

Computerphysik (physik441)

Dozenten: Tom Luu (t.luu@fz-juelich.de)

Andreas Wirzba (a.wirzba@fz-juelich.de)

Andreas Nogga (a.nogga@fz-juelich.de)

Übungen: Marcus Petschlies

(marcus.petschlies@hiskp.uni-bonn.de)

Ort: ~~Hörsaal I, Physikalisches Institut~~ (bis auf weiteres Online)

Zeit: ~~Mi 10:15 – 12:00 Uhr~~

~~Fr 9:15 – 10:00 Uhr~~

LP: 6

ecampus Link: https://ecampus.uni-bonn.de/goto_ecampus_crs_1661906.html

Die Vorlesungen wird bis auf weiteres als Podcast auf eCampus hochgeladen.
Fragen werden jeweils mittwochs und freitags online beantwortet.
Außerhalb dieser Zeiten kann das Forum für Fragen benutzt werden.

Übungen und Benotung

- Anmelde und Abmeldefrist bei BASIS: **27.5.2020 bzw. 03.06.2020**
- **Benotung erfolgt auf Basis der Hausaufgaben**
- Erstmal 12 Übungstermine:
Mo 10-12, 13-15, 15-17, Di 8-10, 13-15, 15-17 Mi 8-10, 13-15, 15-17, Do 8-10, 13-15, 15-17
- Anmeldung über eCampus ab **Montag, den 6.4.2020 um 08:00 Uhr**
- ~~Übungen finden im CIP Pool der Physik (AVZ 1, Endenicher Allee 11–13, Raum 1.008) statt~~
Online Zugriff auf die Linux Rechner wie unter
<https://www.pi.uni-bonn.de/lehre/cip-pool/homepage-des-physik-cip-pools>
beschrieben.
- Jede Woche freitags neues Übungsblatt für "Anwesenheitsübungen" (in eCampus)
- außerdem 6 Hausaufgaben
 - **Ausgabe** an Freitag **24.4 / 8.5 / 22.5 / 5.6 / 19.6 / 3.7** in eCampus
 - zweiwöchige Bearbeitungszeit
 - Abgabe elektronisch (**eCampus, strikte deadline!!!**)
 - jeweils **20 Punkte** pro Blatt, Gesamtpunktzahl legt Note fest (50 % = bestanden)
 - Abgabe zu zweit möglich (außer bei einer HA, die individuell abgegeben werden muss)

Empfohlene Literatur:

Kerningham, Ritchie	Programmieren in C
Press, Teukolsky, Veterling, Flannery	Numerical Recipes in C
Stoer, Bulirsch	Numerische Mathematik 1 und 2
Koonin	Computational Physics
DeVries	Computerphysik
Kinzel	Programmierkurs für Naturwissenschaftler und Ingenieure
Tao Pang	An Introduction to Computational Physics
Schmid, Spitz, Löscher	Theoretische Physik mit dem Personalcomputer
Vesely	Computational Physics - An Introduction

Skript des C-Programmierkurses von 2015: Hüttenhain, Wallenborn

<http://www.ins.uni-bonn.de/teaching/vorlesungen/ProgKursWs15/ProgKursSkript.pdf>

Skript des C-Programmierkurses von 2019 von

B. Kostrzewa, F. Pittler, M. Ueding, C. Urbach <https://github.com/urbach/c-kurs>

1. Einführung: Computerphysik und numerische Methoden

Ziel der Vorlesung: Methoden kennenzulernen, die es erlauben, physikalische Probleme numerisch zu lösen

Warum?

Analytische Methoden reichen nicht aus, um alle in der Natur vorkommenden Prozesse zu verstehen

- Numerische Methoden sind auf eine sehr viel größere Zahl von Problemen anwendbar
- Die Anwendungsmöglichkeiten steigen mit der größer werdenden Geschwindigkeit der Computer

Dazu brauchen wir:

1. Vorstellung der grundlegenden Methode
→ Vorlesung
2. Implementierung der Methode in der Programmiersprache C
→ Vorlesung, Übungen
3. Anwendung der Methode in typischen Problemstellungen
→ Übungen, Hausaufgaben

2. Einführung in die Programmiersprache C

Es gibt viele Programmiersprachen: Warum C?

bester Zugriff auf den Computer: **Assembler**

- große Geschwindigkeit
- viel Programmieraufwand
- abhängig von der Architektur des Computers

einfachste Programmierung: “**Interpretersprachen**” **Python, Matlab, Mathematica,...**

- wenig Programmieraufwand, oft natürliche Formulierung des Problems
- unabhängig von der Architektur des Computers
- kleine Geschwindigkeit, schlechte Parallelisierung

Kompromiss aus Einfachheit und Zugriff: **Compilersprachen C, FORTRAN, ...**

- unabhängig von der Architektur des Computers
- große Geschwindigkeit, gute Parallelisierung
- Möglichkeiten die Architektur des Rechner auszunutzen
- mehr Programmieraufwand als Interpretersprachen



Compilersprachen haben sich für numerische
(und viele andere) Anwendungen durchgesetzt.

C (C++) ist dabei vermutlich die meist genutzte Sprache.

Ein (einfaches) C Programm

```
/* Datei: beispiel-2.1.c
   Datum: 12.4.2016 */

/* #include Anweisungen binden Erweiterungen ein (hier I/O) */

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

/* Funktion "main" wird bei Start des Programms ausgefuehrt */
int main() /* int definiert den Datentyp der Groesse,
              die "main" zurueck gibt */
{
    /* geschweifte Klammern begrenzen Programmblöcke
       hier werden alle Anweisungen der Funktion
       eingefasst */

    double x,y; /* x und y sind Speicherplaetze fuer Gleitkommazahlen */

    x=1.2;        /* 1.2 wird an Speicherstelle x gespeichert */

    printf("x= %7.2f \n",x);

    return 0;           /* Rueckgabe einer 0 an die aufrufende
                          Funktion (Betriebssystem) */
}      /* geschweifte Klammer zum Abschluss der Funktion */

/* Das Programm kann mit
   gcc beispiel-2.1.c -o beispiel-2.1
   kompiliert und dann mit
   ./beispiel-2.1
   ausgefuehrt werden.

   Ergebnis: x = 1.20      */
```

```
/* Datei: beispiel-2.1.c  
Datum: 12.4.2016 */
```

```
/* #include Anweisungen binden Erweiterungen ein (hier I/O) */
```

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

```
/* Funktion "main" wird bei Start des Programms ausgefuehrt */  
int main() /* int definiert den Datentyp der Groesse,  
            die "main" zurueck gibt */  
{ /* geschweifte Klammern begrenzen Programm  
    hier werden alle Anweisungen der Funktion  
    eingefasst */
```

C Programme sind Ansammlungen von “Funktionen”

“#include” statements für den Zugriff auf vordefinierte Funktionen

```
double x,y; /* x und y sind Speicherplaetze fuer Gleitkommazahlen */  
  
x=1.2; /* 1.2 wird an Speicherstelle x gespeichert */  
printf("x= %7.2f \n",x);
```

```
return 0; /* Rueckgabe einer 0 an die aufrufende  
           Funktion (Betriebssystem) */  
} /* geschweifte Klammer zum Abschluss der Funktion */
```

```
/* Das Programm kann mit  
   gcc beispiel-2.1.c -o beispiel-2.1  
   kompiliert und dann mit  
   ./beispiel-2.1  
   ausgefuehrt werden.
```

```
Ergebnis: x = 1.20 */
```

Die Funktion “main” wird beim Start des Programmes ausgeführt

Der Bereich, der zur Funktion “main” gehört, wird durch { } eingeschlossen.

Text innerhalb /* */ sind Kommentare, die für einen selber und andere (Tutoren) gut erklären sollten welches Ziel die Programmteile haben

```
/* Datei: beispiel-2.1.c
   Datum: 12.4.2016 */

/* #include Anweisungen binden Erwe
#include <stdio.h>
#include <stdlib.h>

/* Funktion "main" wird bei Start d
int main() /* int definiert den D
            die "main" zurueck
{
            /* geschweifte Klammer
               hier werden alle Ar
               eingefasst */

    double x,y; /* x und y sind Speicherplaetze fuer Gleitkommazahlen */
    x=1.2;        /* 1.2 wird an Speicherstelle x gespeichert */
    printf("x= %7.2f \n",x);

    return 0;          /* Rueckgabe einer 0 an die aufrufende
                        Funktion (Betriebssystem) */
}      /* geschweifte Klammer zum Abschluss der Funktion */

/* Das Programm kann mit
   gcc beispiel-2.1.c -o beispiel-2.1
   kompiliert und dann mit
   ./beispiel-2.1
   ausgefuehrt werden.

Ergebnis: x = 1.20      */
```

Funktionen beginnen mit Deklarationen (guter Stil)

Das weist den Compiler an Speicherplatz für Daten zu reservieren. Der Speicherplatz wird dann als

- **double, float - Gleitkommazahl**
- **int, long - ganze Zahl**
- **char - Zeichen, Zeichenkette**

interpretiert.

Deklarationen und auch Anweisungen werden durch ; abgeschlossen!

```
/* Datei: beispiel-2.1.c
   Datum: 12.4.2016 */

/* #include Anweisungen binden Erweiterungen ein (hier I/O) */

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

/* Funktion "main" wird bei Start des Programms ausgefuehrt */
int main() /* int definiert den Datentyp der Groesse,
              die "main" zurueck gibt */
{
    /* geschweifte Klammern begrenzen Programmblöcke
       hier werden alle Anweisungen der Funktion
       eingefasst */

    double x,y; /* x und y sind Speicherstellen */

    x=1.2; /* 1.2 wird an Speicherstelle x geschrieben */

    printf("x= %7.2f \n",x);

    return 0; /* Rueckgabe einer 0 an die aufrufende
                Funktion (Betriebssystem) */
}

/* Das Programm kann mit
   gcc beispiel-2.1.c -o beispiel-2.1
   kompiliert und dann mit
   ./beispiel-2.1
   ausgefuehrt werden.

Ergebnis: x = 1.20 */
```

**Zuweisung eines Wertes (hier Gleitkommazahl)
an Variable (Speicherplatz) x mit “=”**

**Aufruf der Funktion “printf” aus stdio.h
Argument 1: Zeichenkette mit Formatierung
Argument 2: Variable
druckt “x = <Wert der Variable>” auf den Bildschirm**

**return beendet die Funktion und gibt den
Wert des Arguments zurück
hier an das Betriebssystem**

```
/* Datei: beispiel-2.1.c
   Datum: 12.4.2016 */

/* #include Anweisungen binden Erweiterungen ein (hier I/O) */

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

/* Funktion "main" wird bei Start des Programms ausgefuehrt */
int main() /* int definiert den Datentyp der Groesse,
              die "main" zurueck gibt */
{
    /* geschweifte Klammern begrenzen Programmblöcke
       hier werden alle Anweisungen der Funktion
       eingefasst */

    double x,y; /* x und y sind Speicherplätze fuer Gleitkommazahlen */

    x=1.2; /* 1.2 wird an Speicherstelle x gespeichert */

    printf("x= %7.2f \n",x);

    return 0; /* Rueckgabe einer 0 an die aufrufende
                 Funktion (Betriebssystem) */
}

/* geschweifte Klammer zum Abschluss der Funktion */

/* Das Programm kann mit
   gcc beispiel-2.1.c -o beispiel-2.1
   kompiliert und dann mit
   ./beispiel-2.1
   ausgeführt werden.

Ergebnis: x = 1.20      */
```

Das Programm wird mit einem beliebigen Editor (“emacs”,...) geschrieben, dann kompiliert und kann dann ausgeführt werden.

Hier der Ablauf auf Linux auf den CIP Rechnern.

Datentypen

C hat eine Reihe vordefinierter Standarddatentypen

```
char ch,kette[100],*pch;
```

→ **char** reserviert Speicher für ein Zeichen oder eine Zeichenkette
(1 Byte pro Zeichen)

ch ist Speicherplatz für ein Zeichen

kette reserviert Speicherplatz für 100 Zeichen

pch kann die Adresse (einen Zeiger auf) ein(e) Zeichen(kette) speichern

Beispiele zum Gebrauch

```
ch='a';
```

Zeichen wird in **ch** gespeichert

```
scanf ("%s",kette);
```



Zeichenkette wird eingelesen
(von Tastatur) und in **kette** gespeichert.

```
pch = "Hallo";
```

Zeichenkette "Hallo" erzeugt und die
Adresse wird in **pch** gespeichert.

Beachten Sie, dass 'a' ≠ "a" = 'a' + Nullzeichen.

```
int n,ind[10][20];  
long ln,lind[10][20];
```

→ **int, long** reserviert Speicher für ein ganze Zahl (mit 4/8 Byte auf CIP Rechner)
d.h. für Werte zwischen -2.147.483.648,..., 2.147.483.647
bzw. zwischen -9.223.372.036.854.775.808,..., 9.223.372.036.854.775.807
(NB: **char** kann auch mit -128,...,127 interpretiert werden)

n,ln ist Speicherplatz für eine ganze Zahl
ind,lind reserviert Speicherplatz für 10x20 ganze Zahlen

Beispiele zum Gebrauch

```
n=1;  
  
ind[0][4]=5;
```



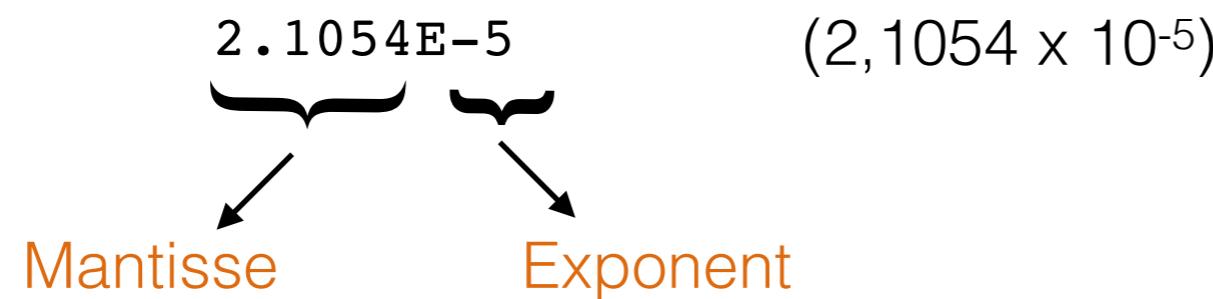
Zuweisung des Wertes 1 an Speicherplatz “n”
An Speicherposition [0x20+4] wird 5 gespeichert.
Indizes fangen bei 0 an und gehen bis 10-1 bzw. 20-1

Vorsicht: [10,20] wird als Index akzeptiert, aber wie 20 interpretiert !!!

```
float x,y;  
double dx,dy;
```

→ **float, double** reserviert Speicher für eine Gleitkommazahl mit 4 bzw. 8 Bytes.

allgemeiner Aufbau einer Gleitkomma (rationalen) Zahl



Es wird Speicherplatz für das Vorzeichen, die Mantisse und den Exponenten benötigt.
Typisch ist (auch auf CIP Rechnern)

float → 4 Bytes → etwa 6 signifikante Stellen im Mantissenbereich
-37,...,38 für Exponent

double → 8 Bytes → etwa 15 signifikante Stellen im Mantissenbereich
-307,...,308 für Exponent

weiter Typen existieren (bool, selbstdefinierte Typen, complex (in C++), ...)
→ C Literatur

Schleifen und bedingte Ausführung

```
/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehren solange i<n ist  3) am Ende i um 1 erhoehen (i++) und wieder zu 2 */
    {
        /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)           /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens 1x ausgefuehrt */

    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));        /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */
```

Schleifen ermöglichen auf einfache Weise ähnliche Anweisung vielfach auszuführen.

1) “for” Schleife

2) “while” Schleife

3) “do-while” Schleife

```

/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehrer */
    { /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)           /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;            /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens 1x ausgefuehrt */

    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;            /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));       /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */

```

Beachte:

Definition der zweiten Funktion "f"
zusätzlich math.h

for-Schleife: typisch für Durchlaufen eines Index
erster Parameter: setzen des Startwertes
zweiter Parameter: Bedingung die Schleife beendet
dritter Parameter: Operation zum Erhöhen des Index

```
/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehren solange i<n ist  3) am Ende i um 1 erhoehen (i++) und wieder zu 2 */
    {
        /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)           /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens 1x ausgefuehrt */

    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));        /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */
```

while-Schleife:
Parameter ist eine Bedingung, die Schleife beendet

```
/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehren solange i<n ist  3) am Ende i um 1 erhoehen (i++) und wieder zu 2 */
    {
        /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)        /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;        /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens einmal durchlaufen. */
    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;        /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));     /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */
```

do-while-Schleife:

Parameter ist eine Bedingung, die Schleife beendet Test am Ende der Schleife, d.h. die Schleife wird mindestens einmal durchlaufen.

Alle Schleifen benötigen **Bedingungen** also logische Operationen

Vergleiche:

`==, <=, >=, <, >, !=` → falsch/wahr, d.h 0 bzw. ≠0

logische Verkäpfungen:

`&&, ||, !` für “und”, “oder” und “nicht”

Beispiele: `(n<=1) || (n>10)`

`! ((a==1.5) && (b==2.0))`

Achtung: typischer Anfängerfehler ist “=” anstatt “==” zu benutzen.
Der C Compiler findet das in der Regel akzeptable, aber das Ergebnis ist anders!

Vergleiche von Gleitkommazahlen mit “==” sind ebenfalls irreführend

`1.5 != 1.499999999`

Bedingte Ausführung

if-Konstrukt ermöglicht die bedingte Ausführung von Programmteilen

```
/* Datei: beispiel-2.3.c    Datum: 12.4.2016 */

...
double x,y;
...

/* if-Anweisung */
if(fabs(x)<0.3)          /* Bedingung */
{
    y = 1+x+0.5*pow(x,2); /* Block wird ausgefuehrt falls Bedingung "wahr" */
}
else
{
    y=exp(x);           /* Block wird ausgefuehrt falls Bedingung "falsch" */
}

...
```

Das schließt die sehr kleine Einführung in C ab.
Weitere Konstrukte, arithmetische Operationen, mathematische Funktionen, ... werden wir in den Beispielen und Übungen kennenlernen.