Dr. Ing. Wilfried Dankmeier
FH Frankfurt
University of Applied Sciences
Fachbereich Informatik und Ingenieurwissenschaften

Eppstein im Taunus, 14.12.2006

Programmiersprachen für technische Anwendungen

Notizen für ein Manuskript

Programmieren in C/C++

Stand 14.12.2006

Inhalt:

1. Was ist Programmieren?
2. Welche Programmiersprachen gibt es?
3. Wo ist der Nutzen für den Ingenieur?
4. Warum C?
5. Organisations-Hinweise (Vorlesungen, Übungen)
6. Literatur/Links
7. Grundregeln
8. Organisations-Hinweise (Klausur)
9. Es geht los – die Entwicklungsumgebung Microsoft Visual.Net
10. Ein Quellprogramm
11. Einige Einzelheiten
12. Hinweise zur Simulation dynamischer Systeme
13. Anhang

Hinweis: Manches in den Beispielen wäre mit „Objekt-orientierten“ Elementen von C++ eleganter zu lösen. Da die entsprechenden Begriffe aber zusätzlich erarbeitet werden müssen, wird zunächst darauf verzichtet (um es in einem nächsten Schritt zu ergänzen).

1. Was ist Programmieren?

... einen Rechner so mit Maschinenbefehlen zu versehen, dass durch den Ablauf dieser Befehle eine vom Programmierer beabsichtigte Aufgabe bearbeitet wird (z. B. die Lösung einer Gleichung, die Erzeugung einer Tabelle, die Übernahme von Tastatureingaben und deren grafische Darstellung, die Regelung einer Raumtemperatur, die Erzeugung von Musik, die Kompression von Bilddaten, die Verwaltung von Warenlagerbeständen ...)

2. Welche Programmiersprachen gibt es?

... viele ...

- die einfachsten sind die **Maschinensprachen** der Prozessoren (CPUs), im primitivsten Fall gibt man sie als HEX-codierte Zahlen über eine kleine Tastatur direkt in den Arbeitsspeicher des Prozessors ein. Nachteil: bei Veränderungen muss meistens der gesamte Programmcode neu eingegeben werden.
- mehr Komfort bieten **Assembler**, da sie dem Programmierer zeitraubende Arbeiten beim Erstellen und Ändern des Programmcodes abnehmen (z. B. werden Speicheradressen als symbolische Namen angegeben, bei Änderungen wird die Neuberechnung aller davon abhängigen numerischen Maschinenadressen vom Assembler erledigt)
- so genannte „**Hochsprachen**“ wie Fortran, Pascal, Visual Basic, Mathematica stellen sehr mächtige Programmierbefehle und Programmierstrukturen bereit, die den Programmierer bei komplexen Aufgaben entlasten. Die Nähe zur Maschinensprache ist aber nicht mehr gegeben (und oft auch nicht erforderlich). Häufig sind diese Sprachen zudem nur mit hohem Aufwand auf verschiedene Rechnersysteme anpassbar.
- zwischen diesen Hochsprachen und den Maschinensprachen gibt es **maschinennahe Sprachen**, die aber zugleich auch mächtige Befehle bereitstellen, wie z. B. C, C++. Mit diesen kann man Programme sowohl für komplexe Aufgabenstellungen als auch für Prozessor-orientierte Probleme entwickeln. z. B. wird zu CPUs heute meistens auch eine Maschinensprache-Entwicklungsumgebung angeboten, die C-Compiler enthält. Standard-Compiler für C lassen sich wegen der einfachen Struktur ohne großen Aufwand auf verschiedenen CPUs „portieren“.

3. Wo ist der Nutzen für den Ingenieur?

Ingenieure müssen oft komplexe technische Aufgabenstellungen bearbeiten, die eine Vielzahl von Rechnungen und die Verarbeitung umfangreicher Datenmengen erfordern. Die Zuhilfenahme von Programmen ist in vielen Fällen der einzige Weg, um sich – z. B. über so genannte **Simulationen** – Lösungen zu beschaffen. Außerdem können die Ergebnisse der Programmläufe wesentliche Hilfen für das Verständnis komplexer technischer Systeme liefern (z. B. zum Verhalten der Fahrzeugräder beim Bremsen mit ABS-Unterstützung, zur Wirksamkeit eines Fehlerkorrekturverfahrens bei der Übertragung von Daten auf „verrauschten“ Kanälen, und unendlich vieles mehr....)

4. Warum C?

- einfache Sprache mit Maschinen- und Hochsprachenelementen, daher gut geeignet für ingenieurtechnische Einsätze
- die Sprache, in der die UNIX- und Windows-Betriebssysteme geschrieben sind
- weit verbreitet, daher gut unterstützt
- robust, hat schon einige andere Sprachen überlebt

Als moderne Erweiterung der „alten“ Sprache **C** setzt sich heute immer stärker **C++** durch. Da diese Variante die Elemente von C (den ANSI-Standard) mit enthält und zugleich den später einmal wünschenswerten Zugriff auf weitere nützliche Leistungen eröffnet, werden wir gleich bei **C++** einsteigen. Dabei beschränken wir uns aber vorerst auf den C-Anteil.

5. Organisatorische Hinweise (Vorlesung, Übung)

(Nach Bedarf)

6. Literatur/Links

Im Internet findet man – wie heute üblich - eine Fülle von Literatur und komplette Entwicklungsumgebungen (IDE = Integrated Develop Environment) als Freeware. Man erliegt schnell dem trügerischen Eindruck, dass ja bereits alles zum Erlernen Nötige vorbereitet wäre und mit dem Herunterladen die Hauptarbeit getan ist. **Falsch!** Leider besteht wie immer der Löwenanteil darin, selbst zu lernen und zu üben.

Einen für den allerersten Einstieg geeigneten Kurzüberblick gibt zum Beispiel *Stefan Thiemert* in www.c-programme.de. Tiefer führt das Buch **Das Einsteigerseminar C++** von *Alexander Niemann/ Stefan Heitsiek*, es ist einfach geschrieben und mit ca. 11 Euro in der Anschaffung attraktiv.

Eine umfassende systematische Darstellung gibt *Dietrich May* im **Grundkurs Software-Entwicklung mit C++** , ISBN 3-528-05859-5 (29,90 Euro), **sehr zu empfehlen** auch *Dietmar Herrmann*, **Grundkurs C++ in Beispielen**, ISBN 3-528-54655-7 (29,90 Euro), wo man **eine Menge** Anregungen und Lösungswege findet.

7. Grundregeln

... üben, üben, üben...
... Aufgaben in Teile zerlegen
... gut ist gut genug (80/20-Regel)
... niemals aufgeben (... es gibt immer eine Lösung)

8. Organisatorische Hinweise (Klausur)

(Nach Bedarf)

9. Es geht los – die Entwicklungsumgebung Microsoft Visual.-Net

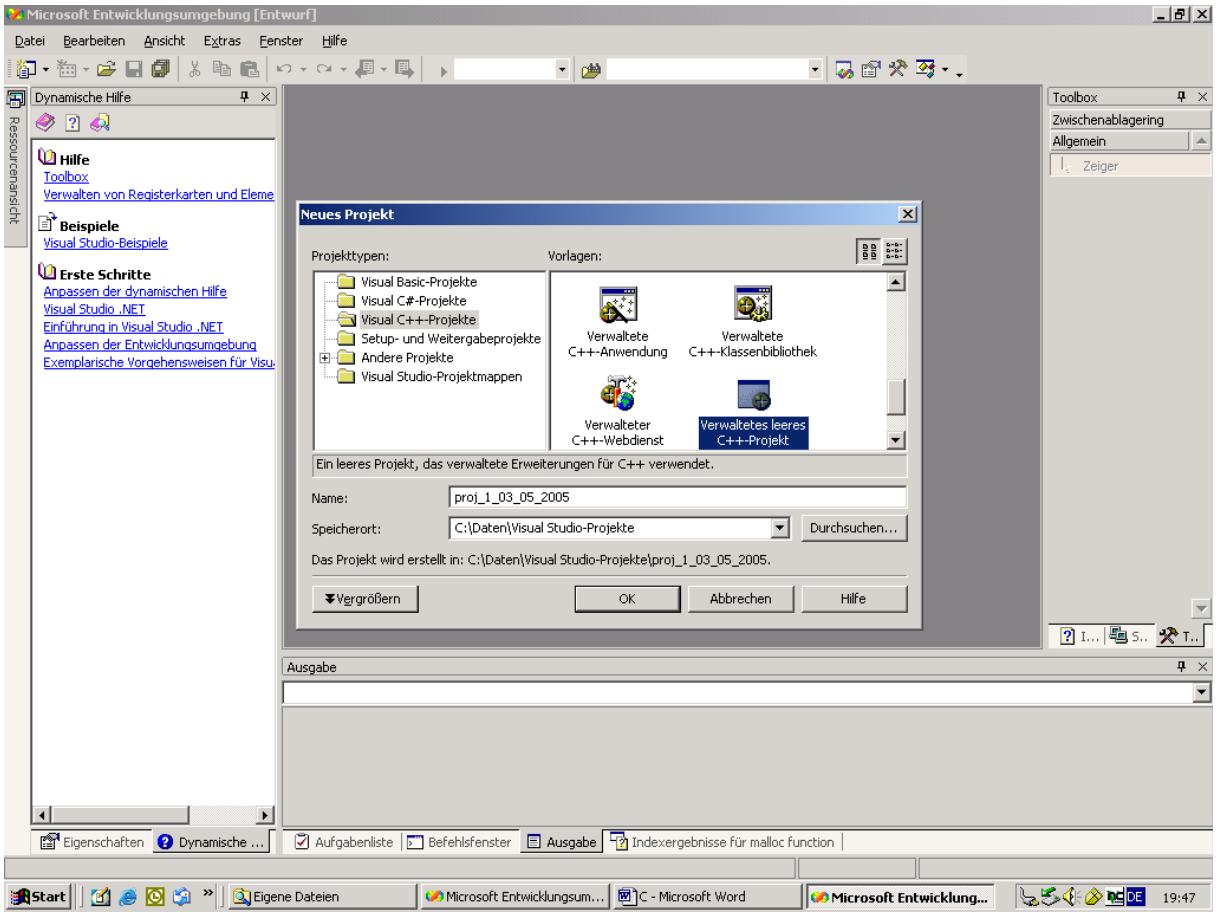
Zuerst brauchen wir einen **Arbeitsplatz**, wo wir unsere Programmiergedanken in einer weiterverwendbaren Form aufschreiben und mit einem passenden **Werkzeugsatz** in ausführbare Programme umsetzen können. Für diesen Arbeitsplatz, auch „C-Entwicklungsumgebung“ oder IDE genannt, gibt es verschiedene Ausführungen. Einige kann man sich als Freeware kostenlos im Internet besorgen, z. B. **lcc32** (besonders für die Programmierung von Windows-Anwendungen) oder **Dev C++** (Bloodshed) für ganz allgemeine Zwecke, beide für den privaten Gebrauch zu empfehlen. Wir verwenden das im PC-Labor der Elektrotechnik, Raum 204, Gebäude 6 des Fachbereichs 2 vorhandene **Microsoft Visual.Net,** ein mächtiges Tool, mit dem man Anwendungsprogramme in Teams entwickeln kann, die über den ganzen Erdball verteilt angesiedelt sein können und – verbunden durch das Internet – an gemeinsamen Programmierprojekten arbeiten.

Diese Mächtigkeit benötigen wir zwar noch nicht, können aber die für uns in dieser Entwicklungsumgebung interessante Teilmenge von C++ gut nutzen. Genau genommen brauchen wir von C++, der objektorientierten Erweiterung der Sprache C, vorerst nur wenige Elemente. Da der **Standardbefehlssatz** von C im Sprachumfang von C++ als Untermenge enthalten ist, stört dies aber nicht.

Wir stellen jetzt also diesen Arbeitsplatz bereit, in dem wir die Entwicklungsumgebung aufrufen. Dazu sind folgende Schritte notwendig (**Hinweis:** Die Arbeit mit **Dev C++** ist sehr ähnlich):

9.1. Unter „**Start Programme**“ das Paket **Microsoft Visual Studio.NET** durch Doppelklick starten

9.2. Unter **Datei Neu Projekt** eine neues „**Verwaltetes leeres C++ Projekt**“ anlegen. Dazu muß ein Projektname angegeben werden. Man sollte ihn so wählen, dass die Projekte geordnet wieder auffindbar sind, z. B. als **proj_<Nummer> <Datum>**, etwa **proj_1_03_05_2004:**



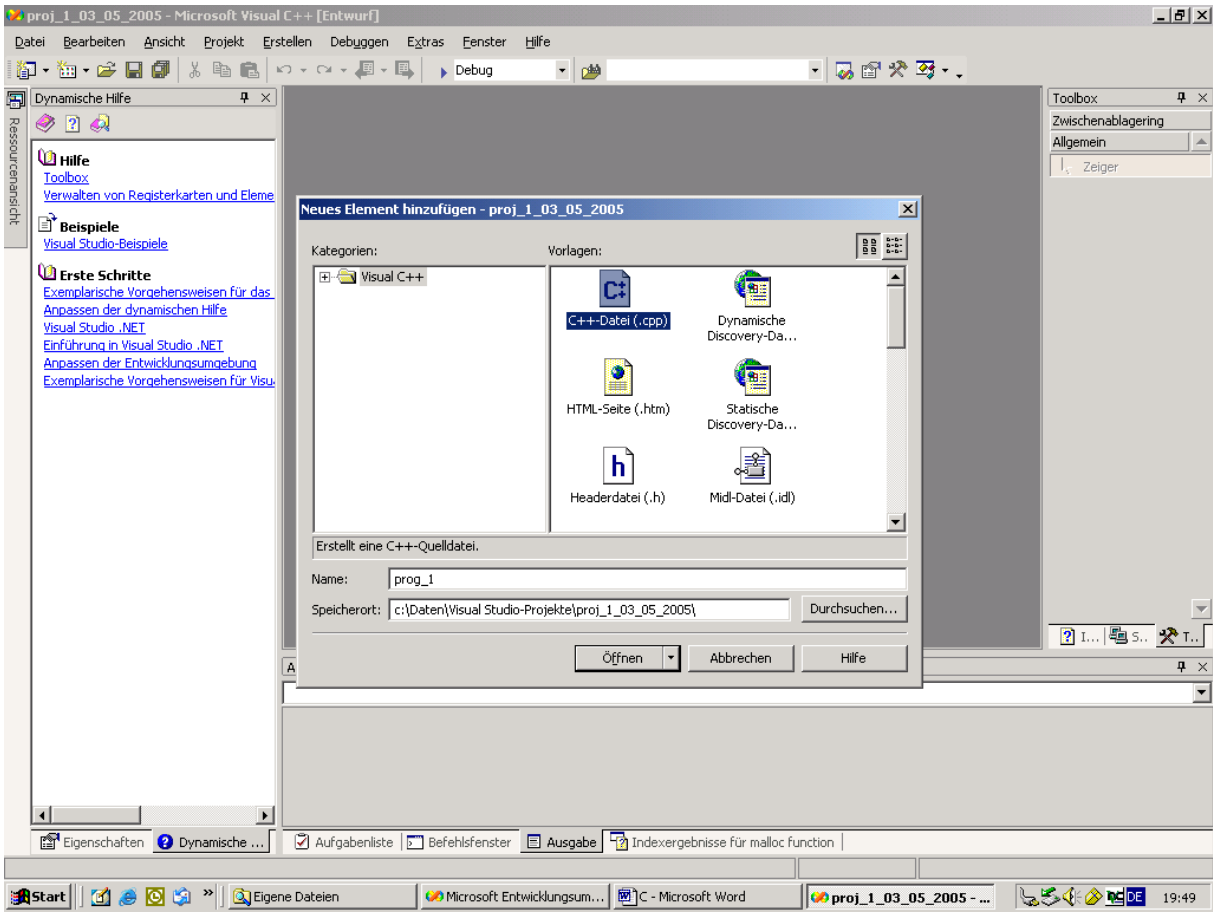
Wie man sieht, bietet **Microsoft Visual Studio.NET** auch weitere Entwicklungsumgebungen, z. B. für **Visual Basic**, **Visual C#** und weitere. Auch unter der von uns gewählten **Visual C++-Projekte** gibt es verschiedene Ausführungen im rechten Auswahlfenster, der Typ **Verwaltetes leeres C++-Projekt** ist für unsere Zwecke im Moment aber das geeignetste.

9.3. Nun fügen wir mit einem beliebigen Namen, z. B. als **prog_1**, eine leere .cpp-Datei hinzu und erhalten ein leeres Editier-Fenster (diese Datei hat also den vollständigen Namen **prog_1.cpp**). Hier kann der so genannte Quellcode eingetragen werden.

Warum brauchen wir diesen Quellcode? Der **Prozessor** (die CPU = Central Processing Unit) des Rechners „versteht“ als Sprache nur Binärfolgen in Einheiten von Bytes oder Vielfachen davon. Diese Folgen holt er sich aus dem **Arbeitsspeicher** (RAM = Random Access Memory). Eine solche Folge könnte etwa so aussehen:

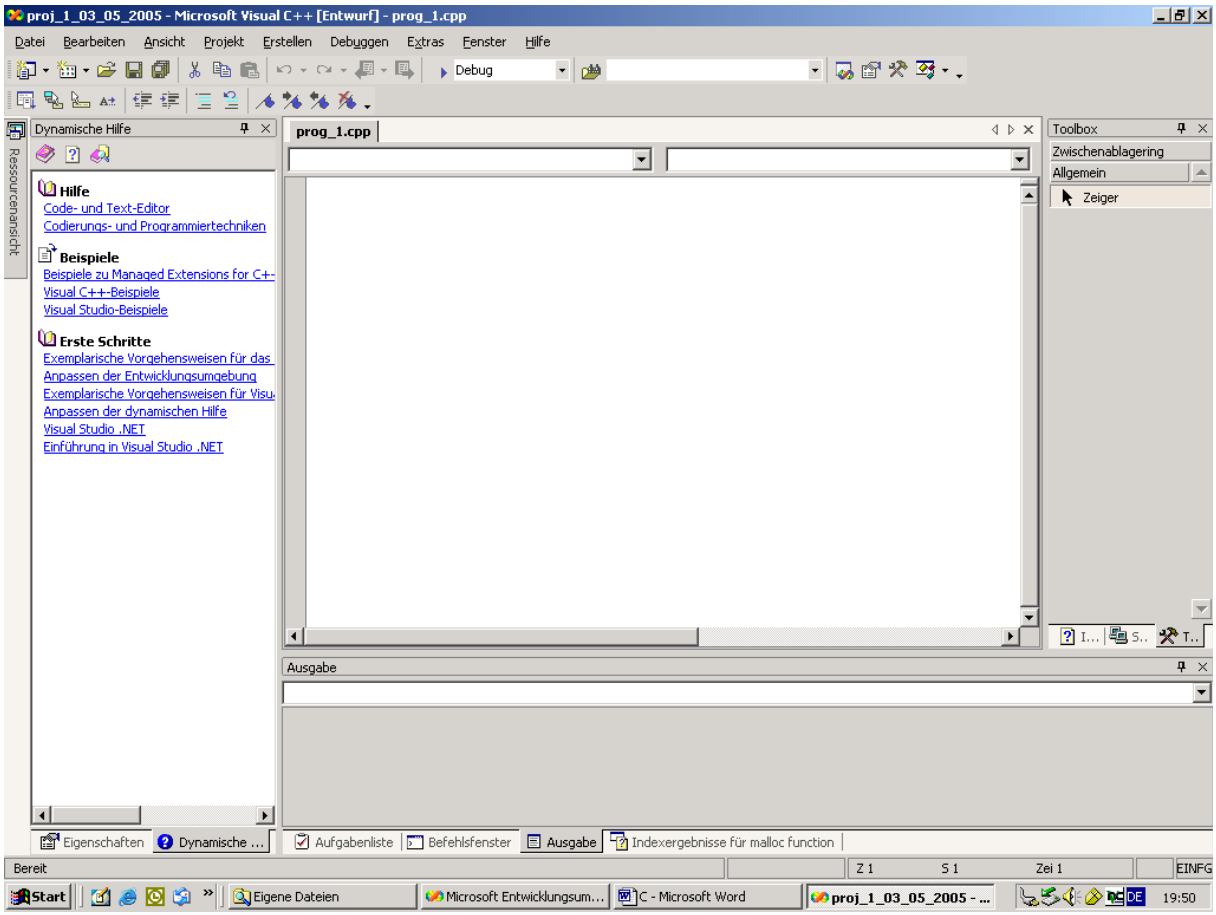
Speicherzelle n - 1								Speicherzelle n								Speicherelle n + 1								Speicherzelle n + 2							
0	1	1	1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	1	1	1	1	1	0	1	0	0	0

Für uns Menschen ist es mühsam, unsere Wünsche an den Rechner als Binärfolgen in den Arbeitsspeicher einzugeben. Einfacher wird es, den Programmablauf in einer uns besser verständlichen Form in einem Editor aufzuschreiben und dem Compiler die Übersetzung in die Binärfolgen zu überlassen.



9.4. Nach **Öffnen** zeigt sich ein leeres Fenster als Eingabefläche, auf der wir unseren Programmcode (= Quellcode) eingeben. Eigentlich haben wir damit einen speziellen Editor geöffnet. Wir können auch einen anderen Editor benutzen, z. B. den normalen **Microsoft Editor** aus dem „**Zubehör**“ oder auch „**Word**“, aber dieser hier hat zugleich einige hilfreiche Funktionen, die das Eingeben erleichtert.

So werden die **Standard-Programmierbefehle** (= Anweisungen = Statements) in **blauer Schrift** angegeben, **Kommentare**, die man sich als rein textliche Erläuterungen ohne Auswirkungen auf das Programm hinzufügen kann, sind **grün** gefärbt. Die Tabulatoren werden entsprechend der Programmstruktur gesetzt, so dass die Übersicht über den Programmaufbau verbessert wird usw.



10. Ein Quellprogramm

Um den Editor auszuprobieren, programmieren wir die Berechnung einer Quadratzahl. Das Programm selbst besteht aus dem Hauptteil **main** (genau genommen eine Funktion) und vorangestellten **Präprozessor-Direktiven,** z. B. **#include <stdio.h>**, die dem Compiler sagen, was er alles beachten und hinzufügen muss, wenn er aus dem Quellprogramm ein ausführbares Objektprogramm (das eigentliche Maschinenprogramm, was die CPU ausführen kann → Binärfolgen) machen soll.

Damit ist eine oft zu verwendende Grundstruktur schon gegeben:

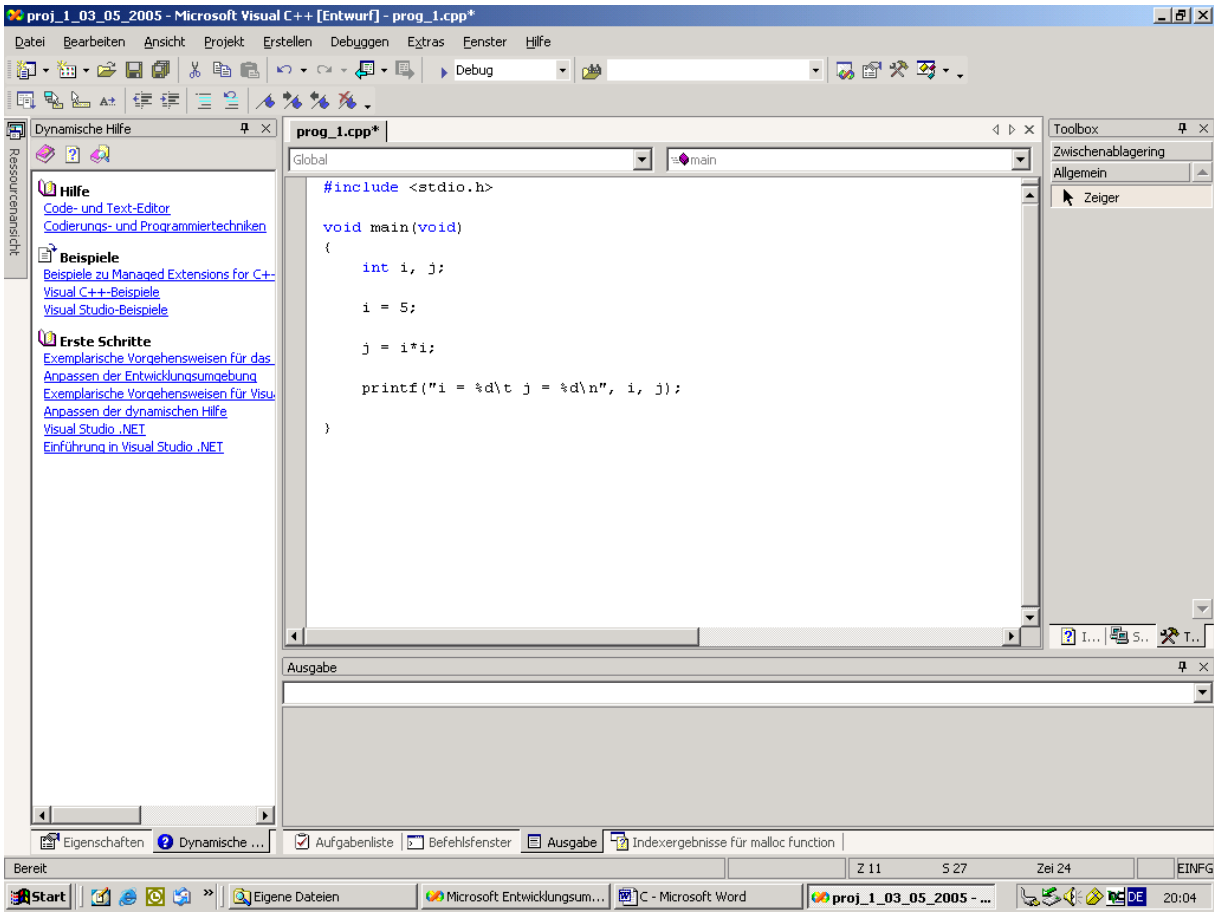
```
#include   <stdio.h>      /* Kommentar 1: Präprozessor-Direktive, die die
                           Programmbibliothek für Ausgabefunktionen, z. B.
                           auf den Bildschirm, enthält */

void main (void)
{
      // Kommentar 2: hier stehen die Anweisungen des individuellen Pro-
                  gramms

}
```

Kommentare können auf zwei verschiedene Arten gekennzeichnet werden:

- /* Kommentar 1 mit **Slash Stern** gilt unabhängig von der dazwischen liegenden Zeilenzahl bis zum Abschlusszeichen **Stern Slash**: */

- // Kommentar 2 mit zwei **Slash's** gilt nur in dieser Zeile

Das Quellprogramm mit dem Namen **prog_1.cpp** für unsere kleine Aufgabe könnte so aussehen:



Es verweist mit **#include** <stdio.h> auf die (Programm-) Bibliothek, welche die Standard-Ein/Ausgabe-Funktionen (**stdio** = standard input/output) wie **printf** (formatiertes „Drucken" auf den Bildschirm) enthält und damit in das Programm einbezogen wird.

Auf den **Haupt**-Programmnamen **main** folgt in geschweiften Klammern { } der Block, welcher das eigentliche, individuelle Programm enthält. Die Bezeichnung **void** (= leer) besagt nur, dass dem Programm kein Typ zugeordnet ist, d. h. **main** erhält keinen Wert und nimmt wegen **main** (void) auch keine Parameter entgegen, die innerhalb von **main** weiter zu verwenden wären (es gibt noch andere Formen, wo das nicht so ist).

int ist eine Typ-Deklaration und besagt, dass die Variablen mit den Namen i und j als **Ganzzahlen** zu verwenden sind. Der Variable i wird mit

i = 5;

der Zahlenwert 5 zugewiesen, die Variable j erhält durch

j = i * i;

den Wert 25. Dieser wird schließlich mit

```
printf ("i = %d\t j = \n", i,  j );
```

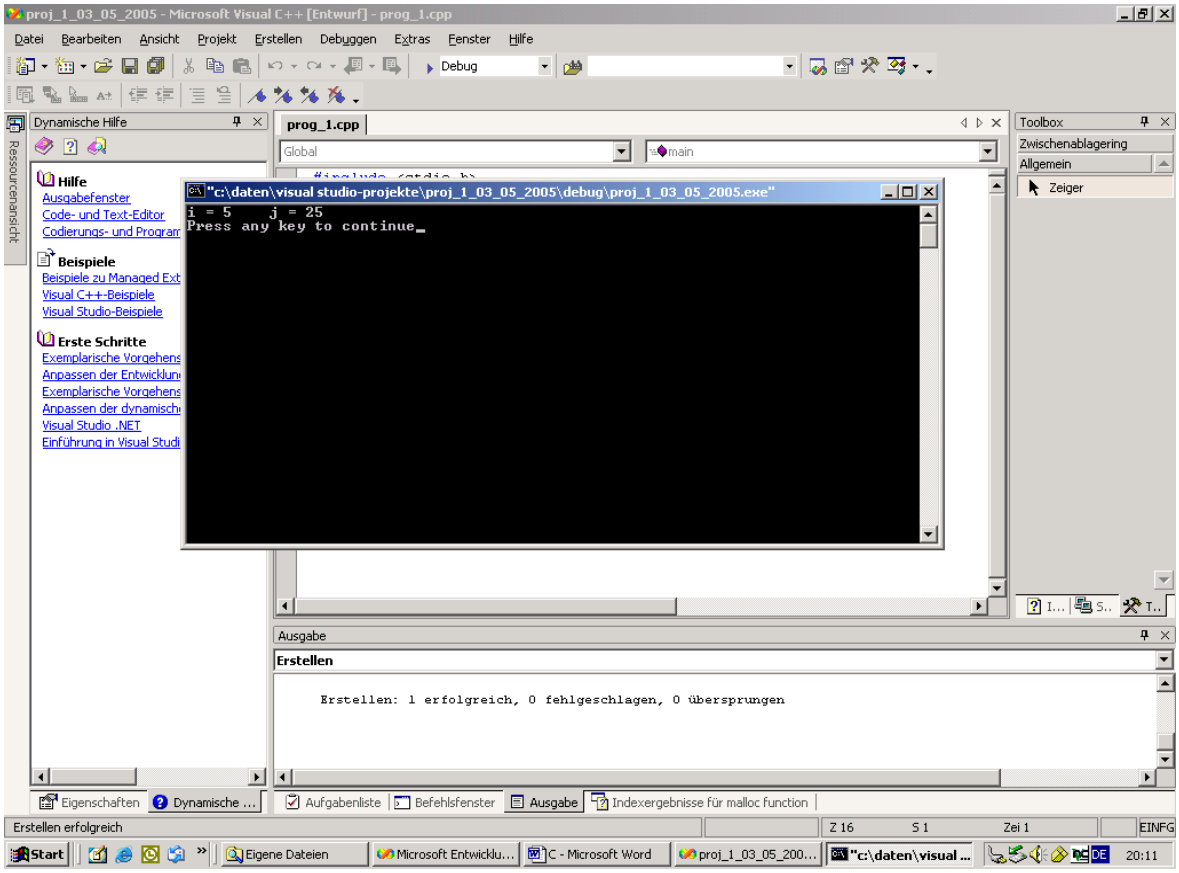
auf dem Bildschirm ausgegeben. **printf** enthält zwei Parameter:

- Die zweiten und dritten Parameter sind die Variablen i und j, deren Werte auszugeben sind

- Der erste Parameter steht in Anführungszeichen und beschreibt den Aufbau des Ausgabeformats. Das Prozentzeichen % zeigt an, dass eine spezielle Formatangabe folgt. Das Zeichen d heißt, dass der erste Wert nach dem Komma (hier die Variable i) als Dezimalzahl auszugeben ist. Die Kombination **\t** ist eine so genannte Escape-Sequenz und veranlasst nach der Ausgabe von i einen Tabulatorsprung, **\n** ein „Return“ (n = newline) auf dem Bildschirm.

  Es gibt einige weitere Elemente zum formatierten Ausgeben, womit sich die Übersichtlichkeit erhalten lässt. **% 6d** etwa besagt, dass der Zahlenwert mit 6 Stellen und einer Vorzeichenstelle erscheint, daher ist zwischen % und 6 ein Leerschritt eingefügt. **%f** erzeugt ein Gleitkommaformat, **% 8.2f** die Ausgabe eines Wertes in Gleitkommadarstellung mit Vorzeichen, insgesamt 8 Stellen, davon 2 Nachkommastellen. Mehr dazu im Anhang A3.

Das Quellprogramm müssen wir über den Menü-Punkt **Debuggen Starten ohne Debuggen** vom Compiler übersetzen und vom Linker mit weiteren erforderlichen Programmteilen (z. B. mit den in **stdio.h** enthaltenen) ergänzen lassen, so dass das ausführbares Maschinenprogramm **proj_1_03_05_2004.exe** entsteht. Es liefert nach dem Starten das Ergebnis:



Es kann auch direkt aus der Explorer-Umgebung oder unter **Start Ausführen** aufgerufen werden. Allerdings schließt sich nach Ausführung ohne weitere Maßnahmen das DOS-Fenster sofort wieder, so dass man

das Ergebnis im Gegensatz zum Aufruf aus der Entwicklungsumgebung nur sehr kurz betrachten kann. Bei anderen Entwicklungsumgebungen, z. B. Bloodshed, hilft hier als letzte Programmanweisung der Befehl **system ("Pause");**

Tip: Das Ausgabefenster lässt sich mit der Tastenkombination ALT RETURN zwischen Vollbild- und verkleinertem Bild hin und her schalten.

11. Einige Einzelheiten

Nach diesem ersten Programmierversuch werden wir jetzt die Sprache C etwas systematischer betrachten und uns dabei mit einigen Grundelementen vertraut machen, mit denen wir schon eine ganze Menge an Ingenieur-technischen Aufgaben bearbeiten können. Tabellarische Aufstellungen sind im Anhang zu finden.

Details werden wir uns vorzugsweise über die **Hilfe**-Funktion der Entwicklungsumgebung beschaffen. Im Hilfe-Menü lässt sich wie üblich über **Index** ein Suchbegriff angeben, zu dem neben den gewünschten Erläuterungen häufig auch Beispiele des Gebrauchs zu finden sind.

11.1. Typ-Deklaration und Namen (Bezeichner) von Variablen

Bevor eine Variable verwendet werden kann, muss ihr Typ festgelegt und ein Name zugewiesen werden. Mit dem Typ ist zugleich ein Speicherplatz in Byte reserviert:

-	Ganzzahl-Typ:	`int`	4 Byte
-	Ganzzahl-Typ kurz:	`short`	2 Byte
-	Gleitkommazahlen-Typ:	`float`	4 Byte
-	Gleitkommazahlen Typ mit doppelter Genauigkeit	`double`	8 Byte
-	Zeichen-Typ:	`char`	1 Byte
-	Logik-Typ:	`bool`	1 Byte

Als Variablennamen sind alle zugelassen, die sich aus

- Gross/Klein-Buchstaben
- Ziffern
- dem Zeichen Underscore (" _ ")

zusammensetzen. **Achtung:** Gross/Kleinschreibung erzeugt Unterschiede! Insgesamt empfiehlt es sich, diese Flexibilität zu nutzen und **sprechende** Namen zu verwenden, auch dann, wenn sie lang werden. Es ist bei der Programmentwicklung vielleicht lästig, nach längerer Unterbrechung bei der Arbeit am Programm hat man aber einen besseren Bezug auf die Bedeutung der Variablen als bei abstrakten Namen. Nicht zulässig sind allerdings vom Compiler reservierte Schlüsselwörter, z. B. `void` usw., siehe Anhang A4.

Die Deklarationen werden wie alle Anweisungen immer mit Semikolon abgeschlossen. Beispiele:

```
void main(void)
{     int   i, j, k, anfangsindex, endindex;
      float x, y, R1, R2, mittelwert, effektivwert;
      double xd;
      char  a, zeichen;
      bool  wahrheitswert;  // 1 (oder true) bzw. 0 (oder false)

}
```

11.2. Ausgabeformatierung

Der in **printf** verwendbare String lässt sich sowohl zur Ausgabe beliebiger Zeichenketten (z. B. für Variablenbezeichnungen, Überschriften usw.) als auch zur Festlegung bestimmter Darstellungsformate von Zahlen und Zeichen verwenden. Die CPU bietet dabei immer nur Binärmuster als Abbilder der Speicherzellen an, die durch Formatanweisungen in einer uns passenden Darstellung gezeigt werden. Einige sind im Anhang A3 zu sehen.

11.3. Anweisungen (Kontrollstrukturen)

Anweisungen steuern den Programmablauf. **Funktionen** bestimmen Werte, verändern Objekte.

11.3.1 Schleifen: for | while | do while

Ein Beispiel für eine Anweisung ist die bestimmte Schleife **for** . Mit ihrer Hilfe können Programmteile mehrfach ausgeführt werden. Wie oft das passiert, wird durch Parameter bestimmt. Will man z. B. eine Tabelle der Quadratzahlen von 10 bis 25 erstellen, so kann man das Programm aus Kapitel 10 verwenden und mit einer passenden **for**-Schleife erweitern. Die Variable j wird hier nicht mehr benötigt, außerdem setzen wir eine Tabellenüberschrift in die Ausgabe:

```
#include <stdio.h>

void main(void)
{
      int i;

      printf("Tabelle der Quadratzahlen von 10 bis 25:\n\n");

      for(i=10;i<26;i=i + 1) printf("i = %d\t i*i = %d\n", i, i*i);
}
```

Mit i = 10 wird gestartet, es wird geprüft, ob i kleiner als 26 ist, dann erfolgt die Ausgabe mit **printf** , anschließend wird i um den Wert 1 erhöht und die Schleife erneut durchlaufen. Nach dem 26. Durchgang (warum 26 ?) hat i den Wert 26, die „<“- Bedingung ist nicht mehr erfüllt, die Schleife ist beendet. Ergebnis:

```
Tabelle der Quadratzahlen von 10 bis 25:

i = 10   i*i = 100
i = 11   i*i = 121
i = 12   i*i = 144
i = 13   i*i = 169
i = 14   i*i = 196
i = 15   i*i = 225
i = 16   i*i = 256
i = 17   i*i = 289
i = 18   i*i = 324
i = 19   i*i = 361
i = 20   i*i = 400
i = 21   i*i = 441
i = 22   i*i = 484
i = 23   i*i = 529
i = 24   i*i = 576
i = 25   i*i = 625
Press any key to continue
```

Die Bedingungen werden mit den Logik-Operatoren gemäß Anhang A2 gebildet.

Mit der **for**-Anweisung kann eine feste Zahl von Anweisungsdurchläufen erreicht werden. Es gibt aber Anforderungen, bei denen die Zahl dieser Durchläufe vom Erreichen eines bestimmten Ergebnisses innerhalb der Anweisung oder innerhalb des Anweisungsblocks abhängen soll. Dabei treten zwei Fälle auf:

Fall a: Die Bedingung wird **vor** Ausführung der Anweisung geprüft, die Anweisung wird also eventuell überhaupt nicht durchlaufen

```
#include <stdio.h>

void main(void)
{
	int i;
	i = 10;
	printf("Tabelle der Quadratzahlen von 10 bis 25:\n\n");
	while (i < 26)

	{
		printf("i = %d\t i*i = %d\n", i, i*i);
		i = i + 1;
	}
}
```

Die Ausgabe ist hier identisch zu der bei der **for**-Schleife (bitte nachprüfen). Der Nutzen zeigt sich erst bei anderen Fällen. Nehmen wir an, wir wollen eine Tabelle der Exponentialfunktion (der so genannten e-Funktion, siehe Anhang A5) berechnen. Diese Funktion hat den Namen `exp` und ein Argument, z. B. x. Wir möchten von x = 0 beginnend das Argument x immer um 0.1 erhöhen, benötigen aber nur Funktionswerte bis etwa `exp(x) = 3`. Allerdings wissen wir nicht, für welches Argument x der Funktionswert `exp(x) = 3` erreicht wird. Daher wird in die Bedingung der **while**-Schleife die Abfrage auf exp(x) < 3 aufgenommen:

```
#include <stdio.h>

void main(void)
{
	int i;
	float x, dx, y;

	x = 0 ;
	dx = 0.1 ;
	y = exp(0);
	i = 0;

	printf("Tabelle der Exponentialfunktion für exp(x) = 1 \n\n");

	while (y < 3)
	{
		printf("x = %f\t exp(x) = %f\n", x, exp(x));
		x = x + dx;
		y = exp(x);
	}
}
```

Fall b: Die Bedingung wird **nach** der Ausführung der Anweisung geprüft, die Anweisung wird also wenigstens einmal durchlaufen:

```
#include <stdio.h>

void main(void)
{
      int i;
      i = 10;
      printf("Tabelle der Quadratzahlen von 10 bis 25:\n\n");
      do
      {
            printf("i = %d\t i*i = %d\n", i, i*i);
            i = i + 1;
      }
      while (i < 26);
}
```

Auch hier erhalten wir eine Ausgabe, die identisch zu der bei der **for**-Schleife ist. Nützlich sind die Kontrollstrukturen mit **while** und **do- while** zum Beispiel bei Iterationen, also immer dann, wenn man die Anzahl der Schritte bis zum „nicht mehr erfüllt“ der Bedingung nicht vorhersehen kann.

11.3.2 Verzweigungen: if | if else | if else if | switch case | switch case break

Wenn Anweisungen abhängig von Bedingungen durchlaufen werden sollen oder nicht, verwenden wir Verzweigungen. Der Grundtyp ist die implizit bereits bei allen Schleifen-Anweisungen enthaltene **if**-Anweisung:

```
#include <stdio.h>

void main(void)
{
      int i;
      i = 8;
      if (i < 10) printf("i ist kleiner als 10  %d\n", i);
}
```

Müssen zwei Fälle unterschieden werden, so hilft die Ergänzung mit **else**:

```
#include <stdio.h>

void main(void)
{
      int i;
      i = 16;
      if (i < 10) printf("i ist kleiner als 10  %d\n", i);

      else printf("i ist grösser als 10  %d\n", i);
}
```

Bei mehr als zwei zu unterscheidenden Fällen verwendet man **else if**:

```
#include <stdio.h>

void main(void)
```

```
{
      int i;
      i = 16;
      if (i < 10) printf("i ist kleiner als 10  %d\n", i);

      else if ((i >= 10) && (i < 20)
      printf("i ist grösser/gleich 10, aber kleiner 20  %d\n", i);

      else  printf("i ist grösser/gleich 20  %d\n", i);

}
```

Konstruktionen mit vielen **`if - else if`** Verzweigungen können unübersichtlich werden. Dann ist der „Schalter“ **`switch`** mit den Fällen **`case`** abgebracht:

```
#include <stdio.h>

void main(void)
{
      int fall;
      fall = 3;

      switch(fall)
      {
            case 1: printf("Fall 1");
            case 2: printf("Fall 2");
            case 3: printf("Fall 3");
            case 4: printf("Fall 4");
      }
}
```

11.4. Funktionen

Wenn in einem Programm mehrfach die gleichen Berechnungen, aber mit unterschiedlichen Eingabewerten gemacht werden sollen, ist es angebracht, den Berechnungsablauf in einem eigenen „Unterprogramm“ aufzuschreiben und dieses als Funktion zu deklarieren. Der Name der Funktion wird nach denselben Regeln vergeben wie bei einfachen Variablen. Die Deklaration erfolgt vor der Funktion **main**. Die Parameter, die der Funktion übergeben werden, stehen mit ihrer Typdeklaration einzeln in der Klammer nach dem Funktionsnamen.

Beispiel: die Funktion **fquadrat** soll aus x, a, b und c den „Parabel“-Wert `y = a*x*x + b*x + c` berechnen:

```
#include <stdio.h>
#define _USE_MATH_DEFINES /* Präprozessor-Direktive, die vor <math.h> stehen
muß und den Zugriff auf bereits definierte Konstanten wie M_PI =3.1415...
gestattet, mehr findet man z.B im Quelltext der <math.h>-Bibliothek */
#include <math.h>

float fquadrat(float x, float a, float b, float c)
{     float yh;
      yh=a*x*x + b*x + c;
      return(yh);
}

void main(void)
{
      double x, y, a, b, c;
```

```
        x = 2 ;
        a = 1 ;
        b = 1 ;
        c = 1 ;

        y = fquadrat(xh, a, b, c);
        printf("fquadrat(% 4.2f) = %4.2f\n\n",x,y);

}
```

Die Funktion ist vom Typ `float` und gibt über die Anweisung `return(yh)` das Ergebnis an den Namen **fquadrat** zurück. Die mathematischen Funktionen in Anhang A5 sind ebenfalls einfache Beispiele.

11.5. Zeiger

Zeiger (englisch pointer) haben in C eine besondere Bedeutung. Sie sind Variable, enthalten aber gegenüber den bereits bekannten allgemeinen Variablen als Werte „nur“ deren Speicheradressen. Damit lassen sich einerseits Programmierstrukturen aufbauen, die ohne dieses Werkzeug nicht möglich wären oder mit mehr Aufwand realisiert werden müssten (z. B. die dynamische Felddeklaration, siehe Abschnitt 11.6), andererseits kann man wegen der direkten Arbeit mit Adressen wesentliche Funktionalitäten der Prozessor-Maschinensprachen nutzen. Allerdings erfordert der Gebrauch gerade deshalb etwas Sorgfalt, weil man sehr direkt in die Hardware-Ebene eingreift.

Die Deklaration erfolgt wie bei allen Variablen mit der Angabe des Typs, vor dem Namen steht ein “*“ (Stern). Der Typ muss immer dem Typ derjenigen Variablen entsprechen, auf den der Zeiger zeigen soll. Ein Beispiel:

```
#include <stdio.h>

void main(void)
{
      int i, ifeld[10],j,*zeiger;
      i=2307;
      zeiger=&i;  //die zeigervariable zeiger erhält den Wert der Adresse
                  //der Variablen i
      printf("%d\t%d\t%p\n",i, zeiger, zeiger); //Ausgabe des Inhalts der
                                                //Zeigervariablen zeiger
                                                //in dezimaler und
                                                //- mit %p - in Hex-Form

      *zeiger=*zeiger-13;     //der Wert der Variablen mit der Adresse zeiger
                              //wird um 13 vermindert
      printf("%d\n\n",i);

      for (j=0;j<10;j++) ifeld[j]=j*100;

      zeiger=&ifeld[0]; //die zeigervariable erhält den Wert der
                        //Adresse der ersten Feldvariablen

      printf("%d\t%d\t%p\n",ifeld[0],zeiger, zeiger);

      zeiger=&ifeld[1]; //die zeigervariable erhält den Wert
                        //der Adresse der zweiten Feldvariablen

      printf("%d\t%d\t%p\n",ifeld[1],zeiger,zeiger);

      printf("%d\t%d\t%d\n",ifeld[1],*zeiger,*zeiger*12);   //der Wert der
                                                            //zweiten Feld-
                                                            //variablen wird
                                                            //mit 12
                                                            //multipliziert
```

```
}
```

Hier das Ergebnis des Programmlaufs, die Zuordnung der Werte entnehmen wir dem Programmaufbau:

```
2307    1243036  0012F79C
2294

0       1242996  0012F774
100     1243000  0012F778
100     100      1200
```

Es lassen sich also zwei Operationen durchführen:

- Stellt man der Variablen das „&“-Zeichen voran, wird der Wert der Adresse übergeben, hier z. B. bei der Anweisung

  ```
  zeiger = &i;
  ```

- Stellt man, nachdem die Variablenadresse an **`zeiger`** übergeben wurde, der Zeigervariablen das Zeichen „*“ voran, so können damit Rechenoperationen an der Variablen ausgeführt werden, wie an dieser selbst, hier z. B.

  ```
  *zeiger = *zeiger – 13;
  ```

  Dies ist gleichwertig zu

  ```
  i = i –13;
  ```

Zwei Anwendungsbeispiele:

- Hat man in einem Mikrocontrollersystem die Parameter-Übergabeadressen eines seriellen Ein/Ausgabebausteins (UART) hardwareseitig ab der Hex-Adresse 0xCA80 verdrahtet und will auf 0xCA85 den 1-Byte-Wert 128 (als Parameter für die Baudrate 128000 Bit/s) festlegen, so fügt man folgende Anweisungen in ein entsprechendes Programm ein:

  ```
  ..............
  char *zeiger;
  zeiger = =0xCA85; //0x.... ist in C die Formateinleitung für Hex-Zahlen
  *zeiger=128;
  ..............
  ```

- Oder: Einer Funktion lässt sich über die **return**-Anweisung scheinbar nur ein einfacher Werte zuweisen (Kapitel 11.4). Übergibt man aber den Zeiger, so hat man Zugriff auf alle Variablen, die hiermit verbunden sind. Beispiele finden wir im nächsten Abschnitt 11.6

11.6. Dynamische Felddeklarationen

Felder sind indizierte Variable. Sie haben einen einheitlichen Namen, werden aber durch den Index unterschieden. Die Deklaration

```
int zufall[20]
```

besagt, dass die Ganzzahlvariable **`zufall`** 20-mal vorhanden ist. Der erste Wert steht in

```
zufall[0]
```

der zweite Wert in

zufall[1] usw.

Die Indizierung beginnt immer bei 0, weshalb die Variable mit dem höchsten Index hier **zufall**[19] ist.

Allerdings kann man in die Deklaration die Feldgrösse (hier 20) nicht selbst als Variablen einbringen, z. B. mit

```
k = 20;
int zufall[k]; falsch!
```

Zwar kann k ausserhalb von **main** im Präprozessorteil definiert und mit einem Wert versehen werden, das ist aber nicht die Lösung für das angesprochene Problem. Hierfür eignet sich die Funktion **malloc** (= memory allocation), die mit der Bibliotheks-Datei **malloc.h** eingebunden wird:

```
#include <stdio.h>
#include <stdlib.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <time.h>
#include <malloc.h>


void main(void)
{
      int i,n,teilung, k, *kasten, rmax, *zufall;
      double x,xh,dx, y, dy,yh, y1,y2;
      rmax = RAND_MAX;
      printf("rmax: %d\n\n", rmax);
      srand((unsigned) time(NULL));


      printf("Gib n an:");
      scanf("%d",&n);

      printf("Gib Teilung an:");
      scanf("%d",&teilung);

      zufall=(int *) malloc (n*sizeof(int)); // dynamische Deklaration
      kasten=(int *) malloc (teilung*sizeof(int)); // dynamische Deklaration

      for (k=0;k<n;k++)
      {     zufall[k] = rand();
            if (n<=20) printf("%d\n", zufall[k]);
      }
      for(k=0;k<teilung;k++)kasten[k]=0;
      for(k=0;k<n;k++)
      kasten[(int)floor((double)zufall[k]/(double)rmax*(double)teilung)]++;

      printf("\n\nVerteilung der Zufallswerte bei Teilung = %d\n\n", teilung);
      for (k=0;k<teilung;k++)printf("%d\n",kasten[k]);

      //free(zufall); → Anweisung zur Freigabe des Speicherplatzes für zufall
      //free(kasten);

}
```

Das Programm gestattet die dynamische Deklaration eines Feldes mit n Zufallszahlen, die nach ihrer Größe in eine Anzahl (Variablenname **teilung**) von „**Kästen**“ eingeordnet und dort gezählt werden.

Die Besonderheit liegt darin, dass nicht die Felder **zufall** und **kasten** , sondern die Zeiger gleichen Namens mit davorgesetztem * (= Stern), also ***zufall** und ***kasten** deklariert werden. Die Berechnung des gesamten erforderlichen Speicherplatzes erfolgt mit Hilfe der Funktion `sizeof()`, hier also `sizeof(int)`. Eine Integer-Variable belegt 4 Byte Speicherplatz, bei n =1000 wird demnach ein Feld **zufall** mit 1000 Plätzen zu je 4 Byte angelegt. Es geht von

zufall[0] ..... bis ..... **zufall**[999]

11.7. Felder in Funktionen und Rückgabe von Feldern an Funktionsnamen

Funktionen sind allgemein gehaltene Programmstrukturen, die erst nach dem dynamischen Aufruf in einem Programm mit Variablen und Werten versorgt werden. Wenn in Funktionen als Parameter auch Felder übergeben werden, müssen diese ohne Feldgrenzen deklariert sein, etwa

```
float funk1 (int n, float x, float felda[ ] )
{
      //internes Programm von funk1
}
```

Bei Aufruf von **funk1** innerhalb von **main** wird für den formalen Parameter **felda** der aktuelle Feldname eingesetzt:

```
void main (void)
{     int i, j , k;
      float  xh, array1[10];
      k=20;
      xh=0.34;
      for (i = 0; i<10; i++) array1iI] = 0;
      funk1(k, xh,array1);
}
```

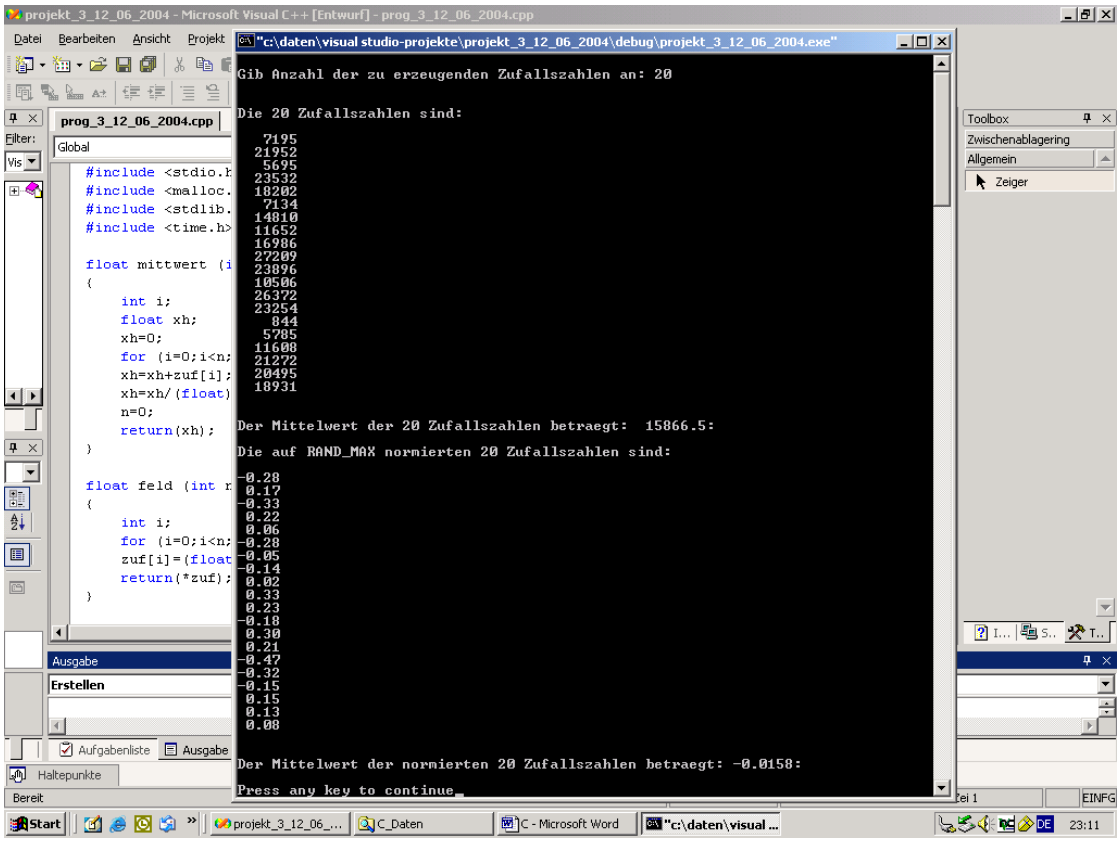
Wie wir in Abschnitt 11.4 sahen, kann über die **return**-Funktion ein Wert an die Funktion selbst zurück übergeben werden. Dies geht auch, wenn es sich um den Wert einer Feldvariablen handelt, jedoch muss dazu der zugeordnete Zeiger auf die Feldvariable eingesetzt werden, siehe Abschnitt 11.5. Das folgende Beispiel erläutert das Verfahren. Das Programm enthält

- eine Funktion **mittwert** zu Berechnung des Mittelwertes einer Anzahl n von Feldwerten **zuf[ ]**
- eine Funktion **feld** zur Normierung der **n** eingegebenen Feldwerte **zuf[]** auf den Wert **rmax** und Erzeugung einer annähernden Mittelwertfreiheit (Mittelwert » 0) sowie Rückgabe der veränderten Feldwerte an den Funktionsnamen. Die Rückgabe erfolgt über den Zeiger ***zuf**

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>

float mittwert (int n, float zuf[])
{
      int i;
      float xh;
      xh=0;
      for (i=0;i<n;i++)
      xh=xh+zuf[i];
      xh=xh/(float)n;
```

```
      n=0;
      return(xh);
}

float feld (int n, int rmax, float zuf[])//feldübergabe an Funktionsnamen
{
      int i;
      for (i=0;i<n;i++)
      zuf[i]=(float)zuf[i]/(float)rmax-0.5;
      return(*zuf);
}

void main (void)
{
      int i,n;
      float *zuf,yh;
      FILE *datei;

      datei=fopen("zufallsdaten.dat","w");

      srand((unsigned) time(NULL));

      printf("\nGib Anzahl der zu erzeugenden Zufallszahlen an: ");
      scanf("%d",&n);

      zuf=(float*) malloc(n*sizeof(float));

      printf("\n\nDie %d Zufallszahlen sind:\n\n",n);

      for(i=0;i<n;i++)
      {
            zuf[i]=rand();
            printf("%7.0f\n",zuf[i]);
      }
      yh=mittwert(n,zuf);

      printf("\n\nDer Mittelwert der %d Zufallszahlen betraegt:  %
      6.1f:\n",n,yh);

      printf("\nDie auf RAND_MAX normierten %d Zufallszahlen sind:\n\n",n);

      *zuf=feld(n,RAND_MAX,zuf);

      for(i=0;i<n;i++) printf("% 4.2f\n",zuf[i]);

      fprintf(datei,"*zuf");
      fclose(datei);

      yh=mittwert(n,zuf);

      printf("\n\nDer Mittelwert der normierten %d Zufallszahlen betraegt:
      % 6.4f:\n\n",n,yh);
}
```

Bei Eingabe von n = 20 erhalten wir:



Man erkennt hier nochmals die Vorzüge beim Gebrauch von Funktionen.

- Kleine Programmteile können für sich entwickelt und getestet werden. Wenn sie korrekt laufen, lassen sie sich wegen ihrer allgemeinen Formulierung überall verwenden, ohne erneut überprüft werden zu müssen. Damit kann man die Komplexität großer Programmstrukturen oft wesentlich entschärfen.

- Für mehrfach benötigte Programmabläufe gleicher oder ähnlicher Art genügt der jeweilige Aufruf der Funktion mit den entsprechenden Parametern, was die Übersichtlichkeit erhöht und die Fehlersuche vereinfacht. Im obigen Beispiel wird die Funktion **`mittwert`** z. B. zweimal verwendet.

- Die allgemeine Formulierung eines Programmablaufs gestattet das Austesten für kleine, überschaubare Datenmengen und den Übergang auf praktisch beliebig große ohne Verlust der Korrektheit. Im vorliegenden Beispiel ist auch die Wahl von z. B. n = 1.000.000 ohne Einschränkung möglich, wenn erforderlich.

11.8. Schreiben von Werten in Dateien, Lesen von Werten aus Dateien

Wenn größere Datenmengen erzeugt und über den Lauf des aktuellen Programms hinaus bewahrt werden müssen, lassen sich diese auf Speichermedien ablegen und von dort wieder in ein Programm zurückholen. Die Nutzung dieser bei Rechnern grundlegenden Funktionalität ist natürlich auch bei der Sprache C gegeben. Eigentlich haben wir das nötige Werkzeug in leicht abgespeckter Fassung bereits dauernd im Einsatz. Wenn wir eine Bildschirmausgabe mit printf veranlassen, ist dies bereits ein Datenschreibvorgang, hier speziell auf das Standardausgabemediom „Bildschirm“. Mit wenigen Erweiterungen erreichen wir, dass die Ausgabe z. B. auf ein Festplatten-File erfolgt. Die Anordnung ist dann folgende:

a) Deklaration eines Zeigers auf einen Zwischenpufferbereich **`datenpuffer`** mit

```
FILE *datenpuffer;
```

b) Eröffnung des Zwischenpufferbereiches **datenpuffer** mit **fopen** und Zuweisung an das Ausgabefile „**Pfadname\\daten.txt**" sowie Angabe der gewünschten Funktion „schreiben" mit **„w"** (von write):

```
datenpuffer = fopen(„C:\\Daten\\C-Daten\\daten:txt","w");
```

c) Schreiben der Werte mit

```
fprintf (datenpuffer, „%f ,  %f , %f\n", x_Wert, y1_Wert, y2_Wert) ;
```

Dabei bedeutet **fprintf** = file print formatiert.

d) Schliessen des Zwischenpuffers mit

```
fclose(datenpuffer);
```

Als Beispiel dient die Berechnung der Leistungsverläufe $P_L(R_L)$ an einem Lastwiderstand R_L und Pges(RL) am gesamten Widerstand Ri + RL bei Speisung mit einer idealen Spannungsquelle der Leerlaufspannung U_i und Innenwiderstand R_i.

Wenn der Zwischenpufferbereich den Namen **leistung** erhält und der Dateiname mit dem Pfad **"C:\\Daten\\C_Daten\\leistung_1.txt"** ist, könnte das Programm so aussehen:

```
#include <stdio.h>

void main (void)
{
      int i, n;
      float Ri, RL, dRL, RLa, RLe, PL, Pges, Ui;
      FILE *leistung;

      printf("Gib Leerlaufspannung Ui in Volt an: ");
      scanf("%f", &Ui);
      printf("Gib Innenwiderstand Ri in Ohm an: ");
      scanf("%f", &Ri);
      printf("Gib Startwert RLa des Lastwiderstandes RL in Ohm an: ");
      scanf("%f", &RLa);
      printf("Gib Endwert RLe des Lastwiderstandes RL in Ohm an: ");
      scanf("%f", &RLe);
      printf("Gib Schrittweite dRL des Lastwiderstandes in Ohm an: ");
      scanf("%f", &dRL);

      leistung=fopen("C:\\Daten\\C_Daten\\leistung_1.txt","w");

      for(RL=RLa; RL<=RLe; RL=RL+dRL)
      {
            Pges = 1/(RL+Ri)*Ui*Ui;
            PL=RL/((RL+Ri)*(RL+Ri))*Ui*Ui;
            fprintf(leistung, "%f , %f\ , %f\n", RL, PL, Pges);
      }

      fclose(leistung);
}
```

Nach Ende des Programmlaufs kann man sich das File „**leistung_1.txt**" mit einem Editor ansehen und findet dort die Wertetabelle für RL, PL und Pges. Die Zahlen sind durch die Ausgabeformatierung mit fprintf mit Kommas getrennt. Als Beispiel für Ui =10 Volt, Ri = 2 Ohm, RLa = 0 Ohm, RLe = 10 Ohm, dRL = 0.2 Ohm hierder Auszug aus dem Textfile mit den ersten 21 Datenpaaren:

```
0.000000 , 0.000000 , 50.000000
0.200000 , 4.132231 , 45.454544
0.400000 , 6.944445 , 41.666668
0.600000 , 8.875740 , 38.461536
0.800000 , 10.204082 , 35.714287
1.000000 , 11.111111 , 33.333332
1.200000 , 11.718750 , 31.250000
1.400000 , 12.110726 , 29.411764
1.600000 , 12.345679 , 27.777777
1.800000 , 12.465374 , 26.315788
2.000000 , 12.500000 , 24.999998
2.200000 , 12.471655 , 23.809523
2.400000 , 12.396694 , 22.727270
2.600000 , 12.287334 , 21.739128
2.800000 , 12.152778 , 20.833332
3.000000 , 12.000000 , 19.999998
3.200001 , 11.834319 , 19.230768
3.400001 , 11.659807 , 18.518517
3.600001 , 11.479591 , 17.857141
3.800001 , 11.296076 , 17.241377
4.000000 , 11.111111 , 16.666666
.....
.....
```

usw.

Man kann diese im File „**leistung_1.txt**“ gespeicherten Daten auf wenigsten zwei Arten verwenden:

a) in einem anderen C-Programm , wenn sie dort weiter verarbeitet werden sollen
b) in einem Programm zur grafischen Darstellung, etwa um einen funktionalen Verlauf zu zeigen, siehe Abschnitt 11.9

Das Einlesen in einem anderen C-Programm könnte so aussehen:

```
#include <stdio.h>

void main (void)
{
      int i, n;
      float RL,PL, Pges;
      FILE *leistung;

      printf("Wieviele Werte-Gruppen sollen aus dem File leistung_1.txt einge-
      lesen werden?");
      scanf("%d",&n);
      leistung=fopen("C:\\Daten\\C_Daten\\leistung_1.txt","r");

      printf("\n  RL(Ohm)\t  PL(Watt)\t  Pges(Watt)\n\n");

      for(i=0;i<=n;i++)
      {
            fscanf(leistung, "%f , %f\ , %f\n", &RL, &PL, &Pges);
            printf("%f\t  %f\t  %f\n", RL,PL,Pges);
      }

      printf("\n\n");

      fclose(leistung);
}
```

Wie beim Schreiben von Daten muß auch zum Lesen zunächst mit

```
FILE *leistung;
```

ein Zeiger auf den Zwischenpufferbereich deklarieren. Danach folgt mit

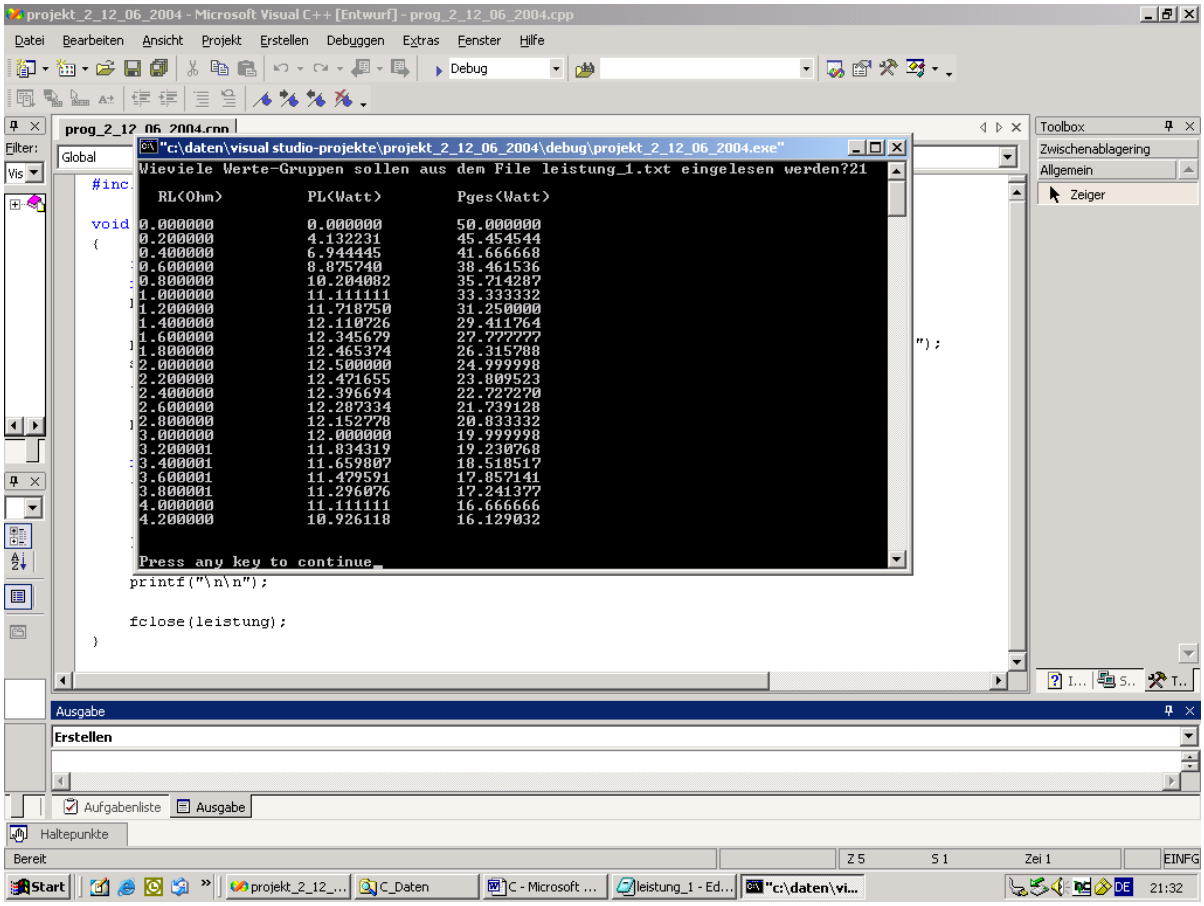
```
leistung=fopen("C:\\Daten\\C_Daten\\leistung_1.txt","r");
```

die Zuweisung zum File für die Funktion „ **r** “ (von read → lesen).

Das eigentliche Einlesen bewirkt die Funktion **fscanf** (→ file lesen formatiert), die der Anordnung im Textfile angepaßt ist:

```
fscanf(leistung, "%f , %f\ , %f\n", &RL, &PL, &Pges);
```

Danach können die Wertegruppen z. B. in einer Tabelle ausgegeben oder anderweitig verwendet werden. Gibt man n = 21 ein, so entsteht folgender Output (Hardcopy des Ausgabefensters):



Ein Vergleich mit dem Inhalt des oben gezeigten Ausschnitts des Textfiles zeigt die Übereinstimmung.

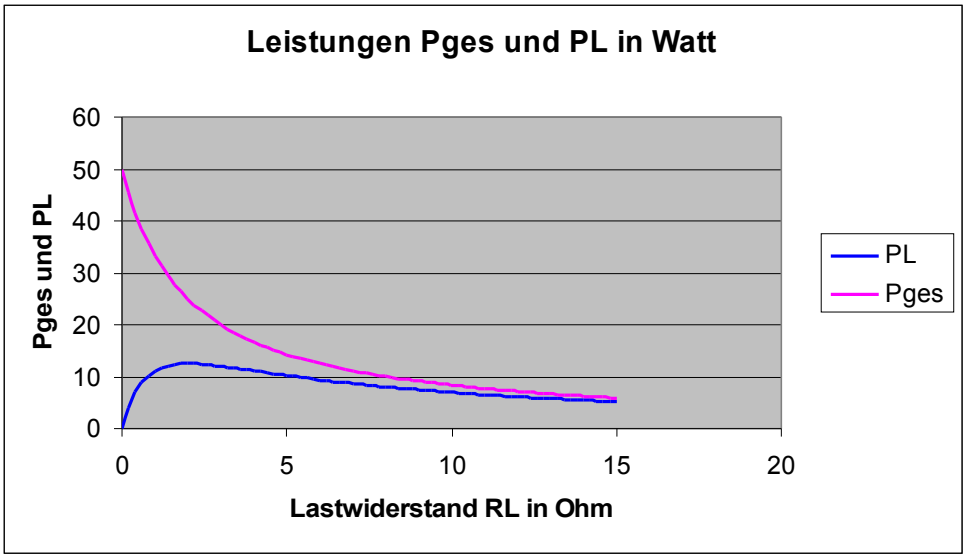
11.9. Erzeugen von grafischen Darstellungen aus C-Programmen mit Microsoft Excel

In technischen Anwendungen werden sehr oft grafische Darstellungen von zeitlichen oder örtlichen Verläufen gewünscht, weil sie dem Ingenieur im Gegensatz zu Wertetabellen einen übersichtlichen Gesamteindruck vom Verhalten eines technischen Systems vermitteln – sofern die Werte richtig berechnet wurden. Die Sprache C bietet keinen direkten Weg hierzu. Zwar kann man über Windows-Programmierung zum Ziel gelangen, muss dann aber in erst in diese Welt einsteigen. Ein nicht ganz so elegantes, vom Darstellungsergebnis aber doch recht brauchbares Verfahren lässt sich nutzen, wenn man die grafischen Darstellungsfunktionen von Excel und dessen Datenschnittstelle einsetzt.

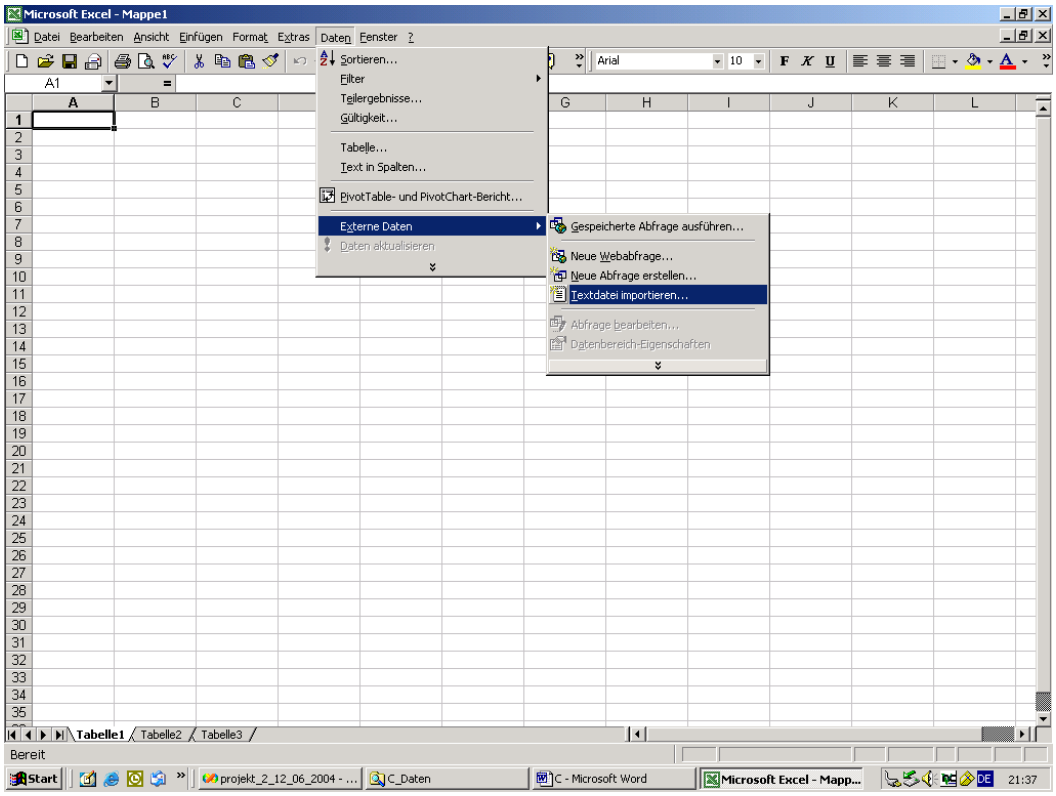
Der Ablauf ist denkbar einfach:

a) Erzeugen der darzustellenden Werte als .txt-File mit einem C-Programm
b) Übernahme der Werte über die Import-Schnittstelle von Excel
c) Erzeugen einer Excel-Grafik
d) Gestaltung der Grafik mit den vielseitigen Funktionen von Excel

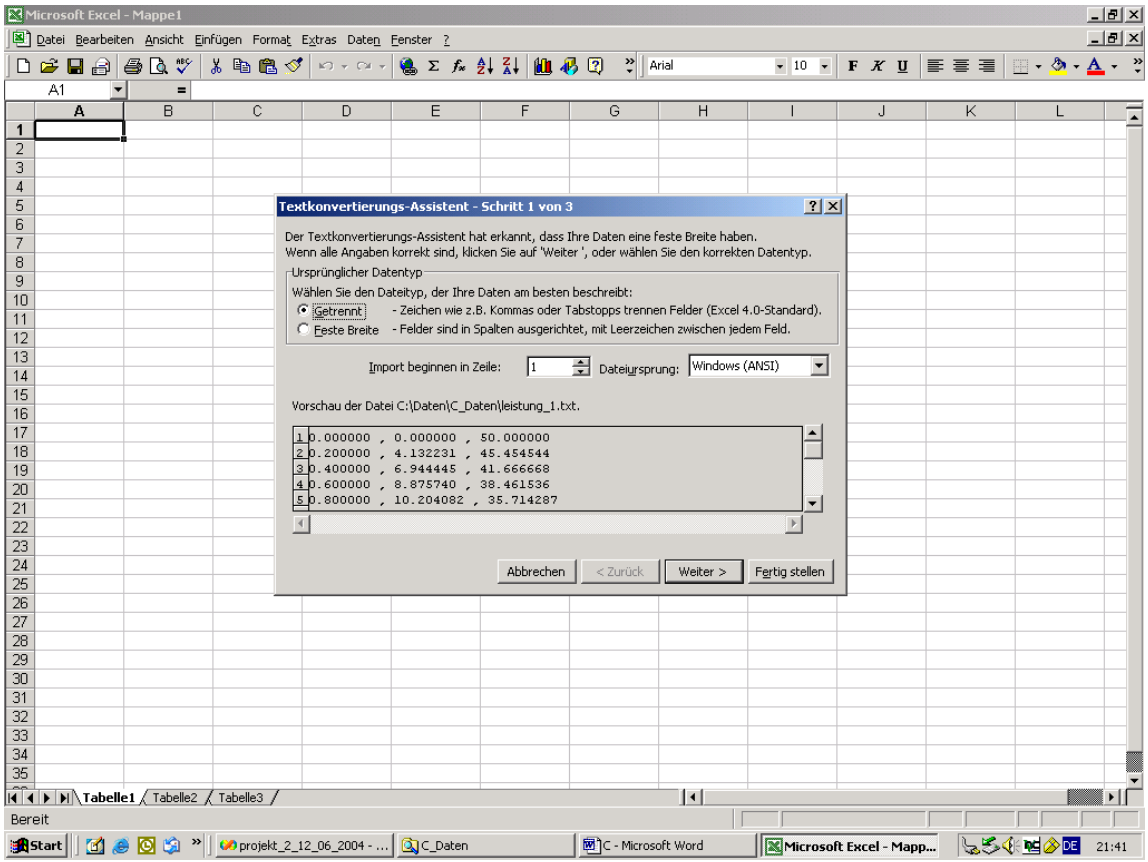
Als Beispiel dient wieder die Berechnung des Leistungsverlaufs $P_L(R_L)$ an einem Lastwiderstand R_L bei Speisung mit einer idealen Spannungsquelle der Leerlaufspannung U_i und mit Innenwiderstand R_i als Funktion von R_L, siehe Abschnitt 11.7. Die Daten wurden unter `"C:\\Daten\\C_Daten\\leistung_1.txt"` abgelegt (Punkt a). Das Ergebnis der grafischen Darstellung könnte dann so aussehen:



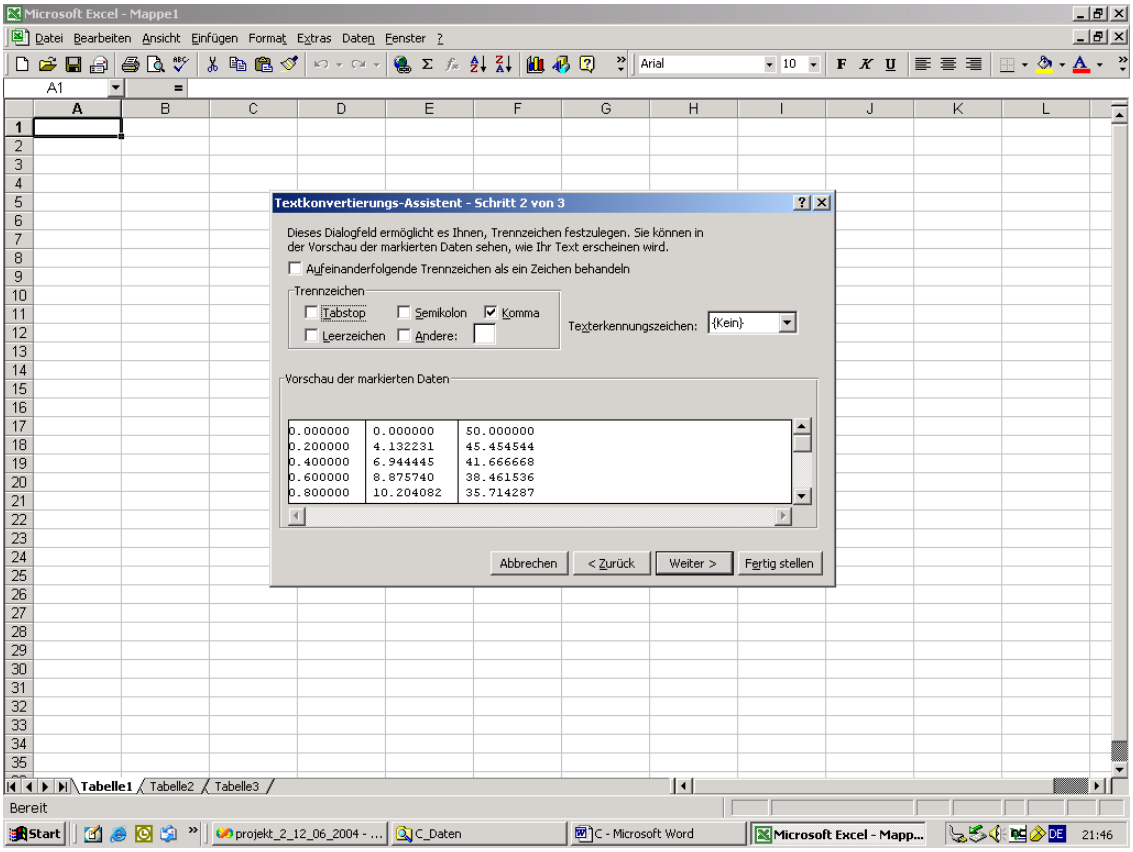
Um diese Darstellung zu erhalten, wird zunächst Excel aufgerufen und das Pulldown-Menue „Daten“ gewählt.



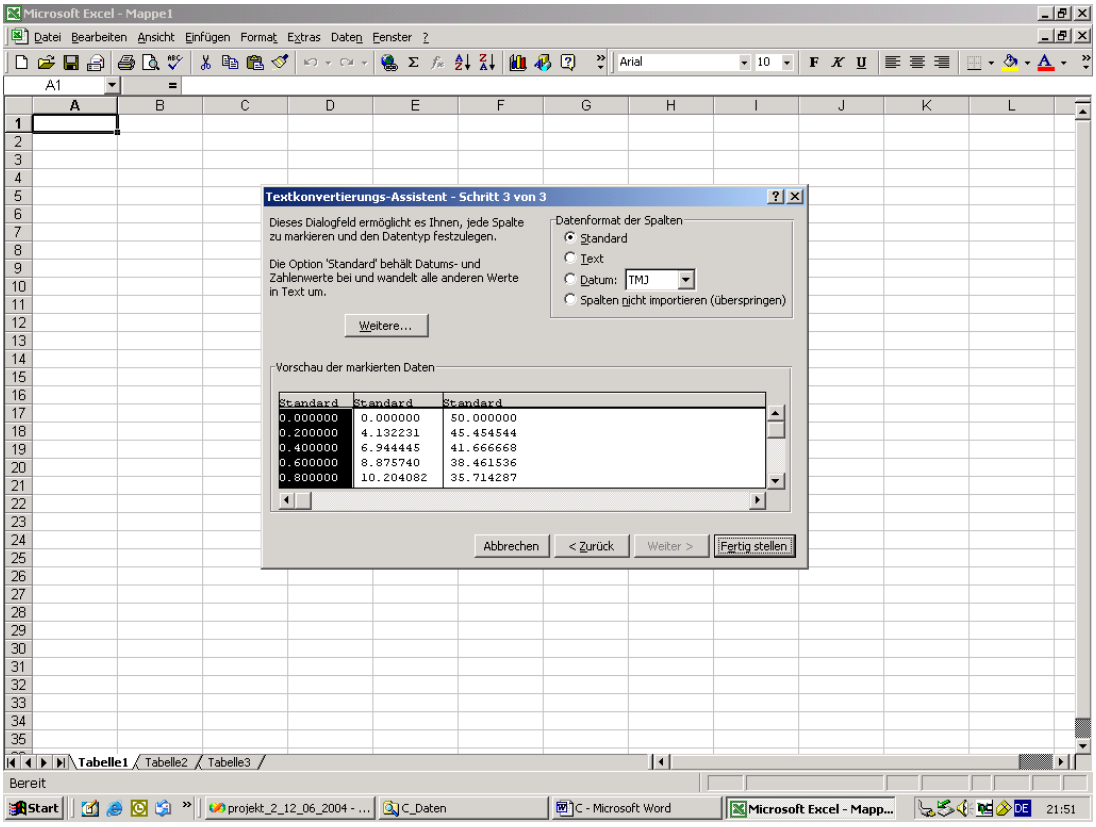
Wird der Punkt **Textdatei importieren** angeklickt, so kann man im Auswahlfenster das gewünschte File ansteuern. Danach erscheint der Textkonvertierungs-Assistent, mit dem sich die Daten aus dem .txt-File in eine für die Weiterverarbeitung in Excel geeignete Form bringen lassen:



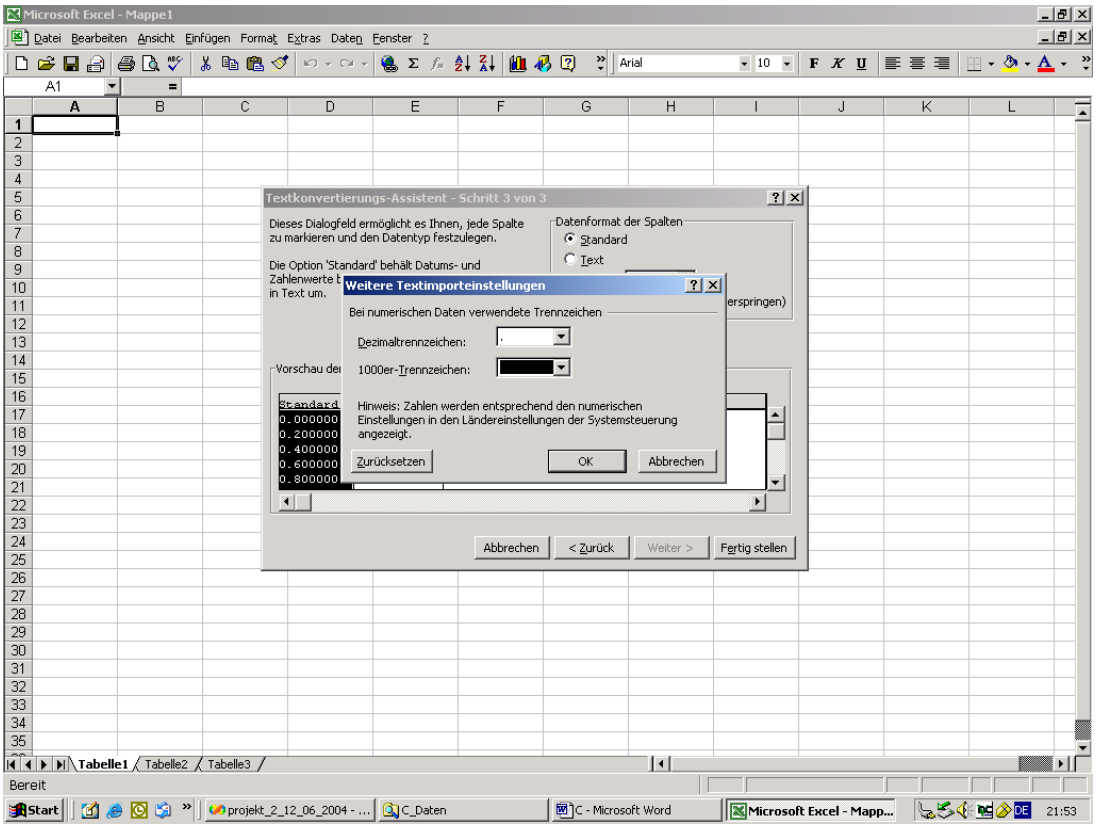
Hier ist der Punkt „Getrennt“ gewählt. Nach **Weiter** erscheint ein Fenster, mit dem die gewünschten Festlegungen zur Datenübernahme getroffen werden können. Wir haben jede Wertegruppe durch Kommas getrennt und benötigen keine Texterkennungszeichen wie etwa Gänsefüsschen.



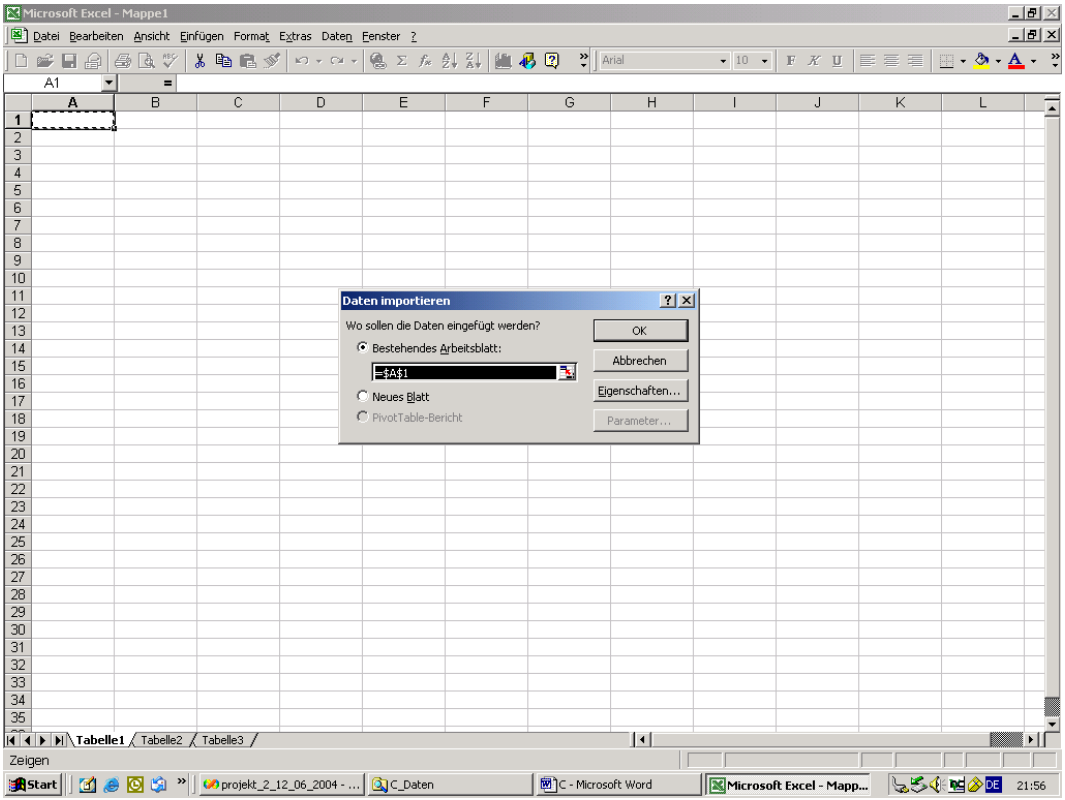
Im folgenden Fenster wird **Weitere...** angeklickt....



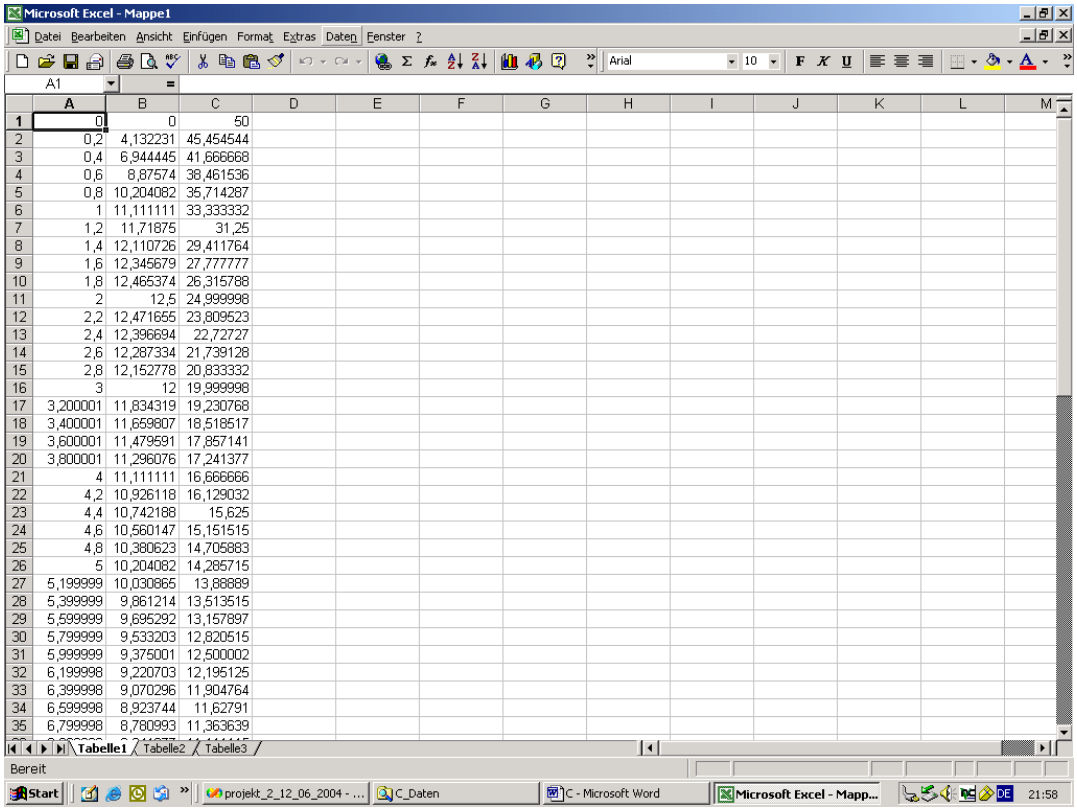
.... als Dezimaltrennzeichen der Punkt und als 1000er-Trennzeichen *nichts* ausgewählt:



Die Ablage in den Zellen der Excel-Tabelle beginnt in der Zelle **A1** links oben:

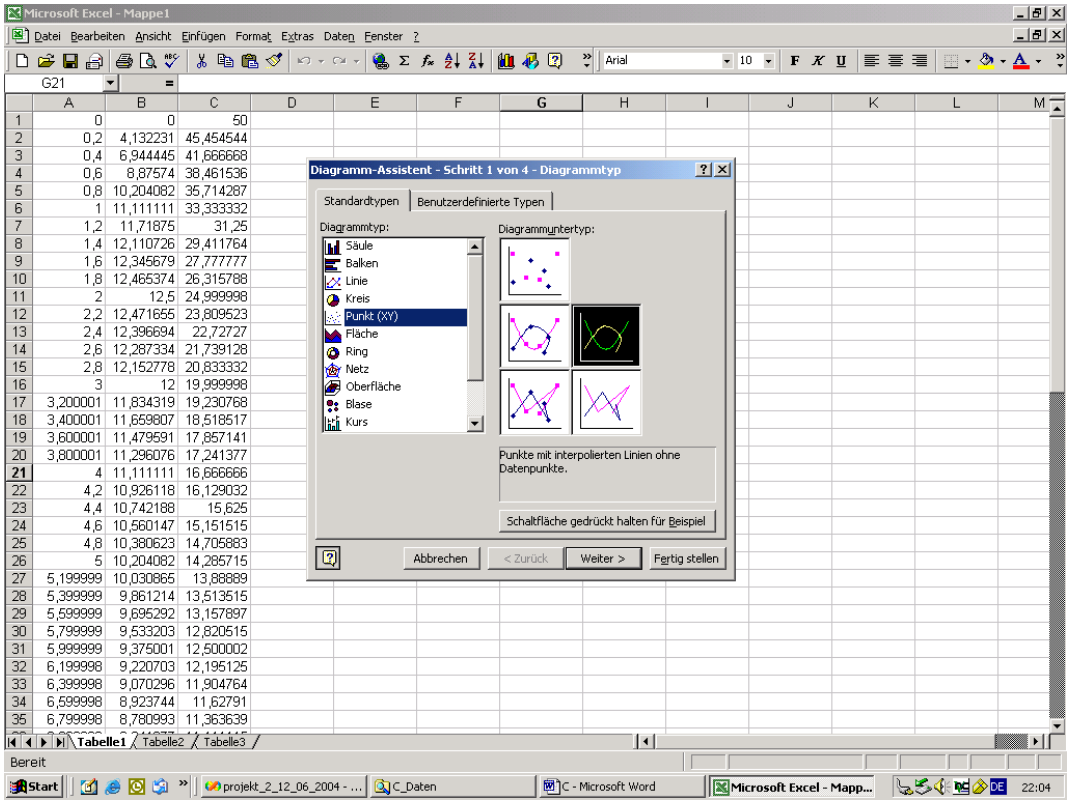


Nach Quittierung mit **ok** erscheinen die Daten des Textfiles im linken oberen Zellenbereich:



	A	B	C
1	0	0	50
2	0,2	4,132231	45,454544
3	0,4	6,944445	41,666668
4	0,6	8,87574	38,461536
5	0,8	10,204082	35,714287
6	1	11,111111	33,333332
7	1,2	11,71875	31,25
8	1,4	12,110726	29,411764
9	1,6	12,345679	27,777777
10	1,8	12,465374	26,315788
11	2	12,5	24,999998
12	2,2	12,471655	23,809523
13	2,4	12,396694	22,72727
14	2,6	12,287334	21,739128
15	2,8	12,152778	20,833332
16	3	12	19,999998
17	3,200001	11,834319	19,230768
18	3,400001	11,659807	18,518517
19	3,600001	11,479591	17,857141
20	3,800001	11,296076	17,241377
21	4	11,111111	16,666666
22	4,2	10,926118	16,129032
23	4,4	10,742188	15,625
24	4,6	10,560147	15,151515
25	4,8	10,380623	14,705883
26	5	10,204082	14,285715
27	5,199999	10,030865	13,88889
28	5,399999	9,861214	13,513515
29	5,599999	9,695292	13,157897
30	5,799999	9,533203	12,820515
31	5,999999	9,375001	12,500002
32	6,199998	9,220703	12,195125
33	6,399998	9,070296	11,904764
34	6,599998	8,923744	11,62791
35	6,799998	8,780993	11,363639

Hieraus kann nun die Grafik über den Menuepunkt **Einfügen Diagramm** erzeugt werden. Der genaue Ablauf wird hier nicht beschrieben, da er sich selbst erklärt. Als Diagrammtyp eignet sich im vorliegenden Fall das Format **Punkt (XY)** mit interpolierender Verarbeitung:

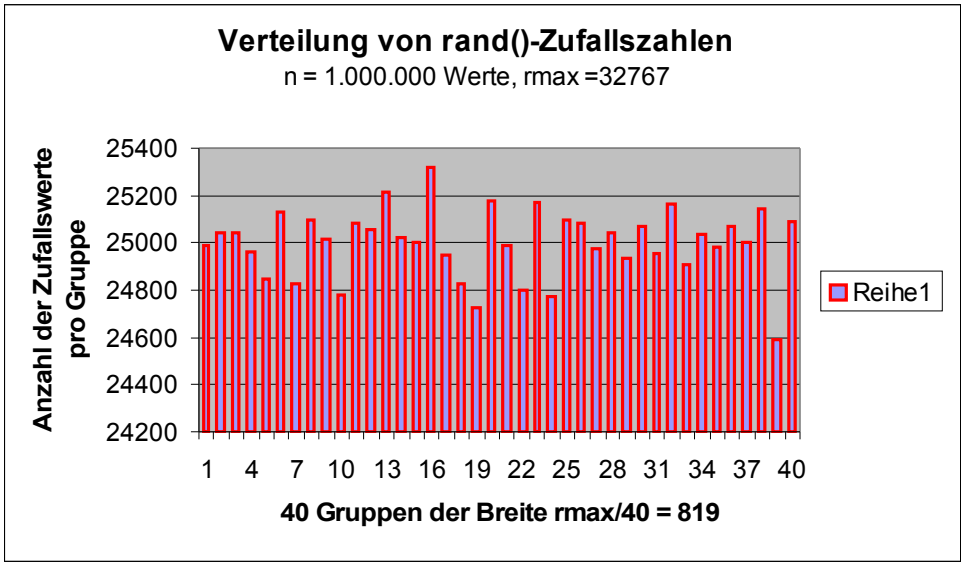


Nach passender Beschriftung der Achsen und Gestaltung weiterer Details (immer sehr zu empfehlen, damit man auch viel später noch erkennt, was eigentlich dargestellt wurde) erscheint das obengezeigte Diagramm.

Auch andere Darstellungen sind möglich. Betrachten wir dazu das folgende Programm, welches mit **rand ()** eine Menge von n Zufallszahlen erzeugt und diese in eine Anzahl von **teilung** Gruppen einsortiert, um die Art der Verteilung festzustellen:

```
// prog_1_19_05_2004
// Berechnung der Verteilung von n Zufallszahlen im Bereich [0, RAND_MAX]
// bei einer Gruppen-Anzahl von teilung Unterteilungen

#include <stdio.h>
#include <stdlib.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <time.h>
#include <malloc.h>

void main(void)
{
      int n,teilung, k, *kasten, rmax, *zufall;
      float x, y;
      FILE *datkasten;
      datkasten=fopen("C:\\Daten\\C_Daten\\gleichvert_1.txt","w");
      rmax = RAND_MAX;
      printf("rmax: %d\n\n", rmax);
      srand((unsigned) time(NULL));
      printf("Gib n an:");
      scanf("%d",&n);
      printf("Gib Teilung an:");
      scanf("%d",&teilung);
      zufall=(int *) malloc (n*sizeof(int));
      kasten=(int *) malloc (teilung*sizeof(int));

      for (k=0;k<n;k++)
      {
            zufall[k] = rand();
            if (n<=20) printf("%d\n", zufall[k]);
      }
      for(k=0;k<teilung;k++)kasten[k]=0;
            for
      (k=0;k<n;k++)kasten[(int)floor((double)zufall[k]/(double)rmax*(double)te
      ilung)]++;
      printf("\n\nVerteilung der Zufallswerte bei Teilung = %d\n\n", teilung);
      for (k=0;k<teilung;k++)
      {
      printf("% 6.0f   %d\n",(float) (k+1)*(float) rmax/(float) teilung,kas-
      ten[k]);
      fprintf(datkasten,"%6.0f, %d\n",(float) (k+1)*(float) rmax/(float) tei-
      lung,kasten[k]);
      }
      printf("\n");
      fclose(datkasten);
}
```

Bei n = 1000000 Werten und 40 Gruppen erhalten wir folgende Excel-Balkengrafik:



Wenn wir berücksichtigen, dass bei dieser Darstellung der Nullpunkt unterdrückt ist (in Excel kann man auch andere Skalierungen wählen) und die Werte alle in einem engen Bereich um 24800 liegen, können wir in guter Näherung von einer Gleichverteilung der Zufallszahlen ausgehen.

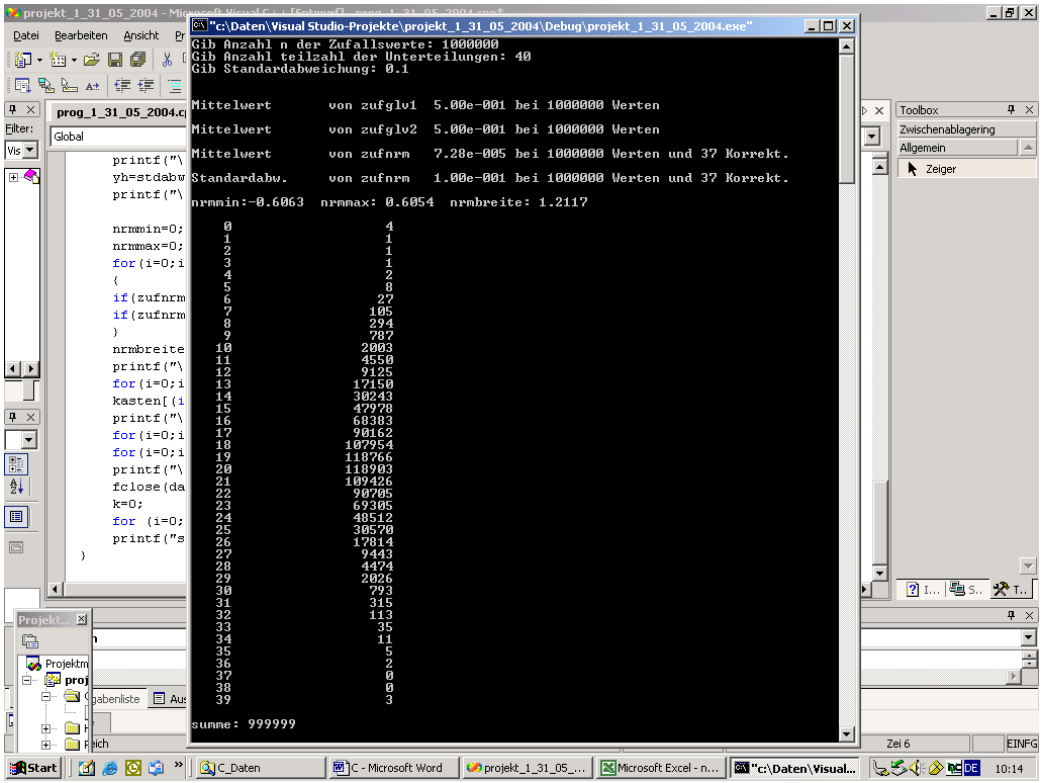
Zum Vergleich sehen wir hier noch ein Programm zur Erzeugung von normalverteilten (Gauss-verteilten) Zufallszahlen. Dazu werden mit rand () zwei statistisch unabhängige gleichverteilte Zahlenfolgen **`zufglv1`** und **`zufglv2`** bestimmt und aus diesen nach dem *Box-Müller*-Verfahren normalverteilte Werte **`zufnrm`** berechnet. Mit dem Faktor **`stabw`** lässt sich die Standardabweichung festlegen. Bei großem n entspricht die Standardabweichung in guter Näherung dem Effektivwert = Quadratwurzel aus dem quadratischen Mittelwert.

```
//Normalverteilte Zufallszahlen mit Mittelwert 0 und vorgebbarer Standardab-
weichung stabw » Effektivwert
//12.06.2004

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#define _USE_MATH_DEFINES
#include <math.h>

double mittwert (int n, double zuf[])
{
      int i;
      double xh;
      xh=0;
      for (i=0;i<n;i++)
      xh=xh+zuf[i];
      xh=xh/(double)n;
      return(xh);
}

double stdabw (int n, double zuf[])
{
      int i;
      double xh;
      xh=0;
      for (i=0;i<n;i++)
      xh=xh+zuf[i]*zuf[i];
      xh=sqrt(xh/(double)n);
      return(xh);
```

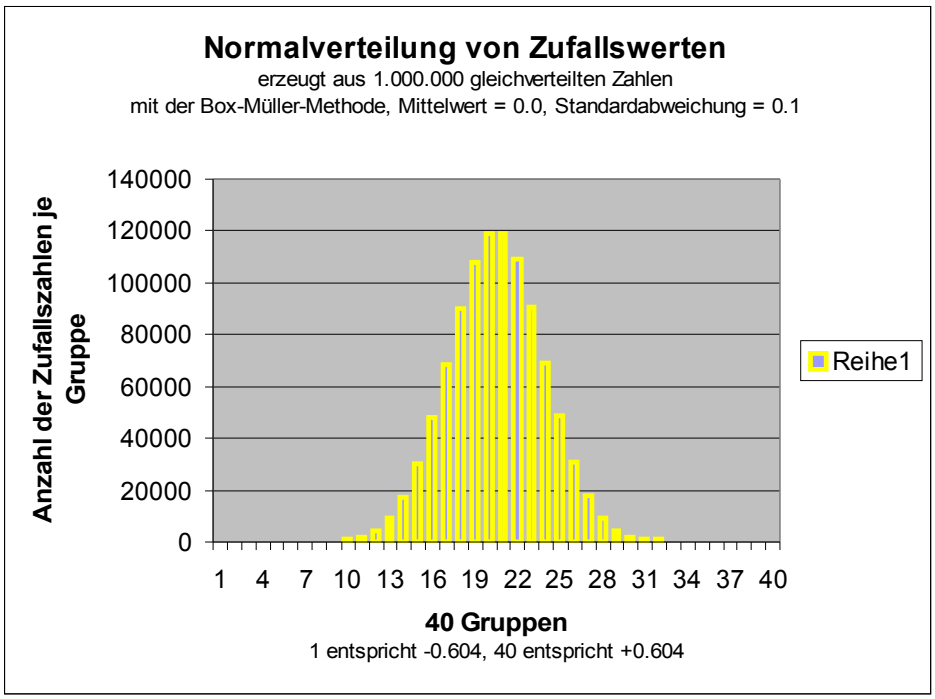
```
}

double feld (int n, int rmax, double zuf[])//Feldübergabe an Funktionsnamen
{
      int i;
      for (i=0;i<n;i++)
      zuf[i]=zuf[i]/(double)rmax;
      return(*zuf);
}

void main (void)
{
      int i,n,k,teilzahl,*kasten ;
      float stabw;
      double *zufglv1,*zufglv2,*zufnrm,yh, nrmmax, nrmmin, nrmbreite,faktor;

      FILE *datzufnrm, *datkasten;
      datzufnrm=fopen("C:\\Daten\\C_Daten\\zufnrm_1.txt","w");
      datkasten=fopen("C:\\Daten\\C_Daten\\normalvert_1.txt","w");

      srand((unsigned) time(NULL));

      printf("Gib Anzahl n der Zufallswerte: ");
      scanf("%d",&n);
      printf("Gib Anzahl teilzahl der Unterteilungen: ");
      scanf("%d",&teilzahl);
      printf("Gib Standardabweichung: ");
      scanf("%f",&stabw);

      faktor=(double)stabw;
      kasten=(int*)malloc(teilzahl*sizeof(int));
      for (i=0;i<teilzahl;i++) kasten[i]=0;
      zufglv1=(double*) malloc(n*sizeof(double));
      for(i=0;i<n;i++)
      zufglv1[i]=(double)rand();
      zufglv2=(double*) malloc(n*sizeof(double));
      for(i=0;i<n;i++)zufglv2[i]=(double)rand();
      zufnrm=(double*) malloc(n*sizeof(double));

      *zufglv1=feld(n,RAND_MAX,zufglv1);
      *zufglv2=feld(n,RAND_MAX,zufglv2);

      k=0;
      for(i=0;i<n;i++)
      {
            if (zufglv1[i]<=0)
            {
                  zufglv1[i]=0.00000001;
                  k++;
            };    //dient der Vermeidung von 0-Werten oder negativen Werten,
                  // für die der Logarithmus nicht definiert ist
            zufnrm[i]=faktor*sqrt(-
            2.0*log(zufglv1[i]))*cos(2.0*M_PI*zufglv2[i]); //Box-Müller-
                                                           //Alorithmus
      }
      printf("\n");

      fclose(datzufnrm);
      yh=mittwert(n,zufglv1);
      printf("\nMittelwert\t von zufglv1 % 5.2e bei %d Werten \n",yh,n);
      yh=mittwert(n,zufglv2);
      printf("\nMittelwert\t von zufglv2 % 5.2e bei %d Werten \n",yh,n);
```

```
        yh=mittwert(n,zufnrm);
        printf("\nMittelwert\t von zufnrm  % 5.2e bei %d Werten und %d
        Korrekt.\n",yh,n, k);
        yh=stdabw(n,zufnrm);
        printf("\nStandardabw.\t von zufnrm  % 5.2e bei %d Werten und %d Kor-
        rekt.\n",yh,n, k);

        nrmmin=0;
        nrmmax=0;
        for(i=0;i<n;i++)
        {
        if(zufnrm[i]>nrmmax) nrmmax=zufnrm[i];
        if(zufnrm[i]<nrmmin) nrmmin=zufnrm[i];
        }
        nrmbreite=nrmmax-nrmmin;
        printf("\nnrmmin:% 6.4f\tnrmmax:% 6.4f\tnrmbreite:% 6.4f\n",nrmmin,nrm-
        max,nrmbreite);
        for(i=0;i<n;i++)
        kasten[(int)floor((double)(zufnrm[i]-
        nrmmin)/(double)nrmbreite*(double)teilzahl)]++;
        printf("\n");
        for(i=0;i<teilzahl;i++) printf("% 5d\t  % 15d\n",i,kasten[i]);
        for(i=0;i<teilzahl;i++) fprintf(datkasten,"%15d\n",kasten[i]);
        printf("\n");
        fclose(datkasten);
        k=0;
        for (i=0;i<teilzahl;i++) k=k+kasten[i];
        printf("summe: %d\n\n",k);
}
```

Bei einer Anzahl von n = 1.000.000 Werten, einer Gruppen-Unterteilung von 40 und einer Standardabweichung von 0.1 sehen wir hier eine Hardcopy des Ausgabefensters:



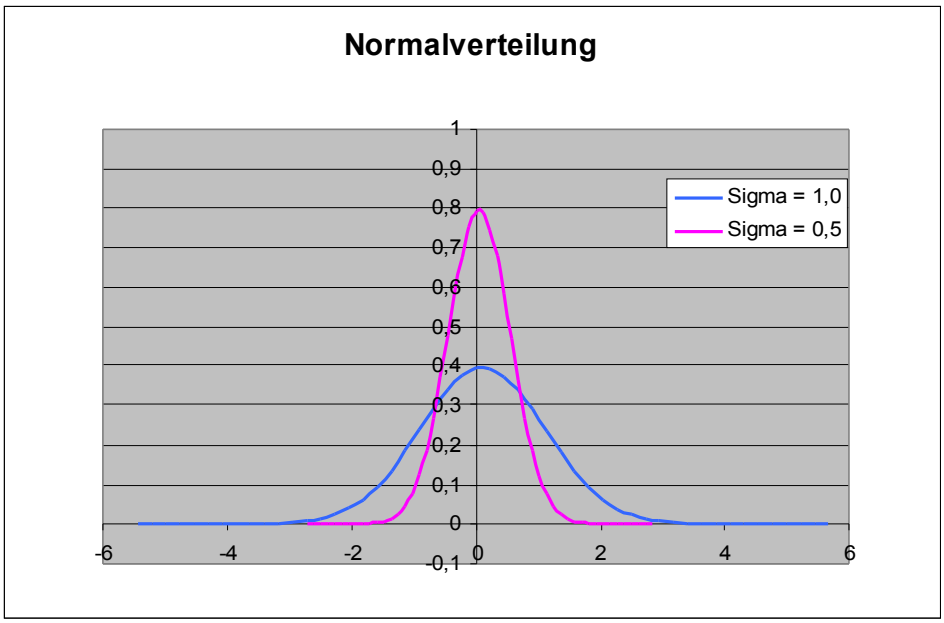
Über Excel erhalten wir aus dieser Wertetabelle folgende grafische (Balken-) Darstellung, welche die Gauss'sche Glockenkurve der Normalverteilungen bereits erkennen läßt :



Wird das obige Programm nun noch so ergänzt, dass die Wertetabellen annähernd auch die mathematisch korrekten Verteilungsdichtefunktionen (x = Zufallswert, σ = Standardabweichung (Effektivwert) der Verteilung, $\bar{x}$ = Mittelwert der Verteilung)

$$p(x,\sigma,\bar{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\bar{x})^2}{2\sigma}}$$

wiedergeben, so lässt sich für stabw = 0.5 bzw, stabw = 1.0 über Excel folgendes Diagramm aufbauen (die Mittelwerte sind in beiden Fällen 0), bei dem die Flächen unter den beiden Kurven jeweils den Wert 1 haben:



11.10. Verarbeitung von Zeichenketten

Insbesondere in der Nachrichtentechnik besteht die Aufgabe oft in der Informations-bezogenen Behandlung von Zeichenketten. Dabei geht es nicht nur um Texte, sondern ganz allgemein um Datenquellen in bestimmten Standardformaten. Ein viel verwendetes Format ist das Byte, doch genauso sind beliebige Vielfache von Bits zur Darstellung von Texten, Zahlen, Bildinformationen und anderem im Einsatz, die zweckmäßigerweise häufig in Teile der Größe von Bytes aufgebrochen werden, da die meisten Rechnersysteme hierfür sehr schnelle Verarbeitungsfunktionen bereitstellen.

Auch in C gibt es eine ganze Anzahl von Zeichenkettenfunktionen, einige wenige werden im folgenden Programm verwendet. Zunächst gilt, dass jede Zeichenkette als Feld vom Typ **`char`** deklariert wird, also als Feld mit je einem Byte als Element. Im Beipiel ist das das Feld **`text[200]`**. Mit **`fgets(text,200,datei);`** (→ file get string) wird eine ganze Zeichenkette aus einemTextfile in das feld **text** übernommen. Gegenüber numerischen Feldern weisen char-Felder die Besonderheit auf, dass vom deklarierten Umfang (hier 200) nur eine Menge vermindert um 1 nutzbar ist (hier also 199), da das Zeichen "\0" automatisch als Endekennung eingefügt wird.

Mit **`strlen(text`**) kann die Zahl der eingelesenen Zeichen festgestellt werden.

Als Beispiel betrachten wir eine sehr häufige Aufgabenstellung der Nachrichtentechnik. Es geht darum, eine Zeichenfolge über eine Leitung oder einen Kanal zu übertragen und die Störungen durch ein Rauschsignal zu beobachten, um Maßnahmen zur Beseitigung der dadurch am Empfangsort entstehenden Fehler untersuchen zu können. Dies ist im ersten Stadium am einfachsten und systematischsten durch eine entsprechende Simulation mit einem Programm möglich.

Die einzelnen Schritte:

- Bereitstellen des Textes (der zu übertragenden Information) als Zeichenkette **`sendetext`** **(**char-Feld)
- Zerlegen des Textes **`sendetext`** in eine Folge von Bits (0,1-Folgen)
- Modulieren eines elektrischen Rechtecksignals **`vmod`** zur Übertragung auf einer Leitung. Dabei gilt die Zuordnung 0-Bit → -1 Volt, 1-Bit → 1 Volt.
- Erzeugung einer normalverteilten, mittelwertfreien Zufallszahlenfolge **`r`** mit vorgebbarer Standardabweichung (= hier praktisch gleichbedeutend dem Effektivwert) als Störsignal
- Addition von **`vmod`** und **`r`** zum Empfangssignal **`wempf = vmod + r.`**
- Hard-Decision-Demodulation des Empfangsignals **`wempf`** zum Signal **`wdmd`**. Dabei gilt die Zuordnung

  wempf[i]>=0 → wdmd[i] =1

  wempf[i] < 0 → wdmd[i] = 0

- Zusammensetzen von jeweils 8 Bit aus der Folge **`wdmd`** zu Bytes
- Zuordnung der Bytes zur Folge des Feldes **`empfangstext`**
- Vergleich des gesendeten Textes mit dem empfangenen Text und Zählung der Bitfehler **`fehler`**
- Die Gesamtsumme der Fehler ist die **`fehleranzahl`**
- Bestimmung der prozentualen **Fehlerrate = fehleranzahl/Zahl aller Bits** in Abhägigkeit vom Effektivwert des Rauschsignals
- Einbau von Fehlerkorrekturverfahren zur wirksamen Verringerung der **Fehlerrate**

Hier nun ein solches, zur Berechnung der Fehlerrate geeignetes Programm:

```
//Einlesen von Text aus Datei Text_1.txt
```

```
//Darstellung der ASCII-Zeichen im Binärformat
//Erzeugung eines modulierten Signals vmod
//Erzeugung eines mittelwertfreien, normalverteilten Rauschsignals r
//mit Standardabweichung stdabw
//Erzeugung eines Empfangssignals wempf=vmod+r
//Erzeugung eines hard-decision Signals wdmd
//Berechnung der Bitfehler fehler[i] und der Fehleranzahl fehleranzahl
//Verwendung der eigenen Bibliothek funktionen_1.txt
//20.06.2004

#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <malloc.h>
#include "C:\\Daten\\C_Daten\\funktionen_1.txt"


void main (void)
{
      int i, j, rmax, nstr, nr, zeichenanzahl,fehleranzahl, *fehler,zweierpo-
      tenz[8], byte[8];
      char b1,*sendetext, *empfangstext;
      unsigned char l1u;
      float stdabw,*zufglv1,*zufglv2,*vmod,*r,*wempf,*wdmd;

      for (i=0;i<8;i++) zweierpotenz[i]=pow(2,7-i);

      zeichenanzahl=400;
      sendetext=(char*) malloc(zeichenanzahl*sizeof(char));
      empfangstext=(char*) malloc(zeichenanzahl*sizeof(char));

      printf("Einlesen von Text aus Datei Text_1.txt\n\n");
      FILE *datei;
      datei=fopen("C:\\Daten\\C_Daten\\Text_1.txt","r");
      fgets(sendetext,zeichenanzahl,datei);
      fclose(datei);

      printf("\n");
      //puts(sendetext);
      nstr=strlen(sendetext);
      printf("\nAnzahl der aus der Datei Text_1.txt eingelesenen
      Zeichen:\t%d\n", nstr);

      nr=nstr*8;
      zufglv1=(float*) malloc(nr*sizeof(float));
      zufglv2=(float*) malloc(nr*sizeof(float));
      vmod=(float*) malloc(nr*sizeof(float));
      r=(float*) malloc(nr*sizeof(float));
      wempf=(float*) malloc(nr*sizeof(float));
      wdmd=(float*) malloc(nr*sizeof(float));
      fehler=(int*) malloc(nr*sizeof(int));

      srand((unsigned) time(NULL));
      stdabw=0.4;
      rmax=RAND_MAX;
      //printf("rmax:\t%d\n",rmax);

      for (i=0;i<nr;i++) zufglv1[i]=rand();
      for (i=0;i<nr;i++) zufglv2[i]=rand();
```

```
*zufglv1=normfeld(nr,rmax,zufglv1);
*zufglv2=normfeld(nr,rmax,zufglv2);
for (i=0;i<nr;i++) r[i]=stdabw*sqrt(-
2.0*log(zufglv1[i]))*cos(2.0*M_PI*zufglv2[i]);

//for(i=0;i<255;i++) printf("% 4d\t%c\n",i,(unsigned char)i);

b1=1;

for(j=0;j<nstr;j++)//Erzeugen eines modulierten Sendesignals "0"->1
Volt, 1-> -1 Volt
{
      l1u=sendetext[j];
      if(l1u==228) l1u=132; //ä
      if(l1u==246) l1u=148; //ö
      if(l1u==252) l1u=129; //ü
      if(l1u==196) l1u=142; //Ä
      if(l1u==214) l1u=153; //Ö
      if(l1u==220) l1u=154; //Ü
      if(l1u==223) l1u=225; //ß

      printf("\n%4d\t%c  %u\t",j,l1u, l1u);
      for (i=7;i>=0;i--)
      {
            if (i==3) printf(" ");
            if(_rotr(l1u,i) & b1)
            {
                  printf("1");
                  vmod[8*j+7-i]=1;
            }
            else
            {
                  printf("0");
                  vmod[8*j+7-i]=-1;
            }
      }
}
printf("\n\n");
for (i=0;i<nstr;i++) printf("% 1.f",vmod[i]);
printf("\n\n");

printf("Mittelwert von vmod:\t\t % 2.6f\n", mittelwert(nstr, vmod));
printf("Effektivwert von vmod:\t\t % 2.6f\n", effektivwert(nstr, vmod));
printf("Mittelwert von zufglv1:\t\t % 2.6f\n", mittelwert(nr, zufglv1));
printf("Effektivwert von zufglv1:\t % 2.6f\n", effektivwert(nr,
zufglv1));
printf("Mittelwert von zufglv2:\t\t % 2.6f\n", mittelwert(nr, zufglv2));
printf("Effektivwert von zufglv2:\t % 2.6f\n", effektivwert(nr,
zufglv2));
printf("Mittelwert von r:\t\t % 2.6f\n", mittelwert(nr, r));
printf("Effektivwert von r:\t\t % 2.6f\n\n", effektivwert(nr, r));

fehleranzahl=0;
for(i=0;i<nr;i++)
{
      wempf[i]=vmod[i]+r[i];
      if (wempf[i]>=0) wdmd[i]=1;
      else wdmd[i]=-1;
      if(wdmd[i]==vmod[i])fehler[i]=0;
      else
      {
            fehler[i]=1;
```

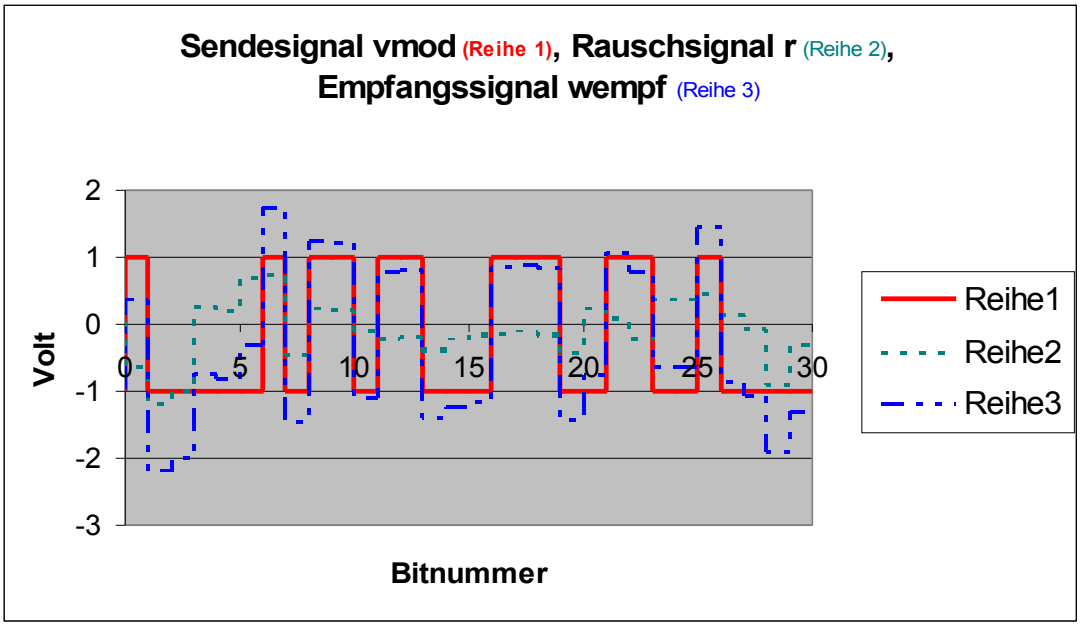
```
                fehleranzahl=fehleranzahl+1;
            };
        }
        printf("\nFehleranzahl bei stdabw %3.1f:\t%d\t und prozentualer Anteil
        %3.1f\n\n",stdabw,fehleranzahl,100*(float)fehleranzahl/(float)nr);

        printf("\nSkalarprodukt %d\n",skalarprodukt(nr,fehler,fehler));


        for(i=0;i<nstr;i++)
        {
            for(j=0;j<8;j++)
            {
                if(wdmd[i*8+j]==1) byte[j]=1;
                else byte[j]=0;
            }
            empfangstext[i]=skalarprodukt(8,byte,zweierpotenz);
        }
        printf("\nGesendeter Text:\n\n");
        j=100;
        for (i=0;i<nstr;i++)
        {
            if (i>j && i<(j+30) && sendetext[i]==32)
            {
                printf("\n");
                j=j+100;
            };
            l1u=sendetext[i];
            if(l1u==228) l1u=132; //ä
            if(l1u==246) l1u=148; //ö
            if(l1u==252) l1u=129; //ü
            if(l1u==196) l1u=142; //Ä
            if(l1u==214) l1u=153; //Ö
            if(l1u==220) l1u=154; //Ü
            if(l1u==223) l1u=225; //ß
            printf("%c",l1u);
        }
        printf("\n\nEmpfangener Text:\n\n");
        j=100;
        for (i=0;i<nstr;i++)
        {
            if (i>j && i<(j+30) && empfangstext[i]==32)
            {
                printf("\n");
                j=j+100;
            };
            printf("%c",empfangstext[i]);
        }
        printf("\n\n");

        FILE *puffer;
        puffer=fopen("C:\\Daten\\C_Daten\\vmod_wempf.txt","w");
        for(i=0;i<30;i++)fprintf(puffer,"%d\t%d\t% 6.4f\t% 6.4f\n%d\t%d\t%
        6.4f\t%6.4f\n%d\t%d\t%
        6.4f\t%6.4f\n",i,(int)vmod[i],r[i],wempf[i],i,(int)vmod[i+1],r[i+1],wemp
        f[i+1],i+1,(int)vmod[i+1],r[i+1],wempf[i+1]);
        fclose(puffer);
}
```

Für den **`sendetext`**

Als in unserer Stadt wohnhaft ist eine Minderjährige aktenkundig, welche infolge ihrer hierorts üblichen Kopfbedeckung gewohnheitsrechtlich Rotkäppchen genannt zu werden pflegt. Vor ihrer Inmarschsetzung wurde die R. seitens ihrer Mutter über das Verbot betreffs Verlassens der Waldwege auf Kreisebene belehrt. Sie machte sich infolge Nichtbeachtung dieser Vorschrift straffällig und begegnete beim Überschreiten des diesbezüglichen Blumenpflückverbotes einem polizeilich nicht gemeldeten Wolf ohne festen Wohnsitz. Dieser verlangte in unberechtigter Amtsanmaßung Einsichtnahme in den zum Transport von Konsumgütern dienenden Korb und traf zwecks Tötungsabsicht die Feststellung, daß die R. zu ihrer verwandten und verschwägerten Großmutter eilends war. Da bei dem Wolfe Verknappungen auf dem Ernährungssektor vorherrschend waren, beschloß er, bei der Großmutter der R. unter Vorlager falscher Papiere vorsprachig zu werden. Da dieselbe wegen Augenleidens krank geschrieben war, gelang dem Wolf die diesfällige Täuschungsabsicht, worauf er unter Verschlingung der Bettlägerigen einen strafbaren Mundraub ausführte. Bei der später eintreffenden R. täuschte er seine Identität mit der Großmutter vor, stellte R. nach und durch Zweitverschlingung derselben seinen Tötungsvorsatz unter Beweis. Der sich auf einem Dienstgang befindliche Förster B. vernahm verdächtige Schnarchgeräusche und stellte Urheberschaft seitens des Wolfsmaules fest. Er reichte bei seiner vorgesetzten Dienststelle ein Tötungsgesuch ein, welches zuschlägig beschieden wurde. Darauf gab er einen Schuß auf den Wolf ab. Dieser wurde nach Infangnahme der Kugel ablebig. Die Beinhaltung des Getöteten weckte in dem Schußabgeber die Vermutung, daß der Leichnam Personen beinhalte.

wird bei einem Effektivwert des Rauschsignals r von r_{eff} =0.5 (stdabw =0.4) die Fehlerrate bestimmt.

Die Diagramme der einzelnen Folgen haben dabei folgendes Aussehen:



Von den vielen Funktionen zur Zeichenkettenverarbeitung hier zwei weitere, die häufig gebraucht werden:

- `sprintf (string, "% 6d % 7.2f  %5.1e", x, y, z);`

- `strcat(string_1, string_2);`

Mit **`sprintf`** kann eine Folge von Zahlen formatiert in eine Zeichenkette `string` umgewandelt werden. Eine Anwendung ist z.B der Formatstring in der **`printf`**-Anweisung. Bisher war diese Zeichenkette immer explizit in die printf-Anweisung einzuprogrammieren, so dass eine Änderung des Ausgabeformats über eine Programmänderung zu erfolgen hatte, ein nur für die Entwicklungsphase hinnehmbares Verfahren.

Will man das Ausgabeformat über Eingabeaufrufe bestimmen, so können - weniger elegant - entweder die Formatstrings selbst oder - besser - die eigentlich interessierenden Parameter, aus denen sie sich aufbauen, eingegeben werden. Für den letzteren Weg benötigen wir die beiden zuvor genannten Funktionen. Die Parameter gibt man über

die `scanf`-Funktion ein, die beiden Zahlen-Parameter *Gesamtstellen* und *Nachkommastellen* werden anschließend über die `sprintf` - Funktion in Zeichenketten gewandelt.
Ein Beispiel: es ist eine Gleitkommazahl x auszugeben. Das Format soll aus

- einer vorgebbaren Gesamtstellenzahl
- einer vorgebbaren Anzahl von Nachkommastellen
- einem reinen Gleitpunktformat (%f) oder einem Exponentialformat (%e)
-

bestehen. Folgendes Programm kann diese Anforderungen erfüllen (wobei noch einiges für das Abfangen von fehlerhaften Eingaben getan werden müsste):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
      int i, gesamtstellenzahl,nachkommastellen;
      double x;
      char typ,str[20];
      for (i=0;i<19;i++) str[i]='\0';
      x=12.3567;
      printf("Gesamtstellenzahl des Ausgabformats: ");
      scanf("%d",&gesamtstellenzahl);
      printf("Nachkommastellenzahl des Ausgabeformats: ");
      scanf("%d",&nachkommastellen);
      printf("Typ des Ausgabformats (f -> Gleitpunktformat, e -> Exponential-
      format: ");
      scanf(" %c",&typ);

      sprintf(str,"%d.%d%c",gesamtstellenzahl,nachkommastellen,typ);

      printf(strcat(strcat("%",str),"\n"),x);//der Formatstring wird aus
      "%"+str+"\n" zusammengesetzt
}
```

Die oben erwähnte Funktion **`strcat(`**`string1, string2`**`)`** baut je zwei Zeichenketten `string1, string2` zusammen, eine oder beide können selbst wieder als `strcat` -Funktion erscheinen, siehe `strcat(strcat("%",str),"\n")` im obigen Beispiel. Die Summenzeichenkette, in der das Ergebnis gespeichert wird, ist die erste, also `string1`. Das Ergebnis `string1` muß gemäß der inneren Struktur der Zeichenkette als Endekennung das \0-Zeichen (=NUL) aufweisen.

Achtung: nicht jeder Compiler sorgt von sich aus dafür! Das kann zu scheinbar unerklärlichen „Hängern“ beim Programmlauf führen. Man kann das vermeiden, wenn man bei `string2` als Endezeichen ein NUL erzwingt. Besteht string2 aus nur einem nutzbaren Zeichen, z. B. ‚q’, so hat es die reale Länge 2, nämlich das Byte für ‚q’ und das Byte für NUL. Um ganz sicher zu gehen, dass dieser Aufbau auch vorliegt, kann man die Variante 1 des folgenden Programmaufbaus verwenden:

```
#include <stdio.h>
#include <string.h>

void main(void)
{
      //Deklaration, Wertzuweisung und Verkettung von Zeichenketten
      //Variante 1: hält sich direkt an den Feldcharakter von Zeichenketten
      char string1[6]={'a','b','c','d','e'}, string2[2]={'q','\0'};
      printf("Variante 1:\n----------\n");
      printf("%s\n",string1);
      printf("%s\n",string2);
      strcat(string1, string2);
      printf("%s\n",string1);
```

```
        printf("\n");

        //Variante 2: Ergebnis identisch zu Variante 1, aber einfachere Wertzu-
        weisung
        char str1[]="abcde",str2[]="q";
        printf("Variante 2:\n-----------\n");
        printf("%s\n",str1);
        printf("%s\n",str2);
        strcat(str1,str2);
        printf("%s\n",str1);
        printf("\n");

        //Variante 3: Ergebnis identisch zu den Varianten 1 und 2, aber Deklara-
        tion der Zeichenketten
        // nicht als Variable, sondern als Zeiger auf die Zeichenketten
        char *str1zeiger,*str2zeiger;
        printf("Variante 3:\n-----------\n");
        str1zeiger="abcde";
        str2zeiger="q";
        printf("%s\n",str1zeiger);
        printf("%s\n",str2zeiger);
        strcat(str1zeiger,str2zeiger);
        printf("%s\n",str1zeiger);
        printf("\n");
}
```

Ausgabe:

```
Variante 1:
-----------
abcde
q
abcdeq

Variante 2:
-----------
abcde
q
abcdeq

Variante 3:
-----------
abcde
q
abcdeq
```

Wie zu sehen, funktionieren bei der von uns verwendeten Entwicklungsumgebung Visual.Net auch die einfacheren Varianten 2 und 3.

11.11. KomplexeZahlen

In der Ingenieurtechnik bietet die Rechnung mit komplexen Zahlen in vielen Teilgebieten von Elektrotechnik und Maschinenbau sehr kompakte, übersichtliche Verfahren und Darstellungen, weshalb die meisten Programmiersprachen, so auch C, hierfür geeignete Strukturen und Funktionen bereitstellen. Allerdings sind diese nicht sehr standardisiert, so dass man sich bei jeder Sprachversion zunächst mit den speziellen Eigenheiten vertraut machen muss.

Komplexe Zahlen sind Zeigergrößen, sie haben in der komplexen Ebene einen Betrag (eine „Länge“) und eine Richtung (gegeben durch einen Winkel, das so genannte Argument). Insofern sind sie mit ebenen Vektoren verwandt. Komplexe Zahlen bestehen aus einem Real- und einem Imaginärteil, der Imaginärteil ist das reelle Vielfache der

Wurzel aus –1. Diese imaginäre Einheit mit dem Betrag 1 und der Richtung 90 Grad in der komplexen Ebene wird oft mit i oder j bezeichnet, in der Elektrotechnik eher mit j, um Verwechslungen mit dem Strom i vorzubeugen.

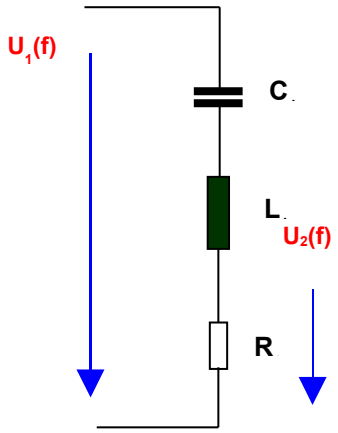
Beispiel: `z1 = 2.5 + j*6.3`

Komplexe Zahlen bilden mathematisch einen Körper, die vier Grundrechenarten sind definiert, es gibt ein Null- und ein Eins-Element.
Um in unserer Entwicklungsumgebung von MS Visual.Net mit komplexen Zahlen arbeiten zu können, benötigen wir die Bibliothek **<complex>** (ohne „.h“) und die Deklaration **using namespace std;** , mit der die Standardregeln für die Namensvergabe auch bei komplexen Zahlen gelten.

Komplexe Variable werden dann als Struktur **complex <float>** oder **complex <double>** an der Stelle im Programm deklariert, wo man sie braucht.

Achtung: Alle Operationen mit komplexen Zahlen dürfen nur mit komplexen Zahlen ausgeführt werden, also nicht mit reellen Zahlen mischen. Statt 1 muß z. B. eine komplexe Struktur wie `ze(1,0)`, statt der imaginären Einheit `j = sqrt (-1)` etwa `zj (0,1)` verwendet werden.

Zur Erläuterung betrachten wir die Berechnung der Frequenz-Übertragungsfunktion z3 eines Reihenschwingkreises mit R_1, C_1 und L_1 in Abhängigkeit von der Frequenz f . Gemäß dem Schaltbild



$$z3 = \frac{\tilde{U}_2}{\tilde{U}_1}(2\pi f) = \frac{\tilde{U}_2}{\tilde{U}_1}(\omega) = \frac{R_1}{R_1 + j\left(-\frac{1}{\omega C_1} + \omega L_1\right)}$$

kann man den obigen Frequenzgang z_3 angeben.

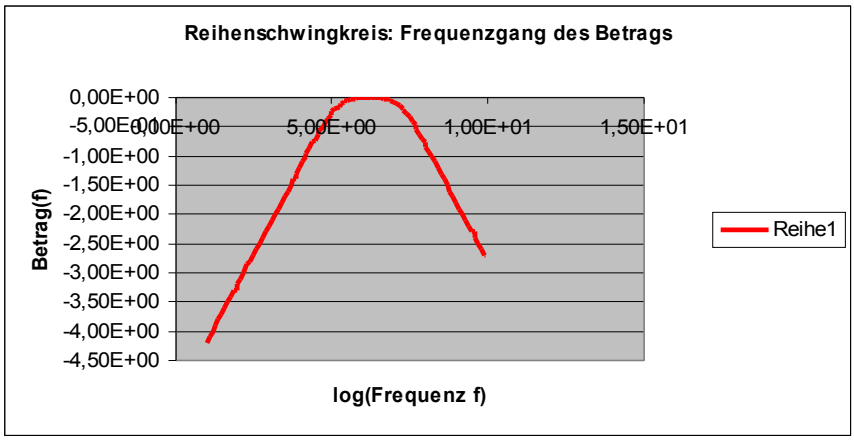
Für Widerstand, Kapazität und Induktivität wurden Werte von 10 Ohm, 100 Nano-Farad und 0.1 Mikro-Henry vorgegeben, die Frequenz variiert zwischen 10 Hz und 100 Giga-Hz. Damit sieht das Programm wie folgt aus:

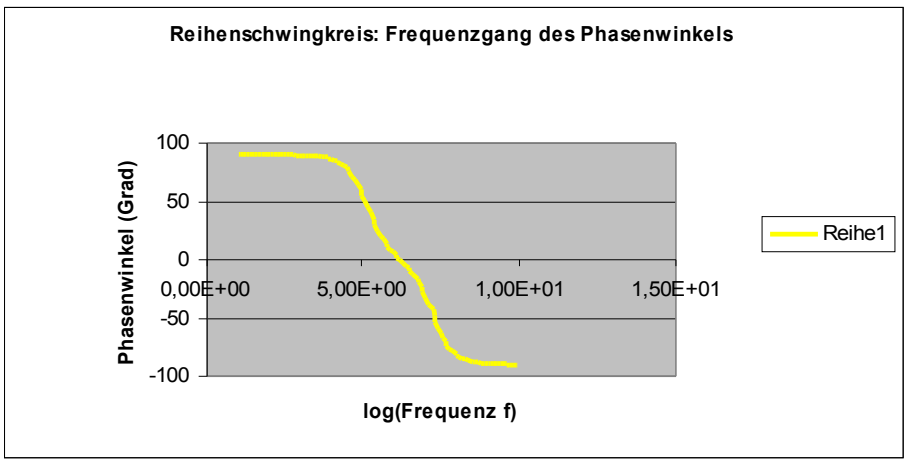
```
/komplexe Zahlen am Beispiel des Frequenzganges eines Reihenschwingkreises
//13.06.2004

#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <complex>

void main()
{
      int j, k;
      double x, y, z, f,omega,R1,C1,L1;
      FILE *puffer;
      puffer=fopen("C:\\Daten\\C_Daten\\Frequ_gang.txt","w");
      using namespace std;
      R1=10;
      C1=0.0000001;
```

```
        L1=0.0000001;
        printf("f
        [Hz]\t\tlog(f)\t\tBetrag(z3)\tlog(Betrag(z3))\tPhase(z3)\n\n");

        for(f=10;f<100000000000;f=f*1.2)
        {
              omega=2*M_PI*f;
              complex <float> z1(R1,0),z2(R1,-1/(omega*C1)+omega*L1), z3;
              z3=z1/z2;

              printf("%4.1e\t%4.1e\t%4.1e\t%4.1e\t%4.1f\n", f,
              log10(f),abs(z3),log10(abs(z3)),arg(z3)*180.0/M_PI);

              fprintf(puffer,"%4.1e\t%4.1e\t%4.1e\t%4.1e\t%4.1f\n", f,
              log10(f),abs(z3),log10(abs(z3)),arg(z3)*180.0/M_PI);
        }
        fclose(puffer);
}
```

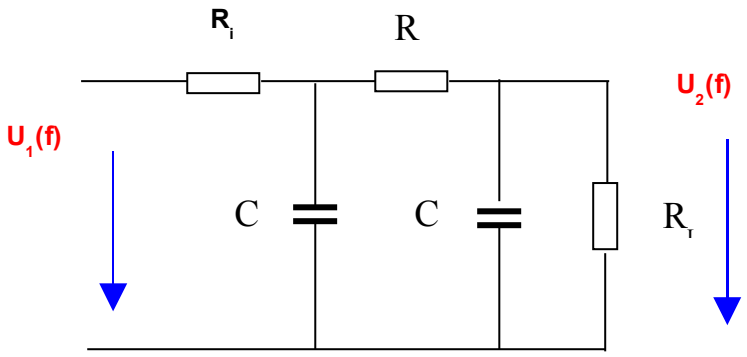
Die komplexen Widerstände lassen sich unmittelbar als komplexe Zahlen schreiben. Die über die Bauelemente und die Frequenz variablen Real- und Imaginärteile werden als reelle Variable in die Struktur der komplexen Zahlen eingefügt, siehe `complex <float> z1(R1,0),z2(R1,-1/(omega*C1)+omega*L1), z3;`

Als Ergebnis erhält man eine Wertetabelle. Die Frequenz f und und der Betrag abs(z3) des Frequenzganges sind wegen der weiten Wertebereiche zusätzlich logarithmisch (Zehner-Logarithmus) angegeben. Die Wertetabelle wird in die Datei „Frequ_gang“ geschrieben und über Excel als Betrags- und Phasendiagramm dargestellt. Man erkennt die Resonanzwirkung der Anordnung, die sich im Maximum des Betrags und als Nullwert des Phasenwinkels zeigt.





Ein anderes Beispiel, bei dem zusätzlich noch die Real- und Imaginärteile des komplexen Frequenzgangs für die so genannte Ortskurve F (j) verwendet werden, liefert die Anordnung eines Glättungsfilters, wie man es zur Verminderung des „Restbrumms“ von Gleichspannungsnetzteilen einsetzt. Es verringert in Verstärkeranlagen das lästige Brummgeräusch, sofern das Netzteil der Übeltäter ist. Schaltbild und Formel sehen wie folgt aus:



$$frg(j\omega)=\frac{U_2(f)}{U_1(f)}=\frac{1}{1-(\omega C)^2 R\cdot R_i+j\omega C(2R_i+R)} \quad , \quad für \quad R_L<<R$$

Der Lastwiderstand R_L ist wegen $R_L >> R,Ri$ vernachlässigt, was sich nicht entscheidend auf das Ergebnis auswirkt (man sollte diese Behauptung überprüfen und das Programm entsprechend erweitern.....). Das Programm macht wieder von der Rechnung mit komplexen Zahlen Gebrauch:

```
//komplexe Zahlen am Beispiel des Frequenzganges eines Glättungsfilters
//Last RL ist vernachlässigt
//20.06.2004

#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <complex>

void main()
{
      double f,omega,R,Ri,RL,C,L;
      FILE *puffer;
      pufer=
      fopen("C:\\Daten\\C_Daten\\Frequ_gang_glaettungsfilter_ohne_Last.txt","w
      ");
      using namespace std;
      Ri=2;//Ohm
      R=20;//Ohm
      //RL=100;//Ohm
      C=0.01;//10000 Mikro-Farad
      printf("f
      [Hz]\t\tlog(f)\t\tBetrag(z3)\tlog(Betrag(z3))\tPhase(z3)\n\n");
      for(f=0.1;f<200;f=f*1.1)
      {
            omega=2*M_PI*f;
            //ohne Lastwiderstand RL
            complex <double> z1(1,0), z2(1-
            (omega*C*omega*C)*R*Ri,omega*C*(2*Ri+R)),frg;
            frg=z1/z2;
            printf("%4.1e\t%4.1e\t%4.1e\t%4.1e\t%4.1f\t%4.1e\t%4.1e\n", f, lo-
            g10(f),abs(frg),log10(abs(frg)),arg(frg)*180.0/M_PI,
            real(frg),imag(frg));
```

```
            fprintf(puffer,"%6.4e\t%6.4e\t%6.4e\t%6.4e\t%6.4f\t%6.4e\t%6.4e\n"
            , f,
            log10(f),abs(frg),log10(abs(frg)),arg(frg)*180.0/M_PI,real(frg),im
            ag(frg));
      }
      fclose(puffer);
}
```

Als Ergebnis lassen sich 3 in der Elektrotechnik oft gebrauchte Diagramme erzeugen:

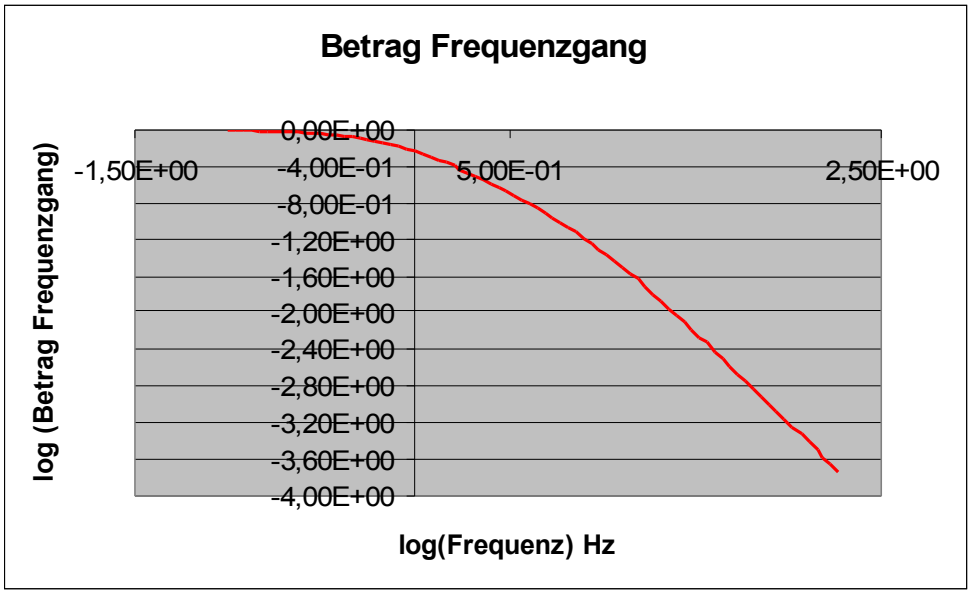
- Der Betragsverlauf `abs(frg)` des Frequenzganges als Funktion der Frequenz (logarithmische Darstellung)
- Der Phasenwinkelverlauf `arg(frg)` des Frequenzganges als Funktion der Frequenz
- Die Ortskurve `frg(j)` mit der Kreisfrequenz als Parameter. Hierfür werden Real- und Imaginärteil von `frg` benötigt.

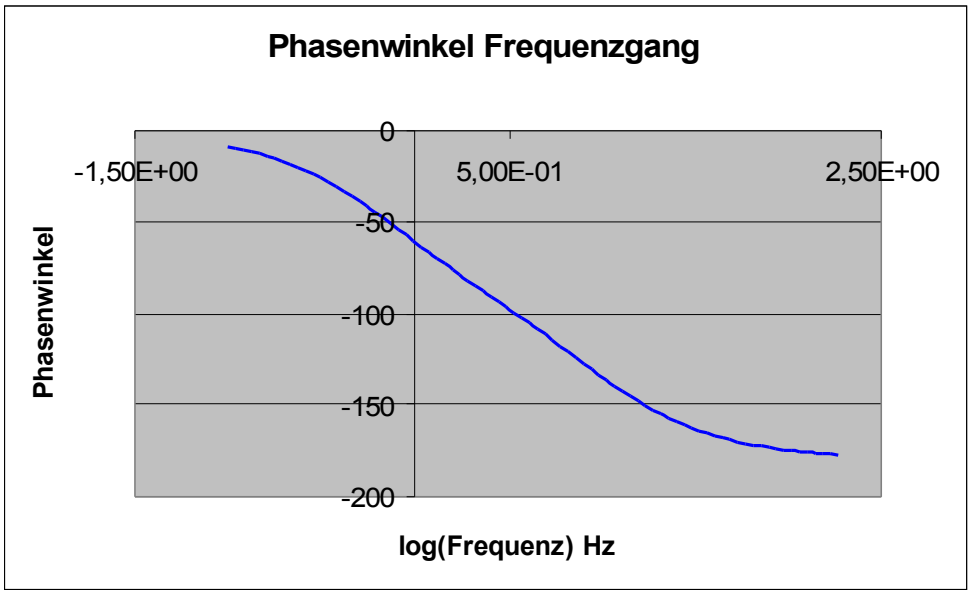
Für die Verwendung von Excel wird die Wertetabelle für alle erforderlichen Werte gemeinsam erzeugt, die Werte für die3 Diagramme können daraus zugeordnet werden.

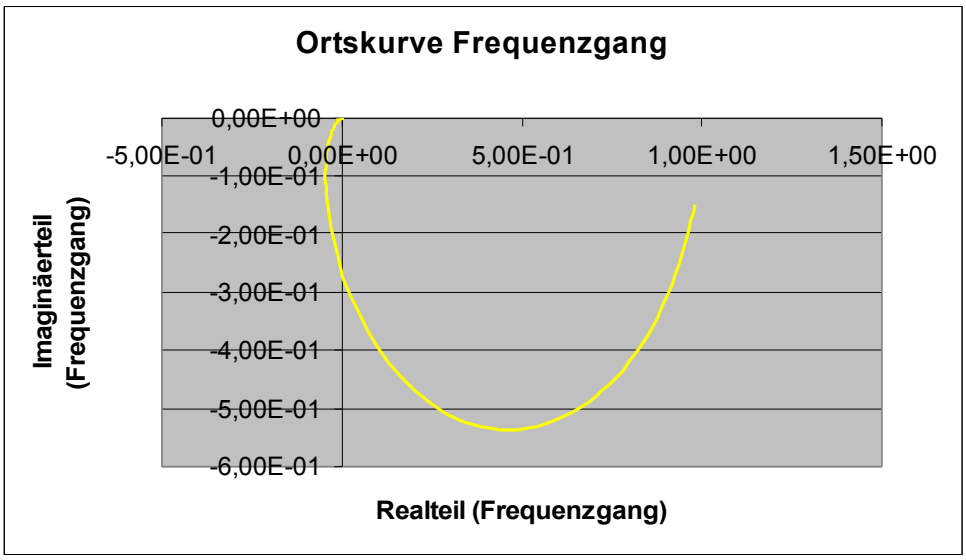
Man erkennt am Ergebnis, dass bei der gewählten Dimensionierung der Bauteile für 50 Herz-Schwingungen eine Dämpfung um den Faktor 0.0025 erfolgt.

Der Vorzug dieses Vorgehens liegt darin, dass man sich zunächst an einem einfachen Rechenmodell über das prinzipielle Verhalten einer solchen Filterschaltung klar werden kann – und dabei auch eine ausgezeichnete Rückkopplung zum eigenen Verständnis der physikalisch-technischen Verhältnisse erhält. Die Diagramme müssen ja im Einklang mit den elektrischen Eigenschaften der Bauelemente sein, woraus sich auf den erwarteten Kurvenverlauf schließen lässt.

Das einfache Modell gibt vielleicht die wirklichen Verhältnisse nicht genau genug wieder. Es kann dann aber leicht erweitert werden, etwa durch parallele Verlustwiderstände an den Kondensatoren, um die dielektrischen Effekte zu erfassen. Die Komplexität der Anordnung wird dann zwar wachsen, da man aber schrittweise von einfachen Anordnungen ausgehen kann, verkleinert sich das Risiko, völlig im "Nebel" zu verschwinden.







11.12. Der Aufbau eigener #include–Bibliotheken

Wenn immer wieder verwendete Funktionen nach dem **#include**-Teil integriert werden müssen, bietet es sich an, diese Funktionen als Quellcode in eine .txt-Datei zu speichern. Diese kann dann über die Direktive **#include „Pfadname“** im #include-Teil aufgeführt werden.

Im vorhergehenden Programmbeispiel wurde das genutzt. Der #include –Teil enthält die Bibliotheken

```
#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
```

```
#include <malloc.h>
#include "C:\\Daten\\C_Daten\\funktionen_1.txt"
```

Die Letzte darunter mit dem Namen **funktionen_1.txt** stellt den Quellcode von 4 Funktionen bereit:

```
int skalarprodukt(int n, int vektor1[], int vektor2[])
{
      int i, k;
      k=0;
      for (i=0;i<n;i++)k=k+vektor1[i]*vektor2[i];
      return(k);
}

float mittelwert (int n, float zuf[])
{
      int i;
      float xh;
      xh=0;
      for (i=0;i<n;i++)
      xh=xh+zuf[i];
      xh=xh/(float)n;
      return(xh);
}

float effektivwert (int n, float zuf[])
{
      int i;
      float xh;
      xh=0;
      for (i=0;i<n;i++)
      xh=xh+zuf[i]*zuf[i];
      xh=sqrt(xh/(float)n);
      return(xh);
}

float normfeld (int n, int rmax, float zuf[])//
{
      int i;
      for (i=0;i<n;i++)
      {
            if(zuf[i]==0) zuf[i]=0.000000001;
            zuf[i]=zuf[i]/(float)rmax;
      }
      return(*zuf);
}
```

Man muss zur Verwendung nur darauf achten, dass bei der **#include** –Anweisung der korrekte Pfadname erscheint.

11.13. Die Verwendung von struct-Elementen

Mit der **struct**-Deklaration kann man Variablen gleichen oder verschiedenen Typs als Datensatz unter einem Namen zusammenfassen und so nicht nur die Übersichtlichkeit verbessern, sondern auch nützliche Funktionalitäten realisieren. Ein Beispiel: bei der Analyse statistischer Daten müssen oft verschiedene Kenngrößen berechnet werden. Nehmen wir

n	: Anzahl aller Datenelemente
mittelwert	: arithmetischer Mittelwert
quadratischer_mittelwert	: quadratischer Mittelwert
effektivwert	: Effektivwert = Quadratwurzel aus dem quadratischen Mittelwert
standardabweichung	: Mittelwert der Quadratsumme von (Einzelwerten – Mittelwert)

Dieser Datensatz besteht aus 5 Einzelwerten, nämlich einem Ganzzahlwert n und 4 Gleitpunktwerten.

Werden diese Größen in einem Programm gebraucht, so kann man sie einzeln deklarieren und verarbeiten. Da sie aber logisch zusammengehören, wäre auch die Zusammenfassung als „Super-Variable“ in

```
struct statistik
{
      int n;
      double mittelwert;
      double quadratischer_mittelwert;
      double effektivwert;
      double standardabweichung;
}
```

nützlich. Diese wird wie ein eigener neuer Variablentyp behandelt. In einem Programm kann man sie z. B. vor dem **main**-Teil deklarieren und dort sowie in Funktionen gebrauchen. Dabei ist auch die Typ-Zuweisung wie bei `int i` als `struct statistik` zulässig, die Rückgabe des Wertes der Super-Variablen erfolgt wie bei einfachen Variablen über die return-Funktion mit `return (statistik)`.

Das folgende Programm nutzt das **struct** – Konzept. Nach der Einbindung der notwendigen Bibliotheken folgt die Deklaration der Super-Variablen *statistik* . Anschließend wird die Funktion `statistikdaten` deklariert. Sie erhält den Variablentyp `struct statistik` und übernimmt mehrere Parameter, u.a. auch ein Feld mit Zufallszahlen, zu dem die interessierenden Statistikdaten berechnet werden sollen. Die return-Funktion übergibt schließlich die in der Funktion berechneten Werte der Supervariablen `statistik` an den Funktionsnamen *statistikdaten*.

Im **main**-Teil werden zwei Zufallsfolgen *zufall* mit n = 10 und n = 10000 erzeugt. Die statistischen Kenngrößen werden berechnet und ausgegeben. Der Berechnungsteil ließ sich mit dem **struct**-Konzept kompakt halten (wobei der hier gezeigte Weg nicht unbedingt bereits der optimale ist).

Hier der Quellcode:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

struct statistik
{
      int n;
      double mittelwert;
      double quadratischer_mittelwert;
      double effektivwert;
      double standardabweichung;
};

struct statistik statistikdaten(statistik stat, int n, double zuf[])
{
      int i;
      double xh, mxh, quadmxh, stdabwxh;
      mxh=0;
      quadmxh=0;
      stdabwxh=0;
```

```
        for(i=0;i<n;i++)mxh=mxh+zuf[i];
        mxh=mxh/n;
        for(i=0;i<n;i++)quadmxh=quadmxh+zuf[i]*zuf[i];
        quadmxh=quadmxh/n;
        for(i=0;i<n;i++)
        {
              xh=(zuf[i]-mxh);
              stdabwxh=stdabwxh+xh*xh;
        }
        stat.n=n;
        stat.mittelwert=mxh;//arithmetischer Mittelwert
        stat.quadratischer_mittelwert=quadmxh;//Mittelwert der Quadrate
        stat.effektivwert=sqrt(stat.quadratischer_mittelwert);//Wurzel aus dem
                                                              // Mittelwert
                                                              //der Quadrate
        stat.standardabweichung=sqrt(stdabwxh/(n-1));//Wurzel aus dem Mittelwert
                                                     // der Quadrate von
                                                     //(Einzelwerte-Mittelwert)
        return(stat);
}

void main (void)
{
        int i, n;
        double *zufall;

        struct statistik statistik_1, statistik_2;
        n=10;

        zufall=(double*) malloc(n*sizeof(double));

        srand ((unsigned) time(NULL));
        for (i=0;i<n;i++) zufall[i]=(double) rand()/RAND_MAX;
        statistik_1 = statistikdaten(statistik_1,n,zufall);

        printf("Die Statistikdaten des Zufallsfeldes zufall_1 sind:\n");
        printf("Anzahl n der Datenelemente: %d\n",n);
        printf("Mittelwert % 6.4f\n",statistik_1.mittelwert);
        printf("quadr. Mittelwert % 6.4f\n",statistik_1.quadratischer_mittel-
        wert);
        printf("Effektivwert % 6.4f\n",statistik_1.effektivwert);
        printf("Standardabweichung % 6.4f\n",statistik_1.standardabweichung);

        printf("\n");
        n=10000;
        zufall=(double*) malloc(n*sizeof(double));
        for (i=0;i<n;i++) zufall[i]=2.0*1.75*((double) rand()/RAND_MAX-0.5);
        statistik_2 = statistikdaten(statistik_2,n,zufall);

        printf("Die Statistikdaten des Zufallsfeldes zufall_2 sind:\n");
        printf("Anzahl n der Datenelemente: %d\n",n);
        printf("Mittelwert % 6.4f\n",statistik_2.mittelwert);
        printf("quadr. Mittelwert % 6.4f\n",statistik_2.quadratischer_mittel-
        wert);
        printf("Effektivwert % 6.4f\n",statistik_2.effektivwert);
        printf("Standardabweichung % 6.4f\n",statistik_2.standardabweichung);
        printf("\n");
}
```

.... und die Ausgabe:

```
Die Statistikdaten des Zufallsfeldes zufall_1 sind:
Anzahl n der Datenelemente: 10
Mittelwert  0.4822
quadr. Mittelwert  0.3146
Effektivwert  0.5609
Standardabweichung  0.3020

Die Statistikdaten des Zufallsfeldes zufall_2 sind:
Anzahl n der Datenelemente: 10000
Mittelwert  0.0047
quadr. Mittelwert  1.0058
Effektivwert  1.0029
Standardabweichung  1.0030

Press any key to continue
```

Es gibt eine Fülle von weiteren Anwendungen dieses Konstruktes, daher lohnt es sich, solche Möglichkeit im Hinterkopf zu behalten.

12. Hinweise zur Simulation dynamischer Systeme

Eine häufige Aufgabe für Ingenieure ist die Analyse des zeitlichen Verhaltens technischer Systeme bei Änderungen der Einwirkungsgrößen. Oft zitiert wird hierbei das Beispiel des schwingenden Feder-Masse-Systems, das sich in seiner elementarsten Form sehr einfach beschreiben lässt, aber in komplexeren Anordnungen (Fahrwerksdynamik) hoch anspruchsvolle Abläufe zeigen kann. Andere Ausprägungsformen sind Schwingkreise, elektronische Filter, chemische Reaktionssysteme, elektrische Antriebe und vieles mehr. Allen gemeinsam ist der darin wirkende zeitliche Austausch zwischen verschiedenen Energieformen, beim Feder-Masse-System zum Beispiel zwischen der potentiellen Feder- und der kinetischen Massen-Energie. Um diese Austauschvorgänge technisch zu beschreiben, stellt man aus den physikalischen Gegebenheiten die Differentialgleichung auf und kann aus der Lösung die Variablen darstellen.

Nur in verschwindend wenigen Fällen ist die analytische Lösung dieser Differentialgleichungen möglich. Es gibt aber gute numerische Verfahren, die solche Lösungen liefern. Einige sind unter den Namen „Runge“, „Runge-Kutta“ oder „Kutta-Merson“ bekannt. Am Beispiel des schon zitierten Feder-Masse-Systems wird ein Simulationsergebnis gezeigt, aus dem der zeitliche Wege- und Geschwindigkeitsverlauf der Masse hervorgeht, wenn sie aus dem Ruhezustand mit der Kraft F nach unten ausgelenkt und dann losgelassen wird.

Um zu einer solchen Simulation zu kommen, ist das Durchgehen folgender Schritte hilfreich:

a) Skizzieren der physikalischen Anordnung

b) Mathematische Beschreibung der interessierenden Größen

c) Aufstellen der Gleichgewichts- oder Energiebedingungen, aus denen sich Differentialgleichungen n-ter Ordnung (DGLs) ergeben. In vielen Fällen erhält man gewöhnliche lineare oder nichtlineare Differentialgleichungen und nur diese werden wir hier betrachten. In anderen Fällen ergeben sich partielle Differentialgleichungen, deren Weiterbehandlung allerdings aufwendiger ist.

d) Skizzieren eines Blockdiagramms als Abstraktion der physikalischen Anordnung

e) Umwandeln der DGLs n-ter Ordnung in ein System von n DGLs 1. Ordnung.

f) Vorgeben von Anfangsbedingungen für die physikalischen Größen zum Zeitpunkt t = 0.

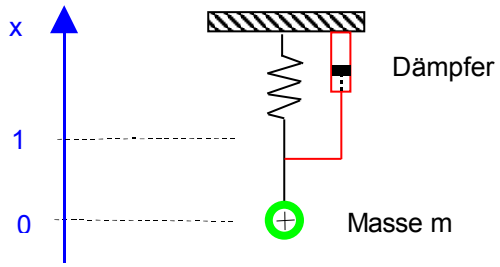
g) Bestimmen der Eigenwerte des DGL-Systems. Der Kehrwert der Integrations-Schrittweite h beträgt für numerisch stabile Lösungen etwa das 5- bis 10-fache des betragsgrößten Eigenwerts. Bei nichtlinearen DGL-Systemen muß man die Eigenwerte bei jedem Integrations-Schritt um den temporären Arbeitspunkt neu berechnen.

h) Lösen des DGL-Systems durch numerische Integration

i) Grafische Darstellung der Lösung

Am Beispiel des oben genannten zeitlichen Verhaltens eines gedämpften Feder-Masse-Systems werden diese Schritte jetzt durchlaufen:

a) **Skizzieren der physikalischen Anordnung**



b) Mathematische Beschreibung der interessierenden Größen

An der Masse m greifen bei Berücksichtigung der eingezeichneten Koordinatenrichtung folgende Kräfte an, wobei die augenblickliche Bewegungsrichtung in die positive x-Richtung zeigen soll:

-	Schwerkraft	:	$-m \cdot g$	nach unten (negativ)
-	äußere Zugkraft	:	$-F$	nach unten (negativ)
-	Trägheitskraft	:	$-m \cdot x''$	nach unten, entgegengesetzt der Bewegung
-	Dämpfungskraft	:	$-c_D \cdot x'$	nach unten, entgegengesetzt der Bewegung
-	Federkraft	:	$+(-c_F) \cdot x = -c_F \cdot x$	nach oben, aber Federkonstante negativ *)

*) weil bei Zug die Feder gedehnt wird (negative Wegrichtung, die Federkraft aber nach oben weist (positiv))

c) Aufstellen der Gleichgewichts- oder Energiebedingungen

Summe aller Kräfte im Gleichgewicht:

$$-m \cdot x'' - c_D \cdot x' - c_F \cdot x - m \cdot g - F = 0$$

Division durch m und Umstellen:

$$x'' + \frac{c_D}{m} \cdot x' + \frac{c_F}{m} \cdot x = x'' + a_1 x' + a_0 x = g + \frac{F}{m} = y$$

Es hat sich also eine lineare DGL 2. Ordnung ergeben.

d) Skizzieren eines Blockdiagramms als Abstraktion der physikalischen Anordnung

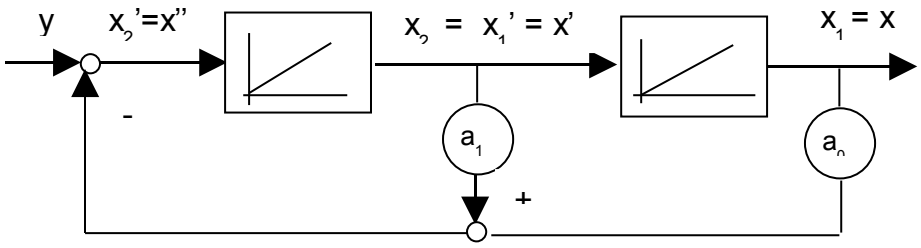
Die Wirkung der beiden Speicher für die kinetische und die potentielle Energie lässt sich mit Hilfe von Integratoren in einem Blockdiagramm darstellen (die Masse summiert und speichert unter dem Einfluss der Beschleunigung die kinetische Energie, die Feder durch Dehnen oder Stauchen die potentielle Energie).

Es bedeuten

$x_1 = x$: Lagekoordinate der Masse

$x_1' = x_2$: Geschwindigkeitskoordinate der Masse (Ableitung des Weges nach der Zeit)

x_2': Beschleunigung der Masse (Ableitung der Geschwindigkeit nach der Zeit)



e) Umwandeln der DGLs n-ter Ordnung in ein System von n DGLs 1. Ordnung.

Die Koordinaten x_1 und x_2 heissen auch Zustandskoordinaten, da sie den Energiezustand des Systems wiedergeben.

Es gilt, wie dem Blockdiagramm zu entnehmen ist: $x' = x_1'$ sowie $x'' = x_1'' = x_2'$

oder in Zustandskoordinaten als System von 2 gekoppelten Differentialgleichungen 1-ter Ordnung ausgedrückt, wie es sich für die Formulierung in Programmen besser eignet:

$$x_1' = x_2$$

$$x_2' = -a_0 x_1 - a_1 x_2 + y$$

Beide Darstellungen sind äquivalent. Man kann also umgekehrt durch Einführung von Zustandsvariablen ein Differentialgleichungssystem n-ter Ordnung in ein System von n gekoppelten Differentialgleichungen 1-ter Ordnung umwandeln.

f) Vorgeben von Anfangsbedingungen für die physikalischen Größen zum Zeitpunkt t = 0

Wir nehmen an, dass die Masse zum Zeitpunkt t = 0 aus dem Gleichgewichtszustand bei x = 1 zum Punkt x = 0 ausgelenkt wurde, also $x(t=0) = 0$, und sich in Ruhe befindet, $x'(t=0) = 0$.

g) Bestimmung der Eigenwerte

Aus der charakteristischen Gleichung des DGL-Systems lassen sich die Eigenwerte ermitteln. Im vorliegenden Fall sind es die Nullstellen des charakteristischen Polynoms

$$z^2 + a_1 z + a_0 = 0$$

mit den beiden Lösungen $z_{1,2} = -\frac{a_1}{2} \pm \sqrt{\left(\frac{a_1}{2}\right)^2 - a_0}$

Die Integrations-Schrittweite h wird daher als

$$h = \frac{1}{10 \cdot \max(z_{1,2})}$$

gewählt.

h) Lösen des DGL-Systems durch numerische Integration

Für die numerische Lösung benötigen wir

- eine Funktion, welche das Differentialgleichungssystem beschreibt,
- eine Funktion, welche dieses Differentialgleichungssystem integriert (=löst)

Im folgenden Programm haben diese beiden Funktionen die Namen **dgl** und **RUNGEKUTTA**. Sie sind für die Formulierung und Lösung auch großer und nichtlinearer Differentialgleichungssysteme geeignet, wenn man die Anforderungen an die Schrittweite beachtet, siehe oben.

In unserem Fall lösen wir das System für die Sprungantwort (y =0 für t <0 und y = 1 für t <=0), die dem oben genannten Vorgang der Massenauslenkung und des Loslassens bei t = 0 entspricht.

```
// Rungekutta zum Lösen von Differentialgleichungen


#include <iostream>
#include <stdlib.h>


double dgl(int n, double y, double a[], double x[],double xp[])
{
    int i;
    double xh=0;
    for (i=0;i<n-1;i++) xp[i]=x[i+1];
    for (i=0;i<n;i++) xh=xh-x[i]*a[i];
    xp[n-1]=xh+y/a[0];
    //for (i=0;i<n;i++)
    //printf("% 6.4f\t",xp[i]);
    //printf("\n");

    return(*xp);
}


double RUNGEKUTTA (int n, double h, double y, double a[], double x[], double
xp[])
{
   int i;
   double *xh,*k1,*k2,*k3,*k4;
   xh=(double*)malloc (n*sizeof(double));
   k1=(double*)malloc (n*sizeof(double));
   k2=(double*)malloc (n*sizeof(double));
```

```
   k3=(double*)malloc (n*sizeof(double));
   k4=(double*)malloc (n*sizeof(double));
   *xp=dgl(n, y, a, x,xp);
   //for (i=0;i<n;i++)
   // printf("M1% 6.4f\t",xp[i]);
    //printf("\n");
   for (i=0;i<n;i++)
   {
     k1[i]= h * xp[i];
     xh[i]= x[i]+k1[i]/2.0;
   }
   *xp=dgl(n,y,a,xh,xp);
   for (i=0;i<n;i++)
   {
     k2[i]= h * xp[i];
     xh[i]= x[i]+k2[i]/2.0;
   }
   *xp=dgl(n,y,a,xh,xp);
   for (i=0;i<n;i++)
   {
     k3[i]= h * xp[i];
     xh[i]= x[i]+k3[i];
   }
   *xp=dgl(n,y,a,xh,xp);
   for (i=0;i<n;i++)
   {
     k4[i]= h * xp[i];
     x[i] = x[i]+(k1[i] + k2[i]*2.0 + k3[i]*2.0 + k4[i])/6.0;
   }
   return (*x);
}

int main()
{
  int i,j,n;
  double y, h,a[2],*x,*xp;
  FILE *puffer;
  puffer=fopen("C:\\DATEN\\C_DATEN\\PT2.txt","w");
  n=2;
  x=(double*) malloc(n*sizeof(double));
  xp=(double*) malloc(n*sizeof(double));
  x[0]=0;
  x[1]=0;
  y=1;
  a[0]=1;
  a[1]=0.2;
  h=0.1;
  printf("\n");
  for (j=0;j<200;j++)
  {
    *x=RUNGEKUTTA(n,h,1.0,a,x,xp);
    printf("%4.2f\t",j*h);
    fprintf(puffer,"%4.2f\t",j*h);
    for (i=0;i<n;i++)
    {
        printf("% 6.4f\t",x[i]);
        fprintf(puffer,"% 6.4f\t",x[i]);
    }
    printf("\n");
    fprintf(puffer,"\n");
  }
  fclose(puffer);
```

```
  system("PAUSE");
}
```

i) Grafische Darstellung der Lösung

Das Ergebnis hängt von der Wahl der beiden Faktoren a_0, a_1 ab. Diese enthalten die Zeitkonstanten, bzw. die Eigenwerte des Systems in der Form von Masse, Federkonstante und Dämpfungskonstante. Bei der hier getroffenen Wahl

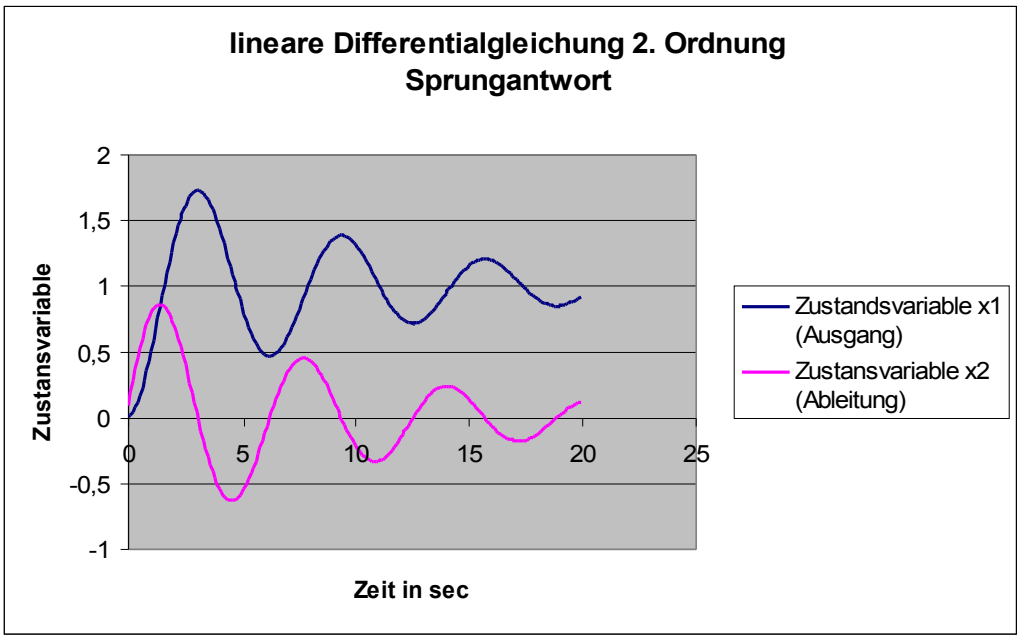
$$a_0 = 1$$

$$a_1 = 0.2$$

mit den konjugiert komplexen Eigenwerten

$$p_{1,2} = 0.1 \pm j\ 0.99$$

und dem Betrag ≈1 erhalten wir für h = 1/[10·abs(p)]= 0.1 s ein schwach gedämpftes Schwingungssystem:



Man kann nun andere Werte für die 3 physikalischen Parameter vorgeben und die Änderungen des jeweiligen zeitlichen Verhaltens beobachten. Selbstverständlich ist hier in komplexeren Fällen Vorsicht und Sorgfalt geboten, da das aufgestellte Modell ja nicht von vornherein richtig sein muss. Es ist also angebracht, fortwährend nach der Glaubwürdigkeit der Ergebnisse zu fragen und bei Widersprüchen nach den Gründen zu suchen, bis wieder Übereinstimmung mit experimentellen oder theoretisch erwarteten Ergebnissen erzielt wird. Genau hier zeigt sich der überragende Nutzeffekt für das Verstehen vor allem komplexer dynamischer Systeme!

Das Programm kann gut für andere und wesentlich aufwendigere Systeme erweitert werden, wobei an der Funktion **RUNGEKUTTA** keine Veränderungen notwendig sind und nur die spezifischen DGL-Systeme in **dgl** erfasst werden müssen

Viel Vergnügen und ebensoviel Erfolg!

Anhang

A1. Arithmetische Operatoren (geben Zahlenwerte zurück)

Operator	Wirkung	Beispiel
+	Addition	`c = a + b`
-	Subtraktion	`c = a - b`
*	Multiplikation	`c = a * b`
/	Division	`c = a/b`

%	Rest der Division =
	Modulo-Division
		`c = a % b`
		`a=7, b=4`
		`c = 7 % 4 = 3`

A2. Logik-Operatoren (geben Wahrheitswerte 0 oder 1 zurück)

Operator	Wirkung	Beispiel
<	kleiner	`a < b`
>	größer	`a > b`
= =	gleich (Vorsicht: **keine** Wertzuweisung wie bei a =b;)	`a = = b`
!=	ungleich	`a !=b`
<=	kleiner gleich	`a <= b`
=>	größer gleich	`a >= b`
&&	logisches UND	`a && b`
&	bitweises UND	`a & 1`
||	logisches ODER	`a || b`

A3. Formatangabe

Formatan-gaben	Wirkung	Beispiel
d	Dezimaldarstellung (Ganzzahlen)	printf("%d", i) → 13
i	Dezimaldarstellung (Ganzzahlen)	printf("%i", i) → 13
f	Gleitkommadarstellung	printf("% 4.2f", x) → -6.25
e	Exponentialdarstellung	printf("%4.2e", x) → -6.25e+00
o	Oktaldarstellung	printf("%o", i) → 76
x	Hexadezimaldarstellung	printf("%x", i) → 0d
c	Charakterdarstellung	printf("%c", i)
s	Zeichenkettendarstellung	printf("%s",text)
p	Hex-Darstellung für Zeiger	printf("%p",zeiger)

A4. Einige Schlüsselwörter (reserviert, nicht für Variablennamen verwendbar)

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

A5. Mathematische Funktionen (über den Index „Floating Point Support" der Hilfe-Funktion erreichbar)

Funktionsname	Wirkung	Beispiel
abs(x)	Absolutwert von x	y = abs(-1.34); y → 1.34
exp(x)	Exponentialfunktion	y = exp(2.1) ; y → 9.583
sin(x)	Sinus-Funktion x im Bogenmass	y = sin(M_PI/6) ; y → 0.5000
cos(x)	Cosinus-Funktion x im Bogenmass	y = cos(M_PI/3) ; y → 0.5000
log(x)	natürlicher Logarithmus	y = log(M_E); y → 1.0
log10(x)	Zehner-Logarithmus	y = log10(1000) y → 3
floor(x)	kleinste ganze Zahl kleiner x	y = floor(-9.468) y → -10
ceil(x)	größte ganze Zahl größer x	y = ceil(-2.74); y → -2

Funktionsname	Wirkung	Beispiel
rand()	erzeugt Zufallszahlen zwischen 0 und RAND_MAX (=32765)	
srand()	legt den Startwert für die Zufallszahlenerzeugung fest	`srand((unsigned) time(NULL))` → erzeugt bei jedem Aufruf eine andere Folge
pow(x,y)	Potenz y zu x → x^y	`y= pow( 5, 3);` → 125
abs(x) abs(z1)	Betrag von x Betrag einer komplexen Zahl z1	y=abs(-5.7); y → 5.7 y = abs(z1(2,3)) ; y → 13
_rotr(x,i)	Rotation eines Bytes x oder eines Wortes x um i Bits nach rechts. Ein Bit-weises AND wird durch das Zeichen & bewirkt, nicht durch &&.	if (_rotr(x,5) & 1) y=1;
_rotl(x,i)	Rotation eines Bytes x oder eines Wortes x um i Bits nach links. Ein Bit-weises AND wird durch das Zeichen & bewirkt, nicht durch &&.	if (_rotl(x,2) & 1) y=-1;

A6. Funktionen zum Schreiben und Lesen von Files

Funktionsname	Wirkung	Beispiel
`FILE *puffer`	Deklaration eines Datenpuffers	`FILE *zwischenpuffer;`
`fopen("Dateiname", w")`	weist dem Puffer `puffer` eine Datei zum **Schreiben** zu (w → write)	`zwischenpuffer =` `fopen(„C:\\text.txt","w");`
`fopen(„Dateiname", „r")`	weist dem Puffer puffer eine Datei zum **Lesen** zu (r → read)	`zwischenpuffer =` `fopen(„C:\\text.txt","r");`
`fprintf(puffer,"Formatstring", x, y)`	Schreibt die Werte x, y mit dem durch Formatstring gegebenen Format in den Puffer	`fprintf(zwischenpuffer,"%d\n", x);`
`fscanf(puffer,"Formatstring", x, y)`	Liest die Werte x, y mit dem durch Formatstring gegebenen Format aus der Datei in den Puffer	`fscanf(zwischenpuffer,"%d %f",x,y);`
`fclose(puffer)`	leert den Pufferspeicher und schließt ihn	`fclose(zwischenpuffer);`

A7. Komplexe Zahlen

Funktionsname	Beschreibung	Beispiel
`#include <complex>` (**ohne** `.h`)	Bibliothek	
`using namespace std;`	damit gelten für komplexe Zahlen die Standardregeln für die Namensvergabe	

`complex <double> ze(1,0),zi(0,1),z1(x,y), z2;`	`Deklaration der Struktur für komplexe Zahlen.`
	`Achtung: Alle Operationen mit komplexen Zahlen dürfen nur mit komplexen Zahlen ausgeführt werden.`
	`z. B. wäre`
	`z2 = 1/z1;`
	`unzulässig, richtig ist`
	`z2 = ze/z1;`
	`da die Zahl 1 nicht die Struktur einer komplexen Zahl besitzt. Numerisch sind aber 1 und ze identisch mit der reelen Zahl 1.`
		`z2 = ze + zi;`

`real(z)`	Realteil einer komplexen Zahl z	`y = real(ze);` `y → 1`
`imag(z)`	Imaginärteil einer komplexen Zahl z	`y= imag(zi) ;` `y → 1`
`abs(z)`	Betrag der komplexen Zahl z	`y= abs(z2);` `y → sqrt(2)`
`arg(z)`	Winkel des komplexen Zeigers z im Bogenmass = `arctg ( real(z)/imag(z))`	`y= arg(z2);` `y → Pi/4 (entspricht 45 Grad)`

A8. Funktionen zur Zeichenkettenverarbeitung

Funktionsname	Wirkung	Beispiel
`char zeich[21]`	deklariert ein Zeichenfeld zeich mit 20 nutzbaren Indices, also zeich[0] ..... zeich[19], die Feldvariable zeich[20] ist für die Zeichenkettenende-Kennung reserviert	`char text[2001];` die Feldvariable `text` kann Zeichenketten mit bis zu 2000 Zeichen aufnehmen
`strlen(text)`	Ermittelt die Anzahl der tatsächlich vorhandenen Zeichen in einem char-Feld (**nicht** die deklarierte Anzahl)	wenn im Feld text nur eine indizierte Variable mit einem Zeichen belegt ist, z. B. `text[0] ='f';,` dann ist mit `k=strlen(text);` k → 1

`sprintf(text, "Formatstring", Variable)`	Speichert die mit Formatstring formatierte Zahlenvariable `Variable` als zeichenkette in `text`	`char text[5];`
		`float x=3.456;`
		`sprintf(text,"%4.2f", x);`
		`text` → `3.46`

`strcat(str1, str2)`	hängt die Zeichenkette str2 an str1	`str1 = "abcd";`
		`str2 ="efg":`
		`strcat(str1, str2);`
		`str1` → `"abcdefg"`

`fgets(charfeld, anzahl, puffer)`	liest aus dem Datenpuffer puffer die angegebene Anzahl anzahl von Zeichen in das Feld charfeld	`fgets(cha rfeld,anzahl, puffer)`
`fputs(charfeld, anzahl, puffer)`	schreibt aus dem Feld charfeld die angegebene Anzahl anzahl in den Datenpuffer puffer	`fputs(charfeld,anzahl, puffer)`