ist, folgt weiter:

$$\boxed{U_1 = U_0 \cdot \frac{R_1}{R_1 + R_2}}$$

Analog lässt sich feststellen, dass $U_2 = U_0 \cdot \frac{R_2}{R_1+R_2}$ ist. Zusammen ergibt sich
$\frac{U_1}{U_2} = \frac{\cancel{U_0} \cdot \frac{R_1}{R_1+R_2}}{\cancel{U_0} \cdot \frac{R_2}{R_1+R_2}} = \frac{R_1}{\cancel{R_1+R_2}} \cdot \frac{\cancel{R_1+R_2}}{R_2}$

$$\boxed{\frac{U_1}{U_2} = \frac{R_1}{R_2}}$$

Stromteilerregel:
Für eine Parallelschaltung zweier beliebiger Widerstände gilt laut der kirchhoffschen Regeln: $I_1 + I_2 = I_0$ und $U_1 = U_2 = U_0$.
Aus $R_{ges} = \frac{R_1 \cdot R_2}{R_1+R_2}$ und dem ohmschen Gesetz folgt $I_0 = \frac{U_0}{R_{ges}} = U_0 \cdot \frac{R_1+R_2}{R_1 \cdot R_2}$.
Da $U_0 = U_1 = R_1 \cdot I_1 = I_0 \cdot \frac{R_1 \cdot R_2}{R_1+R_2}$ ist, folgt weiter:

$$\boxed{I_1 = I_0 \cdot \frac{R_2}{R_1 + R_2}}$$

Analog lässt sich feststellen, dass $I_2 = I_0 \cdot \frac{R_1}{R_1+R_2}$ ist. Zusammen ergibt sich
$\frac{I_1}{I_2} = \frac{\cancel{I_0} \cdot \frac{R_2}{R_1+R_2}}{\cancel{I_0} \cdot \frac{R_1}{R_1+R_2}} = \frac{R_2}{\cancel{R_1+R_2}} \cdot \frac{\cancel{R_1+R_2}}{R_1}$

$$\boxed{\frac{I_1}{I_2} = \frac{R_2}{R_1}}$$

3.3 Berechnung einfacher Netzwerke

Im folgenden soll es um die Lösung der eigentlichen Aufgabenstellung anhand zweier üblicher Verfahren gehen. Zudem soll das zweite Verfahren schematisiert werden um dieses rechnergestützt anwenden zu können.

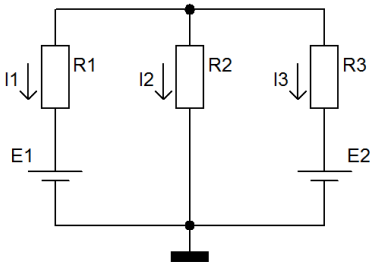
3.3.1 Superpositionsverfahren nach Helmholtz

Beim Überlagerungsverfahren nach Helmholtz handelt es sich um ein sehr praktikables Verfahren zur handschriftlichen Lösung von Schaltungsproblemen mit mehreren Quellen.

Vorgehen:

1. Bis auf eine Quelle werden alle Anderen entfernt und durch ihre idealen Innenwiderstände ersetzt. (Spannungsquelle entspricht Kurzschluss / Stromquelle entspricht Auftrennung)
2. Die gesuchten Ströme mit der verbliebenen Quelle berechnen.
3. Wiederholung für alle weiteren Quellen.
4. Abschließend alle Ströme vorzeichenrichtig addieren.

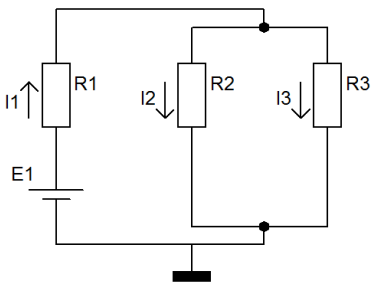
Am vorliegenden Beispiel:
Zunächst lege man sich ein Bezugspotential und Stromrichtungen fest.



$R_1 = 5\Omega$
$R_2 = 10\Omega$
$R_3 = 15\Omega$
$E_1 = -1,5V$
$E_2 = 1,5V$

---

①



Betrachtet man sich die Schaltung ohne Quelle E2, so erkennt man eine Parallelschaltung von R_2 und R_3 in Reihe mit R_1. Es lassen sich folgende Berechnungen anstellen:

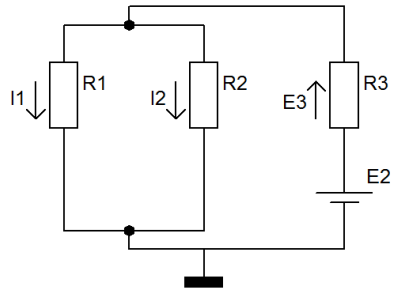
$R_2 \| R_3 = \frac{R_2 \cdot R_3}{R_2 + R_3} = 6\Omega$ $\quad I_1 = \frac{U_{R1}}{R_1} = -0,1364A$

$U_{R1} = U_{E1} \cdot \frac{R_1}{R_1 + (R_2 \| R_3)} = -0,682V$ $\quad I_2 = \frac{U_{R2}}{R_2} = \frac{U_{R_2 \| R_3}}{R_2} = -0,0818A$

$U_{R_2 \| R_3} = U_{E1} - U_{R1} = -0,818V$ $\quad I_3 = \frac{U_{R3}}{R_3} = \frac{U_{R_2 \| R_3}}{R_3} = -0,05453A$

---

②



Betrachtet man sich die Schaltung ohne Quelle E1, so erkennt man eine Parallelschaltung von R_1 und R_2 in Reihe mit R_3. Es lassen sich folgende Berechnungen anstellen:

$R_1 \| R_2 = \frac{R_1 \cdot R_2}{R_1 + R_2} = \frac{10}{3}\Omega$ $\quad I_1 = \frac{U_{R1}}{R_1} = \frac{U_{R_1 \| R_2}}{R_1} = 0,055A$

$U_{R3} = U_{E2} \cdot \frac{R_3}{R_3 + (R_1 \| R_2)} = 1,227V$ $\quad I_2 = \frac{U_{R2}}{R_2} = \frac{U_{R_1 \| R_2}}{R_2} = 0,027A$

$U_{R_1 \| R_2} = U_{E2} - U_{R3} = 0,273V$ $\quad I_3 = \frac{U_{R3}}{R_3} = 0,082A$

---

① + ②

Nun muss auf die richtige Addition geachtet werden. Wie in den Abbildungen zu sehen zeigen einzelne Ströme in ① und ② nicht in die gleiche Richtung wie in der Ausgangsdarstellung. Dies muss entsprechend kompensiert werden.
Es ergibt sich:

$I_1 = -①+②= 191mA$

$I_2 =①+②= -55mA$

$I_3 =①-②= -137mA$

3.3.2 Netzwerkanalyse mit Maschen- und Knotenregeln

Auch mit den Maschen- und Knotenregeln lässt sich das Problem lösen. Hierzu können verschiedene Verfahren verwendet werden. Üblich sind:

- Zweigstromanalyse
- Maschenstromanalyse
- Knotenpotentialverfahren

Im Folgenden soll es uns lediglich um das Maschenstromverfahren gehen. Die Anderen Beiden seien nur der Vollständigkeit halber erwähnt. Auch sie laufen nach einem ähnlichen Prinzip ab und führen zu einem linearen Gleichungssystem.

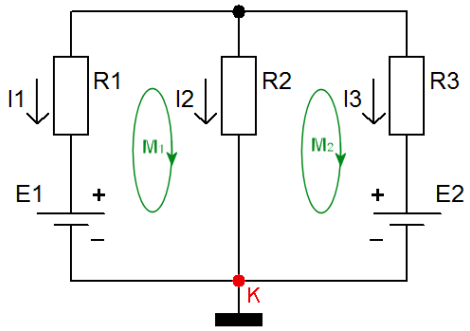
Vorgehen:

1. Einführen beliebig orientierter Zweigströme [2]
2. Eintragen der richtungsabhängigen Spannungsabfälle
3. Aufstellen der linear unabhängigen Knoten- und Maschengleichungen
4. Lösen des Gleichungssystems

Bemerkung:

- Bei K - Knoten sind $(K-1)$ Knotengleichungen linear unabhängig.
- Bei z - Zweigen sind $m = z-(K-1)$ Maschengleichungen linear unabhängig.

Am vorliegenden Beispiel:



$$R_1 = 5\Omega$$
$$R_2 = 10\Omega$$
$$R_3 = 15\Omega$$
$$E_1 = -1,5V$$
$$E_2 = 1,5V$$

Die abgebildete Schaltung enthält, wie leicht zu sehen, zwei Knoten. Das heißt, wir werden $(K-1) = 1$ linear unabhängige Knotengleichungen finden können. Weiterhin gibt es drei Zweige. Dies bedeutet, dass wir $z-1 = 2$ Maschengleichungen aufstellen können.

K: $I_1 + I_2 + I_3 = 0$

M1: $-U_{E1} - U_{R1} + U_{R2} = 0$

M2: $-U_{R2} + U_{R3} + U_{E2} = 0$

Mit dem ohmschen Gesetz ergibt sich nach ein wenig umstellen das folgende lineare Gleichungssystem:

[2] Die Verbindung zweier Knotenpunkte wird als Zweig bezeichnet

I $I_1 + I_2 + I_3 = 0$

II $-R_1 I_1 + R_2 I_2 = U_{E1}$

III $-R_2 I_2 + R_3 I_3 = -U_{E2}$

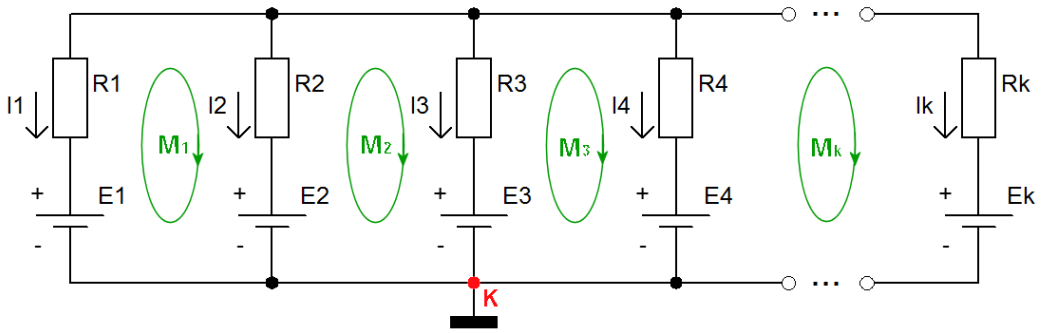
In Matrixform: $\begin{pmatrix} 1 & 1 & 1 \\ -R_1 & R_2 & 0 \\ 0 & -R_2 & R_3 \end{pmatrix} \cdot \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} 0 \\ U_{E1} \\ -U_{E2} \end{pmatrix}$

Mit Zahlenwerten: $\begin{pmatrix} 1 & 1 & 1 \\ -5 & 10 & 0 \\ 0 & -10 & 15 \end{pmatrix} \cdot \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} 0 \\ -1,5 \\ -1,5 \end{pmatrix}$

Dieses Gleichungssystem gilt es nun zu lösen. Auch wenn das bei diesem System eher leicht fällt, soll es im nächsten Punkt erst einmal um das Lösen von Gleichungssystemen im Allgemeinen gehen.

Erweiterte Zielsetzung:

An dieser Stelle möchte ich noch einmal kurz an meine erweiterte Zielstellung erinnern.



Die Ströme der abgebildeten Schaltung sollen für beliebige k zu berechnen sein. Das bedeutet, dass auch hierfür ein entsprechendes (variables) Gleichungssystem aufgestellt werden muss.

Betrachtet man die Schaltung für verschiedene k, so ergibt sich folgende Anzahl von Knoten, Zweigen und daraus resultierend Knoten- und Maschengleichungen.

	Knoten (K)	**Zweige (z)**	**(K-1)**	**z-(K-1)**
$k=2$	0	0	-1	1
$k=3$	2	3	1	2
$k=4$	4	6	3	3
$k=5$	6	9	5	4
$k=6$	8	12	7	5
k	$2\cdot(k-2)$	$3\cdot(k-2)$	$2k-5$	$k-1$

Wie zu sehen erhält man stets wesentlich mehr Gleichungen als Unbekannte. Es kann also sowohl mit Knoten- als auch Maschengleichungen gearbeitet werden. An dieser Stelle soll mit $k-1$ Maschengleichungen und einer Knotengleichung fortgesetzt werden.
Es ergibt sich folgendes Gleichungssystem in Matrixform:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 & 1 \\ -R_1 & R_2 & 0 & 0 & \cdots & 0 & 0 \\ 0 & -R_2 & R_3 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -R_3 & R_4 & \cdots & 0 & 0 \\ 0 & 0 & 0 & -R_4 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & R_{k-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & -R_{k-1} & R_k \end{pmatrix} \cdot \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ \vdots \\ I_{k-1} \\ I_k \end{pmatrix} = \begin{pmatrix} 0 \\ E_1 - E_2 \\ E_2 - E_3 \\ E_3 - E_4 \\ \vdots \\ E_{k-2} - E_{k-1} \\ E_{k-1} - E_k \end{pmatrix}$$

Dieses lineare Gleichungssystem umfasst somit in seiner Komplexität auch die eigentliche Aufgabenstellung. Es ist also nur noch diese Aufgabe zu lösen.

4 Lösen linearer Gleichungssysteme

Im nächsten Abschnitt soll es, losgelöst von der Aufgabenstellung, um lineare Gleichungssysteme gehen. Dazu folgen Herangehensweisen für Standardlösungsverfahren und der Gauß-Jordan-Algorithmus. Im letzten Teil dieses Abschnittes soll es dann um die Implementierung des Gauß-Jordan-Algorithmus unter C/C++ mit dem PC gehen.

4.1 Grundlagenwissen

Hinweis: Ein großer Teil der folgenden, mathematischen Sachverhalte ist dem Vorlesungsskript[3] von Prof. David J. Green entnommen.

Definition: Lineares Gleichungssystem

Sei k ein Körper. Unter einem linearen Gleichungssystem versteht man ein System von m Gleichungen, in dem die Skalare $a_{ij}, b_i \in k$ alle bekannt sind.

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & = & b_m \end{array}$$

Es gilt, das Gleichungssystem für die n unbekannten Elemente $x_1, \ldots, x_n$ von k zu lösen.
Sind alle $b_i = 0$, so wird das Gleichungssystem als homogen bezeichnet.

Matrixdarstellung

Sei $A \in M(m \times n, k)$ die Matrix mit $A_{ij} = a_{ij}$ für alle i,j.

Sei $\vec{b} \in k^m$ der Spaltenvektor $\vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$.

Sei $\vec{x} \in k^n$ der Spaltenvektor $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$.

Das obige Gleichungssystem lässt sich dann wie folgt darstellen:
$A \cdot \vec{x} = \vec{b}$

[3] siehe Literaturverzeichnis

Oder:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Oder:

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right)$$

Über- und unterbestimmte lineare Gleichungssysteme

Überbestimmt: Enthält das Gleichungssystem mehr Gleichungen als Unbekannte, so wird dieses überbestimmt genannt.
Sofern sich die Gleichungen nicht widersprechen kann man selbst wählen welche Gleichungen man zur Lösung verwendet.

Unterbestimmt: Enthält das Gleichungssystem weniger Gleichungen als Unbekannte, so wird dieses unterbestimmt genannt.
In diesem Fall lassen sich nicht alle Unbekannten explizit lösen. Es müssen die überschüssigen Variablen als konstant angenommen und damit der Lösungsraum bestimmt werden.

4.2 Standardverfahren

Da diese Verfahren zum Schulstoff gehören, hier nur kurze Erläuterungen.

Gleichsetzungsverfahren Wie der Name es schon sagt, nutzt man die Möglichkeit Gleichungen gleichzusetzen um Variablen oder ganze Terme aus dem System zu eliminieren. Hierzu werden zwei Gleichungen nach dem selben Term umgestellt und dann gleich gesetzt. Die übrigen Variablen werden durch das Gleiche, oder andere Verfahren bestimmt und mit deren Lösung die zu Beginn eliminierte Variable ermittelt.

Einsetzungsverfahren Hierbei wird eine Gleichung nach einer Variablen aufgelöst und in die restlichen Gleichungen des Systems eingesetzt. Auch hier wird die eingesetzte Variable am Ende mit den Lösungen der Übrigen bestimmt. Nachteil dieses Verfahrens ist, dass es recht schnell sehr aufwendig werden kann.

Additionsverfahren Das Additionsverfahren stellt die Grundlage des gaußschen Eliminationsverfahrens dar. Es wird versucht durch die Addition oder Subtraktion zweier Gleichungen eine oder gleich mehrere Variable aus dem System zu eliminieren. Am Ende wird auch hier die eliminierte Variable sukzessive mit Hilfe der Anderen bestimmt.

weitere Verfahren

Cramer'sche Regel
Verwendet Determinanten zum lösen des Gleichungssystems.

Cholesky-Zerlegung
Ist eine Variante des Gauß-Verfahrens.

Splitting-Verfahren
Sind iterative Verfahren (Bsp. Gauß-Seidel- oder Jacobi-Verfahren).

vorkonditionierte Krylow-Unterraum-Verfahren
Sind moderne und schnelle Verfahren (vor allem für große Systeme).

4.3 Der Gauß-Jordan-Algorithmus

Der Gauß-Jordan-Algorithmus ist ein Algorithmus aus den mathematischen Teilgebieten der linearen Algebra und Numerik. Das Verfahren ist nach Carl Friedrich Gauß, dem Entdecker des gaußschen Eliminationsverfahrens und Wilhelm Jordan benannt, der das Verfahren zum Gauß-Jordan-Algorithmus weiter entwickelte.
Vorteil dieses Verfahrens ist, dass man aus dem bearbeiteten Gleichungssystem (in Matrixdarstellung) dessen Lösung direkt ablesen kann.

4.3.1 Elementare Zeilenoperationen

Bevor wir uns näher mit dem Algorithmus an sich beschäftigen können muss der Begriff der elementaren Zeilenoperationen geklärt werden.
Sei $A \in M(m \times n, k)$ eine Matrix. Die folgenden drei Wege um aus A eine neue Matrix in $M(m \times n, k)$ zu machen heißen elementare Zeilenoperationen:

Typ 1 Zwei Zeilen vertauschen

Typ 2 Eine Zeile mit $\lambda \in k^x := k \setminus \{0\}$ multiplizieren.

Typ 3 Das λ-fache einer Zeile zu einer anderen Zeile addieren. ($\lambda \in k$)

> **Satz**
> Ändert man die Matrix A durch eine oder mehrere elementare Zeilenoperationen, so ändert sich der Nullraum LR(A,0) nicht. Auch wenn man Nullzeilen streicht oder hinzufügt ändert sich der Nullraum nicht.

Dies bedeutet, dass man durch elementare Zeilenoperationen die Matrix eines Gleichungssystems bearbeiten kann, ohne die Lösungsmenge des Selben zu verändern.

4.3.2 Das Gaußsche Eliminationsverfahren

Das gaußsche Eliminationsverfahren, wie auch der Gauß-Jordan-Algorithmus, wenden elementare Zeilenoperationen an, um eine Matrix A in eine besondere Form zu bringen.

Zeilenstufenform und strenge Zeilenstufenform
Diese Form wird als Zeilenstufenform oder strenge Zeilenstufenform bezeichnet. Aus dieser Form lässt sich das Ergebnis eines linearen Gleichungssystems mehr oder weniger direkt ablesen.

Bsp. für Zeilenstufenform
$$\begin{pmatrix} \underline{1} & 5 & 4 & 3 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & \underline{2} & 1 & 3 \\ 0 & 0 & 0 & \underline{7} & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Bsp. für strenge Zeilenstufenform
$$\begin{pmatrix} \underline{1} & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \underline{1} & 0 & 0 \\ 0 & 0 & 0 & \underline{1} & 0 \\ 0 & 0 & 0 & 0 & \underline{1} \end{pmatrix}$$

Die unterstrichenen Einträge werden Pivotstellen genannt.
Um nun aus diesen Matrizen die Lösung des entsprechenden Gleichungssystems zu ermitteln muss man wie folgt vorgehen:

Bei Zeilenstufenform

$$\left(\begin{array}{ccc|c} 1 & 2 & 1 & 5 \\ 0 & 2 & 1 & 4 \\ 0 & 0 & 1 & 3 \end{array}\right)$$

Die gewöhnliche Zeilenstufenform entsteht nach Anwendung des Gauß-schen Eliminationsverfahrens. Hier muss nun die Variable aus der letzten Matrixzeile bestimmt werden. $x_3 = 3$
Nun werden die Anderen Variablen mithilfe von x_3 bestimmt.
$2 \cdot x_2 + x_3 = 4 \Rightarrow x_2 = \frac{1}{2}$
$x_1 + 2 \cdot x_2 + x_3 = 5 \Rightarrow x_1 = 1$

Bei strenger Zeilenstufenform

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 3 \end{array}\right)$$

Auf diese Form der Matrix zu kommen ist Ziel des Gauß-Jordan-Algorithmus. Bei einem weder über- noch unterbestimmten Gleichungssystem gelingt dies auch immer. Die Lösung lässt sich direkt ablesen:
$x_1 = 1$
$x_2 = \frac{1}{2}$
$x_3 = 3$

Vorgehen

Bevor man natürlich auf eine solche Form des Gleichungssystems kommt, muss man einen entsprechenden Algorithmus anwenden um die gegeben Matrix passend zu verändern.

Erste Pivotstelle suchen

- Die erste Spalte muss einen Eintrag $\neq 0$ enthalten
- Gegebenenfalls durch das Tauschen zweier Zeilen dafür sorgen, dass $A_{1,1} \neq 0$ ist.
- Die erste Zeile mit $1/A_{1,1}$ multiplizieren, damit $A_{1,1} = 1$ ist.

Unterhalb der Pivotstelle kehren

- Für jede Zeile i unterhalb von $A_{1,1}$ das $A_{i,1}$-fache der ersten Zeile von dieser abziehen.

Nächste Pivotstelle suchen

- Für ein weder unter- noch überbestimmtes Gleichungssystem wird sich die nächste Pivotstelle bei $A_{2,2}$ finden lassen. (Gegebenenfalls wieder Zeilen unterhalb von Zeile 2 mit dieser tauschen.)
- Sei B die Matrix, die aus den Zeilen 2 bis m besteht. Man bringe B auf Zeilenstufenform (Rekursion).

Matrix auslesen

- Nun befindet sich die Matrix in einfacher Zeilenstufenform und das Ergebnis lässt sich wie oben beschrieben ermitteln.

Gauß-Jordan

Der Gauß-Jordan-Algorithmus stellt eine nur sehr geringfügige Erweiterung des Gaußschen Eliminationsverfahrens dar:

Strenge Zeilenstufenform herstellen

- Zusätzlich zum kehren unterhalb der gerade ermittelten Pivotstelle, wird auch darüber gekehrt.

Matrix auslesen

- Nun befindet sich die Matrix in strenger Zeilenstufenform und das Ergebnis kann wie oben beschrieben direkt abgelesen werden.

4.4 Implementierung mit dem PC

Am Besten lässt sich das Problem sicher in der praktischen Anwendung verstehen. Dazu wurde von mir ein Programm unter C/C++ mit Hilfe von Microsoft Visual C++ 2005 Express Edition entwickelt.
Da dieses Programm nur ein Zwischenschritt auf dem Weg zur Lösung der gegebenen Aufgabenstellung ist, ist dieses nicht übermäßig gegen falsche Eingaben abgesichert. Gibt man jedoch ein weder über- noch unterbestimmtes, lösbares Gleichungssystem in Matrixform ein, so wird dieses mithilfe des Gauß-Jordan-Algorithmus gelöst.
Auf den folgenden Seiten werde ich die einzelnen Funktionen meines Programmes so detailiert wie möglich versuchen zu erläutern. Der vollständige und zusammenhängende Quelltext befindet sich auf CD im Anhang. Ebenso natürlich eine für Windows compilierte Version.

Hauptprogramm

```
1  int main()
2  {
3    double **A, *ptrA;
4    int dimx, dimy;
5    int i;
6
7    cout << "Dieses Programm loest Ihnen ein lineares
8    Gleichungssystem mit n Gleichungen \nund n Unbekannten.
9    Dabei ist darauf zu achten, dass das LGS weder unter- noch
10   \nueberbestimmt ist.\n\n";
11   point1:
12   cout << "Geben Sie die Anzahl der Unbekannten ein: ";
13
14   cin >> dimy; dimx=dimy+1;
15
16   if (dimy <=0)
17   {
18     cout << "Falsche Angabe der Anzahl der Unbekannten!
19     Eingabe wiederholen:\n\n";
20     goto point1;
21   }
22
23   ptrA=(double*) malloc ((dimx+1)*(dimy+1)*sizeof(double));
24   A=(double**) malloc ((dimx+1)*sizeof(double));
25   for (i=0 ; i<=dimx ; i++) {A[i]=ptrA + dimy*i;};
26
27   insertMatrix(A, dimx, dimy);
28   gauss_jordan(A, dimx, dimy);
29   outputMatrix(A, dimx, dimy);
30   readoutMatrix(A, dimx, dimy);
31
32   free (A); free (ptrA);
33
34   cout << "Druecken Sie eine beliebige Taste zum beenden...";
35   FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
36   getch();
37   return 0;
38 }
```

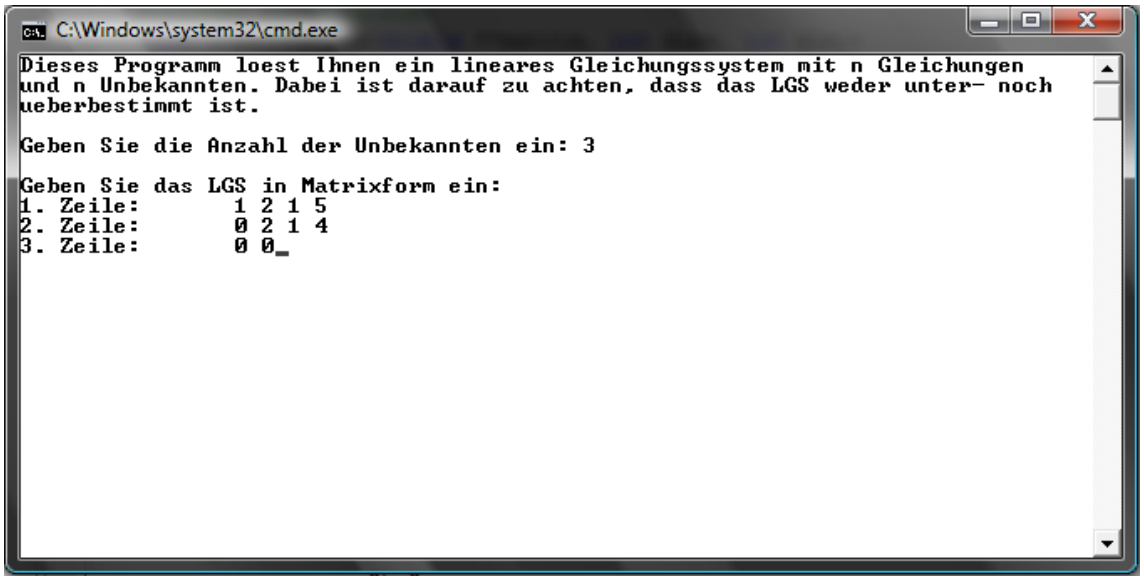
Zunächst braucht natürlich jedes Programm eine Hauptfunktion welche hier dargestellt ist.
Als Erstes erfolgt die Variablendeklaration (Z3-5). **A und *ptrA werden für die Matrix benötigt, dimx und dimy sind die Dimensionen der Matrix.
In den Zeilen 7 bis 10 folgen einige erklärende Worte und in den Zeilen 12 bis 14 die Eingabe der Anzahl der Variablen. Dabei bestimmt die Anzahl der Variablen die Zahl der erforderlichen Gleichungen, also Zeilen der Matrix (dimy). Damit lässt sich natürlich auch $dimx = dimy + 1$ festlegen, da wir immer von einem weder unter- noch überbestimmten Gleichungssystem ausgehen.
Die Zeilen 16 bis 21 sorgen für eine Abfrage ob es sich bei der Anzahl der Variablen um eine Zahl > 0 handelt. Ist dies nicht der Fall wird eine Fehlermeldung ausgegeben und im Quelltext in Zeile 11 zurückgesprungen.
Anschließend wird der Speicherplatz für die Matrix mit dem zwar recht alten aber robusten Befehl „malloc“ angefordert (Z23-25). Es folgen in den Zeilen 27 bis 30 einige Funktionen die später näher erläutert werden. insertMatrix lässt den Nutzer die Matrix eingeben, gauss_jordan bringt diese in strenge Zeilenstufenform, outputMatrix gibt die vollständige Matrix aus und readoutMatrix gibt die Lösung noch einmal einzeln an.
Abschließend wird in den Zeilen 34 bis 36 dafür gesorgt, dass sich die Konsole nicht sofort wieder schließt, mit 32 der Speicher für die Matrix wieder freigegeben und mit 37 dem System eine positive Antwortnachricht „return 0“ übergeben.

Einlesen der Matrix - Die Funktion insertMatrix

```
1  int insertMatrix(double **matrix, int dimx, int dimy)
2  {
3    int i, j;
4    cout << "\nGeben Sie das LGS in Matrixform ein:\n";
5    for (i=1 ; i<=dimy ; i++)
6    {
7      cout << i << ". Zeile:\t";
8      for (j=1 ; j<=dimx ; j++) {cin >> matrix[j][i];}
9    }
10   return 0;
11 }
```

Die Funktion insertMatrix dient, wie bereits gesagt dazu, den Nutzer die Matrix eingeben zu lassen. Dabei wird wie folgt vorgegangen:
Nachdem der Nutzer in Zeile 4 kurz gebeten wurde die Matrix einzugeben folgt

in Zeile 5 die erste Zählschleife. Die Variable i durchläuft dabei alle Zeilen von $i = 1$ bis $i = dimy$. Zeile und Spalte 0 bleiben die ganze Zeit über vollkommen unbenutzt. Es wird noch kurz ausgegeben welche Zeile gerade beschrieben wird, ehe die for-Schleife beginnt, die alle Spalten durchläuft und die entsprechenden Werte einließt. Vorteilhaft bei dieser Variante ist, dass Windows mit Leerzeichen getrennte Werte nacheinander einließt und man so eine Zeile durch Leerzeichen getrennt hinter einander eingeben kann. Das Ganze sieht dann folgendermaßen aus:



Abschließend wird in Zeile 10 wieder eine positive Antwort an das Hauptprogramm übergeben.

Die Gauß-Jordan-Funktion

```
1  int gauss_jordan(double **matrix, int dimx, int dimy)
2  {
3    int i, j;
4    for (j=1 ; j<dimx ; j++)
5    {
6      if (matrix[j][j]==0)
7      {
8        for (i=j+1 ; i<=dimy ; i++)
9        {
10         if (matrix[j][i]!=0)
11         {
12           swapLines(matrix, dimx, dimy, j, i);
13           break;
14         }
```

```
15        }
16      }
17      normalise (matrix, j, j, dimx);
18      sweep (matrix, j, j, dimx, dimy);
19    }
20    return 0;
21 }
```

Diese Funktion stellt das Herzstück des Programms dar und ist dennoch, durch weitere Einzelfunktionen, recht kurz gehalten. Ich habe mich dazu entschlossen den Algorithmus explizit und nicht rekursiv aufzubauen. Zum einen ist er sehr viel schneller und leichter durchschaubar explizit aufzuschreiben. Zum anderen ist die explizite Variante vor allem bei sehr großen Matrizen wesentlich Speicherschonender, da immer wieder der Speicher der Matrix A aus dem Hauptprogramm umgeschrieben wird und kein Neuer vom System angefordert werden muss.
Explizit läuft der Algorithmus folgendermaßen ab:
Es werden alle Spalten von $j = 1$ bis $j = dimx$ durchlaufen. Das bedeutet also, Alle bis auf die letzte Spalte. Hier wird nun jeweils geprüft ob der Eintrag auf der Hauptdiagonalen[4] Null ist. Da wir von einem weder unter- noch über bestimmten linearen Gleichungssystem ausgehen, wird es immer möglich sein hier eine Pivotstelle zu erzeugen. Ist dieser Eintrag nun Null, so wird mit der nächsten for-Schleife (Z8-15) ausschließlich unterhalb dieser Zeile nach Einträgen $\neq 0$ in dieser Spalte gesucht. Ist ein Eintrag gefunden, so werden die beiden Zeilen mit der Funktion „swapLines“ vertauscht und die Schleife abgebrochen. Im Anschluss wird die betroffene Zeile mit der Funktion normalise (Z17) nach der Pivotstelle normiert und sowohl über als auch unter der Pivotstelle mit der Funktion sweep (Z18) gekehrt. Abschließend wird auch hier mit „return 0“ eine positive Antwortmeldung an das Hauptprogramm zurückgegeben.

[4] Der Begriff Hauptdiagonale ist eigentlich für quadratische Matrizen reserviert. Da es sich hier aber um eine erweiterte Matrix handelt, deren linke Hälfte quadratisch ist, ist deren Hauptdiagonale gemeint.

Zeilentausch - Die Funktion swapLines

```
1  int swapLines(double **matrix,int dimx,int dimy,int Z1,int Z2)
2  {
3    int j;
4    double help;
5
6    for (j=1 ; j<=dimx ; j++)
7    {
8      help=matrix[j][Z1];
9      matrix[j][Z1]=matrix[j][Z2];
10     matrix[j][Z2]=help;
11   }
12   return 0;
13 }
```

Das Tauschen zweier Zeilen ist wohl die grundlegendste elementare Zeilenoperation. Aus diesem Grund war diese Funktion auch eine der Ersten die ich geschrieben hatte. Das Prinzip ist denkbar einfach. Die Funktion bekommt zusätzlich zu den Werten für die Matrix noch die Angabe zweier Zeilen übergeben. Nun werden mit einer for-Schleife (Z6-11) alle Spalten abgefahren, dabei wird zunächst der Eintrag von Zeile 1 zwischengespeichert, dann Zeile 1 mit dem Wert von Zeile 2 überschrieben und abschließend Zeile 2 mit dem Wert belegt der vorher auf Zeile 1 stand.
Dieses Verfahren bietet den Vorteil, dass immer nur ein Eintrag und nicht die ganze Zeile zwischengespeichert werden muss.

Normieren einer Zeile - Die Funktion normalise

```
1  int normalise (double **matrix,int zeile,int stelle,int dimx)
2  {
3    int j;
4    double help=matrix[stelle][zeile];
5
6    if (help==0) return 1;
7    for (j=1 ; j<=dimx ; j++)
8    {
9      (matrix[j][zeile])=(matrix[j][zeile])/help;
10   }
11   return 0;
12 }
```

Auch das Normieren einer Zeile nach einer bestimmten Stelle gehört prinzipiell zu den elementaren Zeilenoperationen. Typ 2 besagt, dass man eine Zeile mit einer Zahl $\neq 0$ multiplizieren darf. Wählt man diese Zahl entsprechend, so erreicht man, dass ein bestimmter Eintrag der Zeile eins wird.
Auch dieser Funktion werden mehr als nur die Informationen über die Matrix übergeben. Sie bekommt zusätzliche die Zeile die normiert werden soll und nach welcher Stelle dieser Zeile das geschehen soll. Zu Beginn wird eine Hilfsvariable deklariert (Z4) die mit der entsprechenden, zu normierenden Stelle belegt wird, da diese während des Schleifendurchlaufes irgendwann überschrieben wird. Betrachtet man das Problem nur mit dem hier vorliegenden Algorithmus, so könnte man diesen Schritt umgehen indem man die Zählschleife rückwärts laufen und die Stelle somit als letztes ändern lässt. Die Hilfsvariable ist an dieser Stelle die sicherere Variante.
In Zeile 6 wird noch kurz geprüft, ob die Stelle, die normiert werden soll $= 0$ ist. In diesem Fall ist eine Normierung natürlich nicht möglich, eine Fehlermeldung wird an die übergeordnete Funktion zurückgegeben. Da wir bei der Anwendung dieser Funktion aber vorher Zeilen passend tauschen kann dieser Fall eigentlich nicht vorkommen.
Nun beginnt eine for-Schleife (Z7-10), die alle Spalten durchläuft und jeden Eintrag durch den zu normierenden Eintrag (der auf help steht) teilt. Und wieder wird ein „return 0“ zum Abschluss ausgeführt.

Kehren über und unter einer Pivotstelle - Die Funktion sweep

```
1  int sweep (double **matrix,int z_nicht,int stelle,int dimx,
2                                                    int dimy)
3  {
4    int i;
5    for (i=1 ; i<=dimy ; i++)
6    {
7      if (i==z_nicht) continue;
8      eliminate(matrix,i,z_nicht, stelle, dimx);
9    }
10   return 0;
11 }
```

Das Ziel dieser Funktion ist es nun einerseits dafür zu sorgen, dass die Einträge unter der Pivotstelle wie es beim gaußschen Eliminationsverfahren üblich ist Null werden. Außerdem müssen aber auch über der Pivotstelle wie es beim Gauß-Jordan-Algorithmus nötig ist, Nullen erzeugt werden. Das Vorgehen ist durch eine weiter

geschachtelte Funktion wieder recht kurz gehalten. Eine for-Schleife (Z5-9) durchläuft alle Zeilen von $i = 1$ bis $i = dimy$, wobei die Zeile „z_nicht“ übersprungen wird (Z7). Das ist die Zeile in der sich die Pivotstelle befindet, die erhalten bleiben soll. Es wird jeweils die Funktion „eliminate“ aufgerufen um den entsprechenden Eintrag zu Null zu machen.

Kehren einer Stelle - Die Funktion eliminate

```
1  int eliminate (double **matrix,int z_anwend,int z_quelle,
2                 int stelle,int dimx)
3  {
4    int j;
5    if (matrix[stelle][z_anwend]==0) return 0;
6    if (matrix[stelle][z_quelle]==0) return 1;
7    double help=matrix[stelle][z_anwend]/matrix[stelle][z_quelle];
8
9    for (j=1 ; j<=dimx ; j++)
10   {
11     (matrix[j][z_anwend])=(matrix[j][z_anwend])
12                           -(matrix[j][z_quelle]*help);
13   }
14   return 0;
15 }
```

Diese Funktion nutzt nun den letzten Typ von elementaren Zeilenopertionen, Typ 3. Dieser besagt, dass man zu einer Zeile das λ-fache einer anderen addieren darf.
Als Input bekommt die Funktion natürlich die Daten der Matrix, aber auch die Zeile in der ein Wert zu Null gemacht werden soll, eine Quellzeile, deren Gleichung für die Operation genutzt wird und die Stelle des Wertes der zu Null gemacht werden soll.
In der Funktion sweep wurde eliminate so aufgerufen, dass die Anwendzeile mit der for-Schleife durchläuft, die Quellzeile die nicht zu ändernde Zeile ist in der sich die Pivotstelle befindet und die Stelle wiederum die bleibt die sweep selbst von gauss_jordan bekommen hat.
In eliminate wird zunächst geprüft ob die Stelle bereits Null ist (Z5), dann wird sofort mit positiver Antwort beendet. Und es wird geprüft ob die Quellstelle Null ist (Z6), dann würde die Funktion Probleme mit division durch Null bekommen. Aus diesem Grund wird hier abgebrochen. Anschließend wird wieder eine Hilfsvariable deklariert. Diese wird mit dem Quotienten aus Anwendstelle und Quellstelle belegt.

Multipliziert man die Quellstelle mit diesem Quotienten, so nimmt diese den Wert der Anwendstelle an. Das wird im folgenden ausgenutzt. In einer for-Schleife (Z9-13) werden alle Spalten $j = 1$ bis $j = dimx$ durchlaufen und von ihren Einträgen das „help"-fache der Quellzeileneinträgen abgezogen. Da bei der Anwendstelle beide Einträge gleich sind wird der Anwendeintrag Null.

Damit ist der Algorithmus eigentlich beendet. Es folgt lediglich die Ausgabe des Ergebnisses!

Vollständige Matrixausgabe - Die Funktion outputMatrix

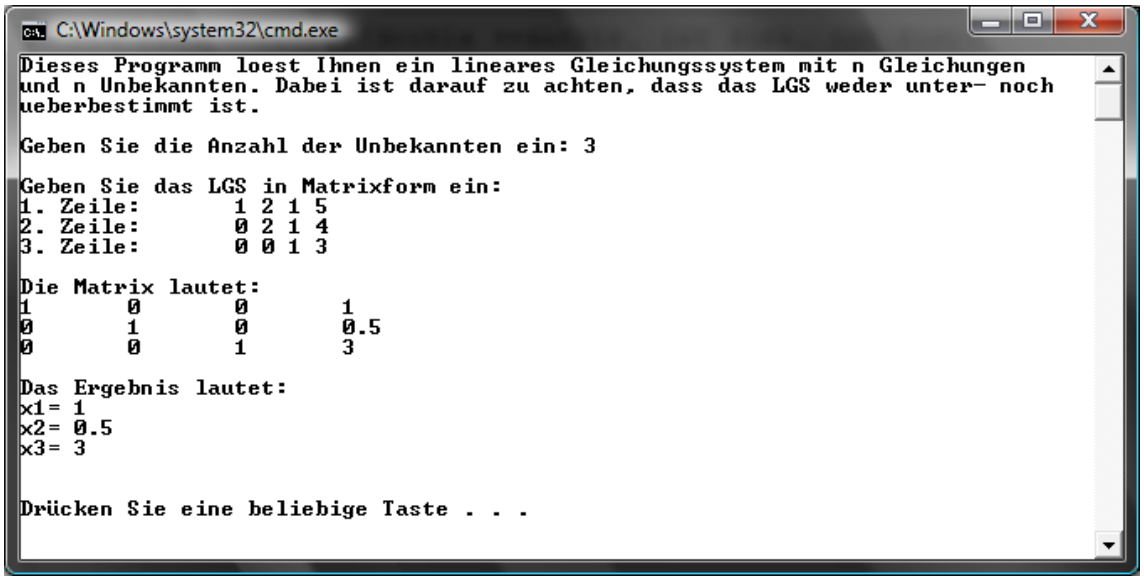
```
1  int outputMatrix(double **matrix, int dimx, int dimy)
2  {
3    int i, j;
4    cout << "\nDie Matrix lautet:";
5
6    for (i=1 ; i<=dimy ; i++)
7    {
8      cout << "\n";
9      for (j=1 ; j<=dimx ; j++) {cout << matrix[j][i] <<"\t";}
10   }
11
12   return 0;
13 }
```

Diese Funktion ist im Prinzip nicht erforderlich. Sie stellt aber eine gute Kontrolle dar, ob der Algorithmus richtig gearbeitet hat. Im Grunde funktioniert sie genau umgekehrt zu insertMatrix. Zwei for-Schleifen (Z6-10) durchlaufen alle Zeilen und Spalten und geben dabei jeweils den entsprechenden Matrixeintrag aus. Am Ende einer jeden Zeile wird nach unten gesprungen (Z8).
Erwartet wird natürlich eine Matrix in strenger Zeilenstufenform. Ist eine Nullzeile enthalten, so wurde ein unterbestimmtes lineares Gleichungssystem eingegeben (einige Zeilen waren linear voneinander abhängig). In diesem Fall wird die folgende Funktion (readoutMatrix) nicht richtig arbeiten.

Ausgabe des Ergebnisses - Die Funktion readoutMatrix

```
1  int readoutMatrix(double **matrix, int dimx, int dimy)
2  {
3    int i;
4    cout << "\n\nDas Ergebnis lautet:\n";
5
6    for (i=1 ; i<=dimy ; i++)
7    {
8      cout << 'x' << i << "= " << matrix[dimx][i] << "\n";
9    }
10
11   cout << "\n\n";
12   return 0;
13 }
```

Die eigentliche Ausgabe wird mit readoutMatrix realisiert. Da unsere Matrix nun in strenger Zeilenstufenform ist brauchen wir nur noch die hinterste Spalte von oben nach unten durchfahren und die Ergebnisse ausgeben. Dazu dient wieder eine Schleife (Z6-9). Die Variablen werden hier als $x_1 \dots x_{dimy}$ bezeichnet.
Am Ende sieht das dann folgendermaßen aus:



5 Berechnung von Netzwerken mit dem PC

Nach einführenden Worten zu elektrischen Schaltungen und der Herleitung eines Schema's zum Lösen der Aufgabenstellung folgte ein eher mathematischer Abschnitt über das Lösen linearer Gleichungssysteme. Der Höhepunkt dieses zweiten Abschnittes war ein Programm zum Lösen solcher Gleichungssysteme mit dem PC.
Im folgenden und letzten Abschnitt dieser Arbeit werde ich kurz einiges zur allgemeinen Simulation von elektrischen Schaltungen mit dem Computer ausführen und dabei ein weit verbreitetes Programm vorstellen. Mit Hilfe dieses Programms werde ich zu kontrollzwecken sowohl die eigentliche Aufgabenstellung als auch meine erweiterte Zielstellung (mit festgelegten Werten) berechnen lassen.
Letztendlich stelle ich detailiert am Quelltext dar, wie ich die Aufgabenstellung mit dem Gauß-Jordan-Algorithmus verknüpft habe und lasse die entsprechenden Werte berechnen. Es sollten die selben Ergebnisse entstehen wie sie das Simulationsprogramm zuvor ausgegeben hat.

5.1 Bekannte Software zur Netzwerkanalyse

Zur Netzwerkanalyse gibt es eine Vielzahl von Softwareangeboten die mehr oder weniger für bestimmte Anwendungen spezialisiert sind. Ein großer Nachteil derartiger Software ist, dass es für Windows nahezu keine kostenlosen Angebote gibt.

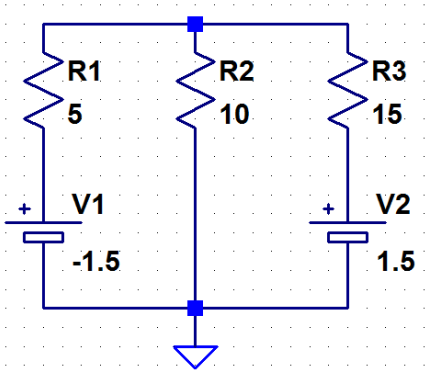
5.1.1 LTspice

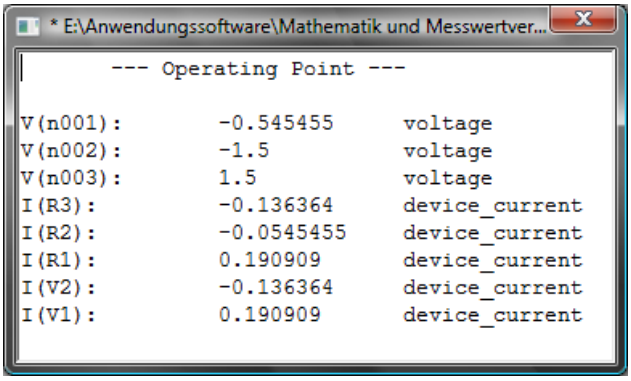
LTspice / SwitcherCadIII ist eines der wenigen freien Programme unter Windows. Es ist dem weiter verbreiteten Pspice sehr ähnlich. Wärend Pspice jedoch nur in der Studentenversion (sehr alte Version) kostenfrei ist, gibt es für LTspice auch die Vollversion als Freeware.[5] LTspice wird von der Firma Linear Technology Corporation zur Entwicklung von Schaltnetzteilen und Steuerbausteinen eingesetzt. Die Firma stellt die Software jedermann kostenfrei zur Verfügung. Es folgen Abbildungen aus

LTspice mit der Lösung der eigentlichen Aufgabenstellung und meiner erweiterten Zielsetzung.

[5] Download unter: http://www.linear.com/software

Eigentliche Aufgabe:





Erweitere Zielsetzung:

Betrachten wir, um die Aufgabe mit Inhalt zu füllen folgendes Problem: Die Birne eines Autoscheinwerfers ($U = 12V; P = 80W$) soll anstatt mit einer Autobatterie, mit mehreren versorgt werden um die Kapazität zu steigern. Die Batterien sollen folgende Werte haben:

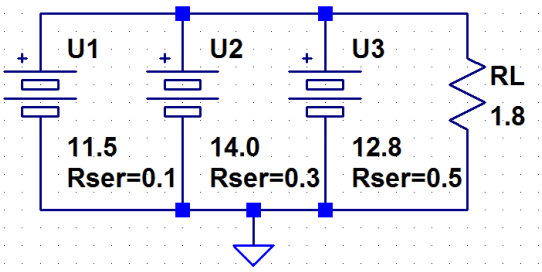
$U_1 = 11,5V; R_1 = 0,1\Omega$

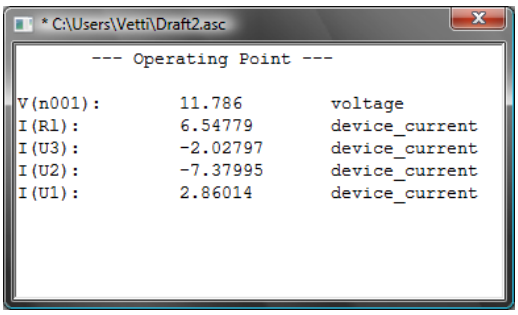
$U_2 = 14,0V; R_2 = 0,3\Omega$

$U_3 = 12,8V; R_3 = 0,5\Omega$

Den Widerstand R_L der Lampe erhält man aus $R_L = \frac{U^2}{P} = 1,8\Omega$

Es ergibt sich:





5.2 Verknüpfung von Netzwerkanalyse und Gauß-Jordan-Algorithmus

Nun haben wir Vergleichsmöglichkeiten geschaffen um den Algorithmus am Ende testen zu können. Doch bevor dies geschehen kann müssen die theoretischen Vorbetrachtungen aus Punkt 3.3.2 mit dem Gauß-Jordan-Algorithmus aus Punkt 4.4 verknüpft werden.

Hauptprogramm

```
1  int main()
2  {
3    double **A, *ptrA;
4    int dimx, dimy;
5    int i;
6
7    cout << "Dieses Programm berechneten Ihnen die Stroeoeme
8    durch die Widerstaende R1...Rk \nvon k parallel geschalteten
9    realen Spannungsquellen. \nDabei sind Rk die
10   Innenwiderstaende der Spannungsquellen.";
11   cout << "\n\nTipp: Fuer Uk=0 und Rk beliebig laesst sich eine
12   Lastsituation simulieren.\nHinweis: Ein- und Ausgaben
13   erfolgen in Ohm, Volt und Amper!";
14   point1:
15   cout <<"\n\nGeben Sie die Anzahl der Spannungsquellen ein: ";
16
17   cin >> dimy; dimx=dimy+1;
18
19   if (dimy <=0)
20   {
21     cout << "Falsche Angabe der Anzahl der Spannungsquellen!
22     Eingabe wiederholen:\n";
23     goto point1;
24   }
25
26   ptrA=(double*) malloc ((dimx+1)*(dimy+1)*sizeof(double));
27   A=(double**) malloc ((dimx+1)*sizeof(double));
28   for (i=0 ; i<=dimx ; i++) {A[i]=ptrA + dimy*i;};
29
30   initialiseMatrix(A,dimx,dimy);
31   insertConnection(A, dimx, dimy);
32   gauss_jordan(A, dimx, dimy);
33   readoutMatrix(A, dimx, dimy);
34
35   free (A); free (ptrA);
36
37   cout << "Druecken Sie eine beliebige Taste zum beenden...";
38   FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
```

```
39   getch();
40
41   return 0;
42 }
```

An der Hauptfunktion hat sich nicht viel verändert. Die erklärende Textausgabe ist ein klein wenig anders und es wird nach der Anzahl der Spannungsquellen und nicht der Variablen gefragt. Da die Anzahl der Spannungsquellen aber auch die Anzahl der Widerstände und somit die Anzahl der zu berechnenden Ströme bestimmt ist dies im Grunde das Selbe.
Der wirkliche Unterschied besteht im Aufruf diverser Funktionen in den Zeilen 30 bis 33. Die einzige Funktion, die hier aus dem vorigen Programm identisch geblieben ist, ist gauss_jordan mit ihren einzelnen Unterfunktionen.
Zu den anderen Funktionen werde ich im folgenden kommen:

Initialisieren der Matrix - Die Funktion initialiseMatrix

```
1 int initialiseMatrix(double **matrix, int dimx, int dimy)
2 {
3   int i, j;
4   for (i=1 ; i<=dimy ; i++)
5   for (j=1 ; j<=dimx ; j++) {matrix[j][i]=0;}
6   return 0;
7 }
```

In der Funktion insertmatrix beim Gauß-Jordan-Verfahren wurden alle Einträge der Matrix abgefahren und durch den Benutzer Werte eingegeben. Dies ist bei der im nächsten Punkt folgenden Funktion insertConnection nicht der Fall. Hier werden nur die Widerstände und Spannungsdifferenzen (wie in Punkt 3.3.2 erläutert) eingefügt. Der Rest der Matrix muss also mit Null belegt sein. Dies ist aber nicht standardmäßig der Fall, da wir den Speicher eben erst vom System angefordert haben. Wir müssen die Matrix zunächst also mit Nullen füllen. Dies gewährleistet die Funktion initialiseMatrix. Sie durchläuft in zwei for-Schleifen (Z4-5) alle Einträge der Matrix und belegt diese mit Null.

Die Werteeingabe - Die Funktion insertConnection

```
1  int insertConnection(double **matrix, int dimx, int dimy)
2  {
3    int i,j;
4    double help;
5
6    cout << "\nEs folgt die Eingabe der Widerstaende:\n";
7    for (i=2 ; i<=dimx ; i++)
8    {
9      point2:
10     cout << "R" << i-1 << "= "; cin >> help;
11     if (help<0)
12     {
13       cout << "Negative Widerstaende sind unzulaessig!
14       Eingabe wiederholen:\n";
15       goto point2;
16     }
17     matrix[i-1][i]=-help;
18     matrix[i-1][i-1]=help;
19     matrix[0][i-1]=help;
20   }
21
22   cout << "\nEs folgt die Eingabe der Spannungswerte:\n";
23   for (i=2 ; i<=dimx ; i++)
24   {
25     cout << "U" << i-1 << "= "; cin >> help;
26     matrix[dimx][i]=help; matrix[dimx][i-1]-=help;
27   }
28
29   for (j=1 ; j<dimx ; j++) {matrix[j][1]=1;}
30   matrix[dimx][1]=0;
31
32   return 0;
33 }
```

Diese Funktion stellt, mal abgesehen von der Funktion gauss_jordan, in diesem Programm das Herzstück dar. Sie stellt dem Gauß-Jordan-Algorithmus die Matrix zur Verfügung die er zur Berechnung der Ströme benötigt. Dabei arbeitet die Funktion wie folgt:

In den Zeilen 6 bis 20 erfolgt das Einlesen der Widerstände. Dabei werden die Widerstandswerte bereits passend auf die Matrix an die Stelle geschrieben an der sie am Ende auch stehen sollen. In einer for-Schleife (Z7-20) werden alle Zeilen von $i = 2$ bis $i = dimx$ durchlaufen. Das ist im Grunde eine Zeile mehr als die Matrix besitzt. Das spielt aber keine weitere Rolle (bezüglich Abstürzen) da diese Zeile nicht beschrieben wird (der Wert von i wird aber benötigt). Jeweils wird der Benutzer gebeten den $i-1$. Widerstand (also $2-1 \ldots dimx-1 = 1 \ldots dimy$) einzugeben. Das Ganze wird auf eine Variable help geschrieben (Z10). Es folgt eine Abfrage (Z11-16) ob ein negativer Wert eingegeben wurde. Ist dies der Fall wird eine Fehlermeldung ausgegeben und im Quelltext in Zeile 9 zurück gesprungen. Nach dieser Abfrage erfolgt das eigentliche Einlesen. Dabei wird der negative Wert des eingegeben Widerstandes auf die Position links von der Hauptdiagonalen geschrieben (Z17) und der positive Wert genau darüber (Z18). Ein besonderes Augenmerk muss hier auf Zeile 19 gelegt werden. Die Widerstände werden zusätzlich auf die 0. Spalte der Matrix geschrieben. Diese Spalte wird von keiner weiteren Funktion verändert. Es wird so gewährleistet, dass die Widerstände auch später noch zur Verarbeitung zur Verfügung stehen ohne einen weiteren dynamischen Array anlegen, übergeben bzw. verwalten zu müssen. In der Funktion readoutMatrix wird dies genutzt um zusätzlich zu den Strömen, die am Widerstand umgesetzte Leistung zu berechnen.
Wir haben nun die folgende Matrix erstellt:

$$\left(\begin{array}{c|cccccccc} R_1 & R_1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ R_2 & -R_1 & R_2 & 0 & 0 & \cdots & 0 & 0 & 0 \\ R_3 & 0 & -R_2 & R_3 & 0 & \cdots & 0 & 0 & 0 \\ R_4 & 0 & 0 & -R_3 & R_4 & \cdots & 0 & 0 & 0 \\ R_5 & 0 & 0 & 0 & -R_4 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ R_{k-1} & 0 & 0 & 0 & 0 & \cdots & R_{k-1} & 0 & 0 \\ R_k & 0 & 0 & 0 & 0 & \cdots & -R_{k-1} & R_k & 0 \end{array}\right)$$

Als nächstes folgt die Eingabe der Spannungswerte (Z22-27). Der Ablauf der for-Schleife ist der Selbe. Eine Abfrage ob es sich um negative Werte handelt entfällt, da Spannungen durchaus negativ eingegeben werden dürfen. Das Einlesen in Zeile 26 erfolgt überschreibend. Wir wollen ab Zeile 2 beginnend jeweils $E_{k-1} - E_k$ auf die letzte Spalte schreiben. Dazu wird der gerade eingelesene Wert zunächst auf die richtige Zeile geschrieben und weiterhin von der oberen abgezogen. Dies hat den Vorteil, dass die Zählschleife nur einmal durchlaufen werden muss.

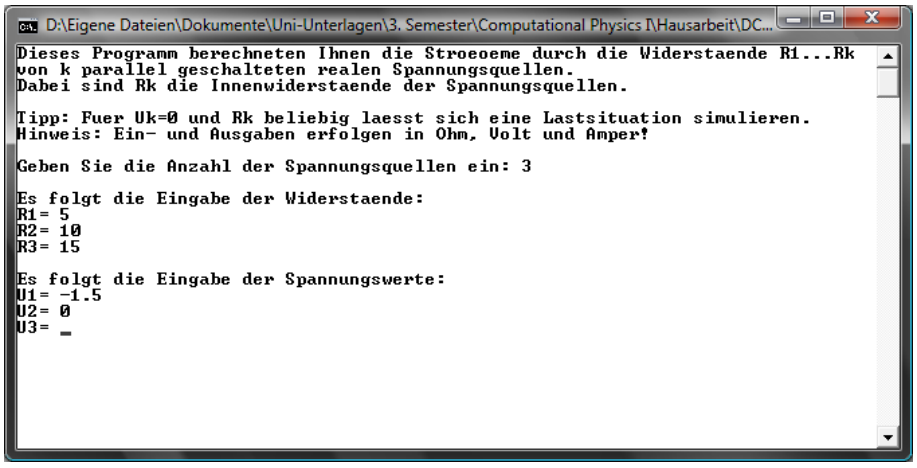
Die Matrix sieht nun wie folgt aus:

$$\left(\begin{array}{c|cccccccc} R_1 & R_1 & 0 & 0 & 0 & \cdots & 0 & 0 & -E_1 \\ R_2 & -R_1 & R_2 & 0 & 0 & \cdots & 0 & 0 & E_1 - E_2 \\ R_3 & 0 & -R_2 & R_3 & 0 & \cdots & 0 & 0 & E_2 - E_3 \\ R_4 & 0 & 0 & -R_3 & R_4 & \cdots & 0 & 0 & E_3 - E_4 \\ R_5 & 0 & 0 & 0 & -R_4 & \cdots & 0 & 0 & E_4 - E_5 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ R_{k-1} & 0 & 0 & 0 & 0 & \cdots & R_{k-1} & 0 & E_{k-2} - E_{k-1} \\ R_k & 0 & 0 & 0 & 0 & \cdots & -R_{k-1} & R_k & E_{k-1} - E_k \end{array}\right)$$

Im letzten Schritt schreiben wir Zeile 1 um. Diese soll schließlich die Hauptknotengleichung enthalten. Hätten wir diesen Teil als erstes gemacht, wäre es wesentlich schwieriger geworden die Widerstände und Spannungen einzulesen ohne den ersten und letzten Eintrag wieder durch R_1 und $-E_1$ zu überschreiben. Es folgt eine kleine for-Schleife (Z29), die die Einsen setzt und ein einzelner Befehl (Z30), der die abschließende Null ganz oben rechts einfügt.
Als Letztes wird wieder ein „return 0“ zurückgegeben und wir haben die fertige Matrix der folgenden Form erhalten:

$$\left(\begin{array}{c|cccccccc} R_1 & 1 & 1 & 1 & 1 & \cdots & 1 & 1 & 0 \\ R_2 & -R_1 & R_2 & 0 & 0 & \cdots & 0 & 0 & E_1 - E_2 \\ R_3 & 0 & -R_2 & R_3 & 0 & \cdots & 0 & 0 & E_2 - E_3 \\ R_4 & 0 & 0 & -R_3 & R_4 & \cdots & 0 & 0 & E_3 - E_4 \\ R_5 & 0 & 0 & 0 & -R_4 & \cdots & 0 & 0 & E_4 - E_5 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ R_{k-1} & 0 & 0 & 0 & 0 & \cdots & R_{k-1} & 0 & E_{k-2} - E_{k-1} \\ R_k & 0 & 0 & 0 & 0 & \cdots & -R_{k-1} & R_k & E_{k-1} - E_k \end{array}\right)$$

Im Programm sieht das Ganze dann folgendermaßen aus:



Ausgabe des Ergebnisses - Die Funktion readoutMatrix

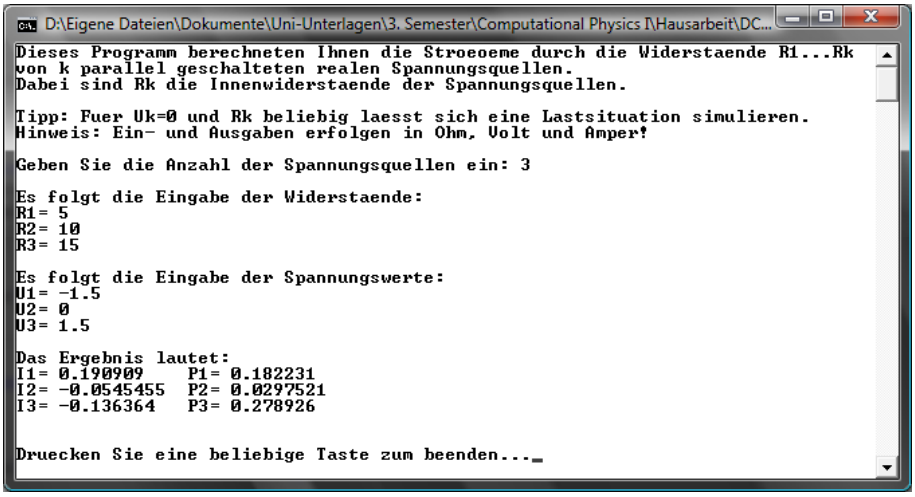
```
1  int readoutMatrix(double **matrix, int dimx, int dimy)
2  {
3    int i;
4    cout << "\nDas Ergebnis lautet:\n";
5    for (i=1 ; i<=dimy ; i++)
6    {
7      cout << 'I' << i << "= " << matrix[dimx][i] << "\tP" << i
8      << "= " << matrix[dimx][i]*matrix[dimx][i]*matrix[0][i]
9      << "\n";
10   }
11
12   cout << "\n\n";
13   return 0;
14 }
```

Die Funktion readoutMatrix wurde bereits beim Gauß-Jordan-Algorithmus zur Ausgabe des Endergebnisses genutzt, ist aber für dieses Programm ein wenig erweitert wurden. Die in insertConnection erstellte Matrix wurde mittlerweile mithilfe der Funktion gauss_jordan in strenge Zeilenstufenform gebracht und muss nun nur noch ausgelesen werden.
Im Prinzip ist die Ausgabe wieder simpel. Eine for-Schleife (Z5-10) durchläuft alle Zeilen von $i = 1$ bis $i = dimy$ und gibt die entsprechenden Werte des Stromes aus. Zusätzlich dazu passiert aber noch folgendes: Nach der Gleichung $P = I^2 \cdot R$ lässt sich die Leistung berechnen. Dies geschieht (gleichzeitig mit der Ausgabe) in Zeile 8. Die Werte der Widerstände wurden (wie bereits weiter vorn erläutert) in der 0. sonst unbenutzten Spalte der Matrix gespeichert und konnten somit hier aufgerufen werden.

5.3 Lösung der Aufgabenstellung

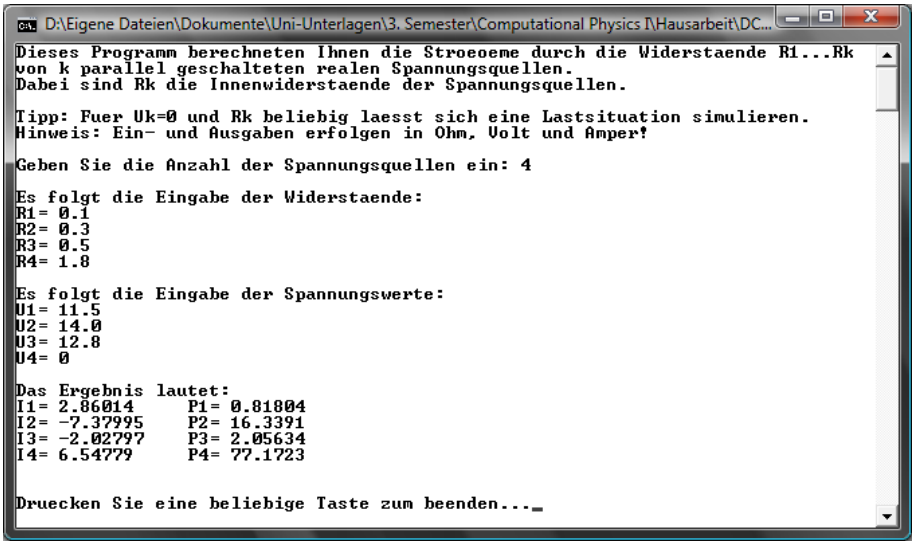
Eigentliche Aufgabenstellung

Gibt man nun die Größen der Aufgabenstellung in das entwickelte Programm ein (U_2 ist dabei $= 0$ zu wählen), so erfolgt diese Ausgabe:



Erweitere Zielsetzung:

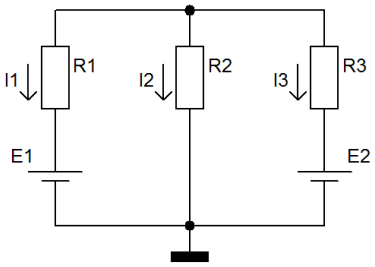
Um auch die erweiterten Funktionen meines Programmes testen zu können wurde weiter vorn ein Fallbeispiel eingeführt. Die Werte dieser Aufgabenstellung in das Programm eingegeben führt zu folgendem Ergebnis:



6 Schlusswort

Vergleicht man die Endergebnisse der verschiedenen Verfahren, d.h handschriftliches Lösen, das Lösen mit LTspice oder das Berechnen mit dem selbstgeschriebenen Programm, so erkennt man das sämtliche Ergebnisse übereinstimmen.
Natürlich habe ich weitere Testläufe mit wesentlich mehr Spannungsquellen durchgeführt und mit LTspice überprüft. Das Programm befindet sich ebenso auf CD im Anhang und kann gerne genutzt werden um die Funktionstüchtigkeit der von mir erstellten Programme weiter zu prüfen.

Für die eigentliche Aufgabenstellung folgt das Ergebnis:



$$I_1 = 190,9mA$$
$$I_2 = -54,5mA$$
$$I_3 = -136,4mA$$

Zum Ende dieser Arbeit hin möchte ich feststellen, dass es mir gelungen ist, die gegebene Aufgabenstellung zu bewältigen, meine eigene erweiterte Zielsetzung zu erfüllen und das Ganze meiner Meinung nach auch gut in Form dieser Arbeit zusammenzufassen.
Weitere erklärende Worte sind an dieser Stelle meiner Meinung nach nicht nötig. Ich möchte die Arbeit mit einem kleinen Zitat beenden...

„My software never has bugs.
It just develops random features!“

7 Literaturverzeichnis

|1| PD Dr. Frank Schmidl:
Script zur Vorlesung „Elektronik für Physiker“
WS 07/08

|2| Prof. David J. Green:
Script zur Vorlesung „Lineare Algebra und Analytische Geometrie“
WS 06/07

|3| Prof. Dr. H. Süße:
Mitschriften zur Vorlesung „Informatik für Physiker“
SS 07

|4| Wolfgang Demtröder:
Experimentalphysik 2, Elektrizität und Optik, ISBN 3-540-33794-6
4. Auflage, Springer-Verlag

|5| Dr. Jürgen Wolff:
Mathematik - Oberstufe, Analysis, ISBN 3-464-57216-1
1 Auflage, Cornelsen-Verlag

|6| Diverse Authoren:
http://www.wikipedia.org/

|7| Linear Technology:
http://www.linear.com/

|8| Patrick Schnabel:
http://www.elektronik-kompendium.de/

8 Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass ich alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken oder dem Internet entnommen sind, durch Angabe der Quellen als Entlehnung kenntlich gemacht habe. Mir ist bewusst, dass Plagiate als Täuschungsversuch gewertet werden und im Wiederholungsfall zum Verlust der Prüfungsberechtigung führen können.

Ort, Datum	Unterschrift

9 Anhang (in digitaler Form)