

# Computerphysik (physik441)

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

Andreas Nogga ([a.nogga@fz-juelich.de](mailto:a.nogga@fz-juelich.de))

Deborah Rönchen ([d.roenchen@fz-juelich.de](mailto:d.roenchen@fz-juelich.de))

Ort: Hörsaal I, Physikalisches Institut

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\\_3268420.html](https://ecampus.uni-bonn.de/goto_ecampus_crs_3268420.html)

Auf ecampus ist ein Forum für Fragen zu Vorlesung, Übungen und Organisation eingerichtet.

# Vorlesungsübersicht

**10.4/12.4** Einstieg in C, numerische Genauigkeit, numerische Ableitung, Integration (einfache Verfahren)

**17.4/19.4** Nullstellensuche, Numerische Integration (Romberg, Gauss)

**24.4/26.4** gewöhnliche Differentialgleichungen (Anfangswertproblem)

**3.5** gewöhnliche Dgl. (Randwertproblem, Eigenwertproblem)

**8.5** gewöhnliche Dgl. (Randwertproblem, Eigenwertproblem)

**10.5** gewöhnliche Dgl. mit Green's Funktion

**17.5** gewöhnliche Dgl. und linear Gleichungssysteme

**29.5/31.5** Systeme linearer Gleichungen: Matrix-Inversion, LU-Zerlegung, Eigenwerte/Eigenvektoren

**5.6/7.6** Integralgleichung, Iterative Lösung, Singularität in Integralgleichung, Interpolation

**12.6/14.6** Partielle Differentialgleichungen (Randwertproblem, Anfangswertproblem)

**19.6/21.6** Partielle Differentialgleichungen (Anfangswertproblem)

**26.6/28.6** Fast Fourier Transformation(FFT), Monte Carlo Methode

**3.7/5.7** Metropolis-Algorithmus

**10.7/12.7** Metropolis-Algorithmus (Anwendung auf Pfadintegrale)

**17.7/19.7** Metropolis-Algorithmus, Parallelisierung mit MPI

Nogga

Luu

Rönchen

Luu

Rönchen

(Änderungen möglich!)

# Übungen und Benotung

- Anmelde und Abmeldefrist bei BASIS: **06.06.2024**
- **Benotung erfolgt auf Basis der Hausaufgaben**
- 11 Übungstermine und ein Drop-In (Mi 8-10):  
Mo 10-12, 13-15, 15-17, Di 8-10, 13-15, 15-17 Mi 13-15, 15-17, Do 8-10, 13-15, 16-18
- Anmeldung über eCampus bereits laufend
- Übungen finden im CIP Pool der Physik ([AVZ 1, Endenicher Allee 11-13, Raum 2.001](#)) statt  
Online Zugriff auf die Linux Rechner wie unter  
<https://www.physik-astro.uni-bonn.de/de/fachgruppe/einrichtungen/cip-pool-physik>  
und auch auf ecampus beschrieben.
- Jede Woche freitags neues Übungsblatt für Anwesenheitsübungen bei den Übungen
- außerdem 6 Hausaufgaben
  - Ausgabe an Freitag **19.4 / 3.5 / 17.5 / 7.6 / 21.6 / 5.7** in eCampus
  - zweiwöchige Bearbeitungszeit (+1 Woche über Pfingsten)
  - Abgabe elektronisch (**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)
- Hausaufgaben-Abgabe per **GitHub Classroom** — wird in den ersten Übungen geübt

# 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, Lösch	Theoretische Physik mit dem Personalcomputer
Vesely	Computational Physics - An Introduction
Kinzel, Reents	Physik per Computer
Hüttenhain, Wallenborn	Skript des C-Programmierkurses von 2015 <a href="http://www.ins.uni-bonn.de/teaching/vorlesungen/ProgKursWs15/ProgKursSkript.pdf">http://www.ins.uni-bonn.de/teaching/vorlesungen/ ProgKursWs15/ProgKursSkript.pdf</a>
B. Kostrzewska, F. Pittler, M. Ueding, C. Urbach	Skript des C-Programmierkurses von 2019 <a href="https://github.com/urbach/c-kurs">https://github.com/urbach/c-kurs</a> aktuelle Version von Justin Schmitz <a href="https://ecampus.uni-bonn.de/goto.php?">https://ecampus.uni-bonn.de/goto.php?</a>

# 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
- oft kleine Geschwindigkeit, oft 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 sind für numerische  
(und viele andere) Anwendungen weit verbreitet.

**C (C++)** ist dabei vermutlich die zur Zeit 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 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
   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 */
```

## “#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**

**Wir werden jeweils 2 Punkte für korrektes Einrücken bzw. hilfreiche Kommentare vergeben!**

```
/* 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**

**printf benötigt einen Formatstring  
mehr Info z.B im C Skript (2015: S.54)**

**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;
```



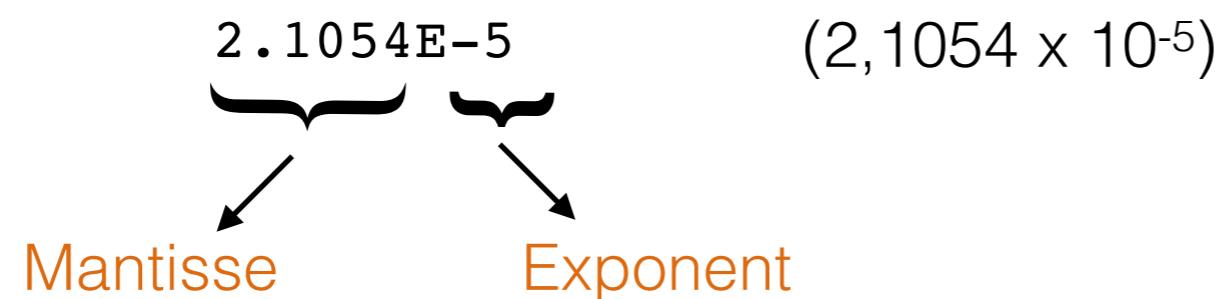
Zuweisung des Wertes 1 an Speicherplatz “n”  
An Speicherposition [0x20+4] wird 5 gespeichert.  
Indizes fangen bei 0 an und gehen bis 10-1 bzw. 20-1

**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 Literature