# Praktisches Material zur Vorlesung Einführung in die Numerik

Peter Bastian
Universität Heidelberg
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Im Neuenheimer Feld 368, D-69120 Heidelberg
email: Peter.Bastian@iwr.uni-heidelberg.de

19. Januar 2010

Dieses Dokument enthält Zusatzmaterial in der Form von Beispielen und Algorithmen sowie deren praktischer Implementierung in C++ zur Vorlesung *Einführung in die Numerik* gehalten im Wintersemester 2009/2010.

## Inhaltsverzeichnis

**1 Motivation** **2**
1.1 Modellbildung und Simulation . . . . . . . . 2
1.2 Ein einfaches Beispiel: Das Fadenpendel . . . . . . . . 4
1.3 Inhaltsübersicht der Vorlesung . . . . . . . . 12

**2 Gleitpunktzahlen** **12**
2.1 Beispiel zur Fehlerakkumulation . . . . . . . . 13

**3 Interpolation und Approximation** **16**
3.1 Motivation . . . . . . . . 16
3.2 Polynominterpolation . . . . . . . . 16
3.3 Spline Interpolation . . . . . . . . 16
3.4 Praktisches zur Diskreten Fourier Analyse . . . . . . . . 25

**4 Quadratur** **26**
4.1 Newton-Cotes Formeln . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.2 Gauß- und adaptive Quadratur . . . . . . . . . . . . . . . . . . . . . . . . 28

**5 Ein kleiner Programmierkurs** **29**
5.1 Hallo Welt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2 Variablen und Typen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.3 Entscheidung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.4 Wiederholung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.5 Funktionen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**6 Heidelberg Educational Numerics Library** **42**
6.1 Einführung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
6.2 Vektoren . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.3 Matrizen . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

# 1 Motivation

## 1.1 Modellbildung und Simulation

### Die Wissenschaftliche Methode

- Experiment: Beobachte was passiert.
- Theorie: Versuche die Beobachtung mit Hilfe von Modellen zu erklären.
- Theorie und Experiment werden sukzessive verfeinert und verglichen, bis eine akzeptable Übereinstimmung vorliegt.
- In Naturwissenschaft und Technik liegen Modelle oft in Form mathematischer Gleichungen vor.
- Oft können die Modellgleichungen nicht geschlossen (mit Papier und Bleistift oder Mathematica . . . ) gelöst werden.

### Simulation

- Simulation: Gleichungen numerisch lösen.
  - Undurchführbare Experimente werden möglich (z. B. Galaxienkollisionen).
  - Teuere Experimente werden eingespart (z. B. Modelle im Windkanal).
  - Parameterstudien schneller durchführbar.
  - (Automatische) Optimierung von Prozessen.

- Vielfältiger Einsatz in Naturwissenschaft, Technik und Industrie: Strömungsberechnung (Klimasimulation), Festigkeit von Bauwerken …
- Grundlage für alle diese Anwendungen sind numerische Algorithmen!

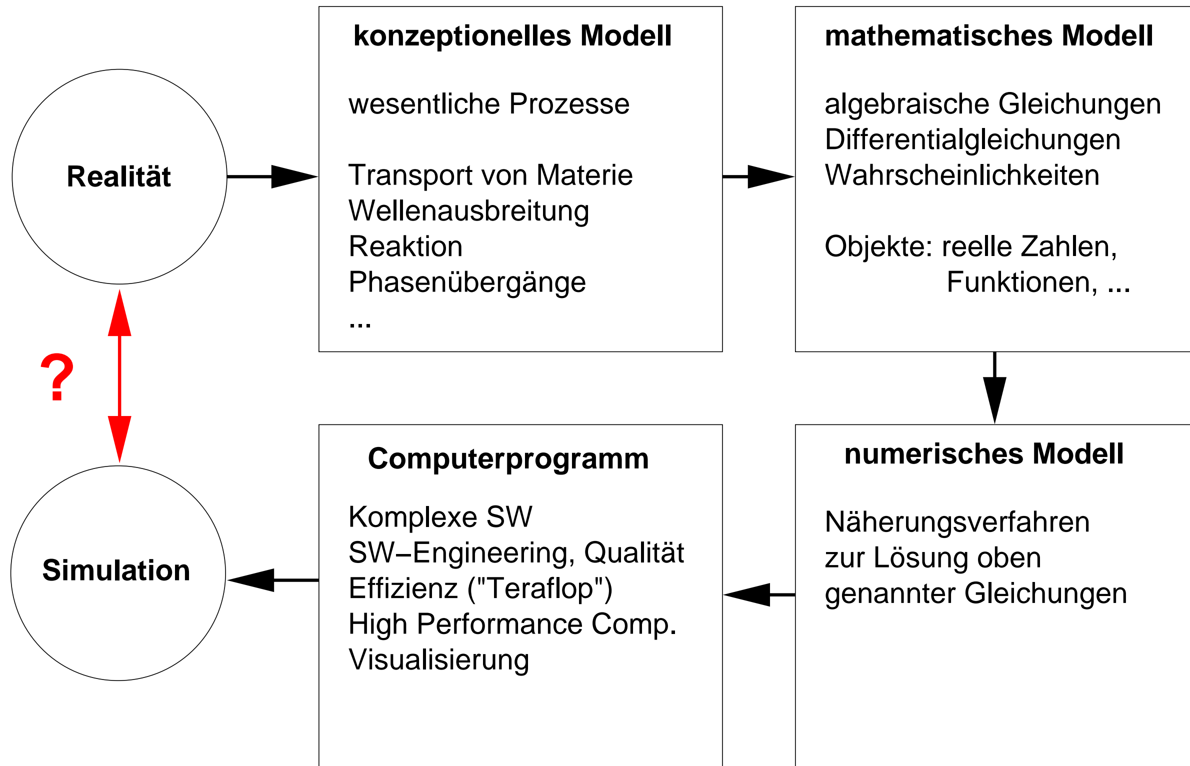



Abbildung 1: Prinzipielles Vorgehen im Wissenschaftlichen Rechnen.

Die prinzipielle Herangehensweise im Wissenschaftlichen Rechnen zeigt Abbildung 1. Die erfolgreiche Durchführung einer Simulation erfordert die interdisziplinäre Zusammenarbeit von Physikern oder Ingenieuren mit Mathematikern und Informatikern.

**Fehlerquellen**

Unterschiede zwischen Experiment und Simulation haben verschiedene Gründe:

- *Modellfehler*: Ein relevanter Prozess wurde nicht oder ungenau modelliert (Temp. konstant, Luftwiderstand vernachlässigt, …)
- *Datenfehler*: Messungen von Anfangsbedingungen, Randbedingungen, Werten für Parameter sind fehlerbehaftet.

- *Abschneidefehler*: Abbruch von Reihen oder Iterationsverfahren, Approximation von Funktionen
- *Rundungsfehler*: Reelle Zahlen werden im Rechner genähert dargestellt.

Untersuchung von Rundungsfehlern und Abschneidefehler ist ein zentraler Aspekt der Vorlesung!

## 1.2 Ein einfaches Beispiel: Das Fadenpendel

### Pisa, 1582

Der Student Galileo Galilei sitzt in der Kirche und ihm ist langweilig. Er beobachtet den langsam über ihm pendelnden Kerzenleuchter und denkt: „Wie kann ich nur die Bewegung dieses Leuchters beschreiben?“.

### Konzeptionelles Modell

Welche Eigenschaften (physikalischen Prozesse) sind für die gestellte Frage relevant?

- Leuchter ist ein Massenpunkt mit der Masse $m$.
- Der Faden der Länge $l$ wird als rigide und masselos angenommen.
- Der Luftwiderstand wird vernachlässigt.

Nun entwickle mathematisches Modell.

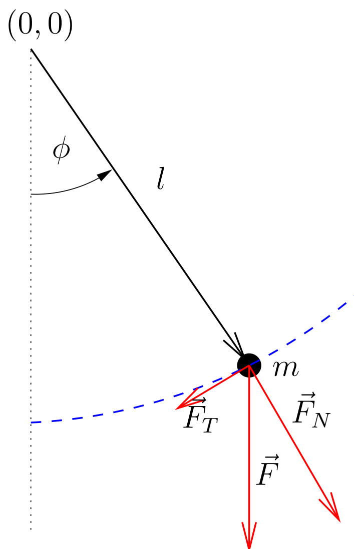



Abbildung 2: Das Fadenpendel.

Abbildung 2 zeigt das Fadenpendel, welches aus dem sogenannten konzeptionellen Modell resultiert.

### Kräfte

- Pendel läuft auf Kreisbahn: Nur *Tangentialkraft* ist relevant.
- Tangentialkraft bei Auslenkung $\phi$:

$$\vec{F}_T(\phi) = -m\,g\sin(\phi)\begin{pmatrix}\cos(\phi)\\ \sin(\phi)\end{pmatrix}.$$

- Also etwa:

$$\begin{aligned}\vec{F}_T(0) &= -mg\begin{pmatrix}0\\ 0\end{pmatrix},\\ \vec{F}_T(\pi/2) &= -mg\begin{pmatrix}0\\ 1\end{pmatrix}.\end{aligned}$$

- Vorzeichen kodiert Richtung.

Dies überlegt man sich so. Die Gewichtskraft zeigt immer nach unten, also

$$\vec{F}(\phi) = mg\begin{pmatrix}0\\ -1\end{pmatrix}.$$

Die Normalkomponente zeigt immer in Richtung $\vec{n}(\phi) = (\sin\phi\,, -\cos\phi)^T$ und damit ist die Kraft in Normalenrichtung

$$\begin{aligned}\vec{F}_N(\phi) = (\vec{F}(\phi)\cdot\vec{n}(\phi))\vec{n} &= \left[mg\begin{pmatrix}0\\ -1\end{pmatrix}\cdot\begin{pmatrix}\sin\phi\\ -\cos\phi\end{pmatrix}\right]\begin{pmatrix}\sin\phi\\ -\cos\phi\end{pmatrix}\\ &= mg\cos\phi\begin{pmatrix}\sin\phi\\ -\cos\phi\end{pmatrix}.\end{aligned}$$

Damit rechnet man die Tangentialkraft aus $\vec{F}_T(\phi) + \vec{F}_N(\phi) = \vec{F}(\phi)$ aus:

$$\begin{aligned}\vec{F}_T(\phi) = \vec{F}(\phi) - \vec{F}_N(\phi) = mg\begin{pmatrix}0\\ -1\end{pmatrix} - mg\cos\phi\begin{pmatrix}\sin\phi\\ -\cos\phi\end{pmatrix} &= -mg\begin{pmatrix}\cos\phi\sin\phi\\ 1-\cos^2\phi\end{pmatrix}\\ = -mg\sin\phi\begin{pmatrix}\cos\phi\\ \sin\phi\end{pmatrix}.\end{aligned}$$

### Weg, Gechwindigkeit, Beschleunigung

- Weg $s(t)$, Geschwindigkeit $v(t)$, Beschleunigung $a(t)$ erfüllen:

$$v(t) = \frac{ds(t)}{dt}, \qquad a(t) = \frac{dv(t)}{dt}.$$

- Für den zurückgelegten Weg (mit Vorzeichen!) gilt $s(t) = l\phi(t)$.
- Also für die Geschwindigkeit

$$v(t) = \frac{d\, s(\phi(t))}{dt} = \frac{d\, l\phi(t)}{dt} = l\frac{d\phi(t)}{dt}$$

- und die Beschleunigung

$$a(t) = \frac{d\, v(\phi(t))}{dt} = l\frac{d^2\phi}{dt^2}(t).$$

### Bewegungsgleichung

- Einsetzen in das 2. Newton'sche Gesetz $m\, a(t) = F(t)$ liefert nun:

$$ml\frac{d^2\phi(t)}{dt^2} = -mg\sin(\phi(t)) \qquad \forall t > t_0.$$

- Die Kraft ist hier skalar (vorzeichenbehafteter Betrag der Tangentialkraft), da wir nur den zurückgelegten Weg betrachten.
- Ergibt *gewöhnliche Differentialgleichung 2. Ordnung* für die Auslenkung $\phi(t)$:

$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{l}\sin(\phi(t)) \qquad \forall t > t_0. \tag{1}$$

- Eindeutige Lösung erfordert *zwei* Anfangsbedingungen ($t_0 = 0$):

$$\phi(0) = \phi_0, \qquad \frac{d\phi}{dt}(0) = u_0. \tag{2}$$

### Lösung bei kleiner Auslenkung

- Allgemeine Gleichung für das Pendel ist schwer „analytisch“ zu lösen.
- Für *kleine* Winkel $\phi$ gilt

$$\sin(\phi) \approx \phi,$$

  z.B. $\sin(0.1) = 0,099833417$.

- Diese *Näherung* reduziert die Gleichung auf

$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{l}\phi(t).$$

- Ansatz $\phi(t) = A\cos(\omega t)$ liefert mit $\phi(0) = \phi_0$, $\frac{d\phi}{dt}(0) = 0$ dann die aus der Schule bekannte Formel

$$\phi(t) = \phi_0 \cos\left(\sqrt{\frac{g}{l}}t\right) \tag{3}$$

**Volles Modell; Verfahren 1**

- Löse das volle Modell mit zwei numerischen Verfahren.
- Ersetze Gleichung zweiter Ordnung durch *zwei* Gleichungen *erster* Ordnung:

$$\frac{d\phi(t)}{dt} = u(t), \qquad \frac{d^2\phi(t)}{dt^2} = \frac{du(t)}{dt} = -\frac{g}{l}\sin(\phi(t)).$$

- Ersetze Ableitungen durch Differenzenquotienten:

$$\begin{aligned}\frac{\phi(t+\Delta t) - \phi(t)}{\Delta t} &\approx \frac{d\phi(t)}{dt} = u(t),\\ \frac{u(t+\Delta t) - u(t)}{\Delta t} &\approx \frac{du(t)}{dt} = -\frac{g}{l}\sin(\phi(t)).\end{aligned}$$

- Mit $\phi^n = \phi(n\Delta t)$, $u^n = u(n\Delta t)$ erhält man Rekursion (*Euler*):

$$\phi^{n+1} = \phi^n + \Delta t\, u^n \qquad \phi^0 = \phi_0 \tag{4}$$

$$u^{n+1} = u^n - \Delta t\,(g/l)\,\sin(\phi^n) \qquad u^0 = u_0 \tag{5}$$

**Volles Modell; Verfahren 2**

- Nutze Näherungsformel für die zweite Ableitung, sog. *Zentraler Differenzenquotient*:

$$\frac{\phi(t+\Delta t) - 2\phi(t) + \phi(t-\Delta t)}{\Delta t^2} \approx \frac{d^2\phi(t)}{dt^2} = -\frac{g}{l}\sin(\phi(t)).$$

- Auflösen nach $\phi(t+\Delta t)$ ergibt Rekursionsformel ($n \geq 2$):

$$\phi^{n+1} = 2\phi^n - \phi^{n-1} - \Delta t^2\,(g/l)\,\sin(\phi^n) \tag{6}$$

  mit der Anfangsbedingung

$$\phi^0 = \phi_0, \qquad \phi^1 = \phi_0 + \Delta t\, u_0. \tag{7}$$

  (Die zweite Bedingung kommt aus dem Eulerverfahren oben).

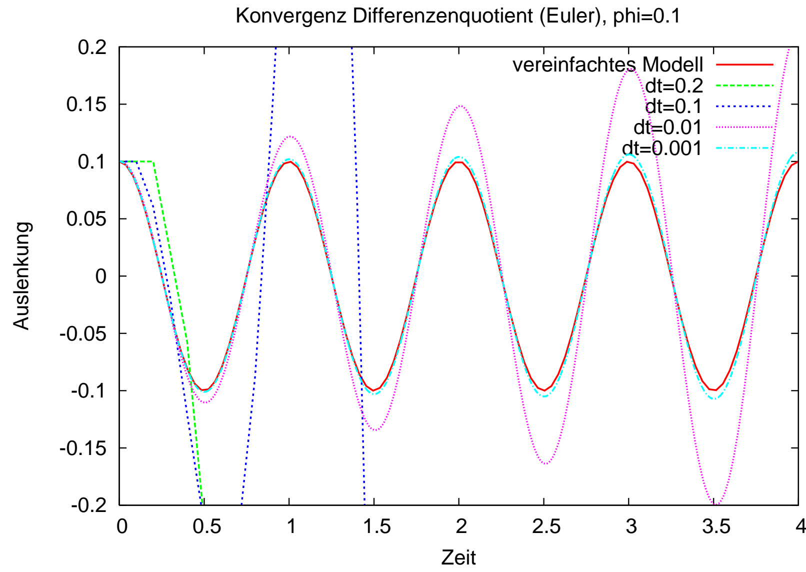



Abbildung 3: Simulation des Fadenpendels (volles Modell) bei $\phi_0 = 0.1 \approx 5.7°$ mit dem Eulerverfahren.

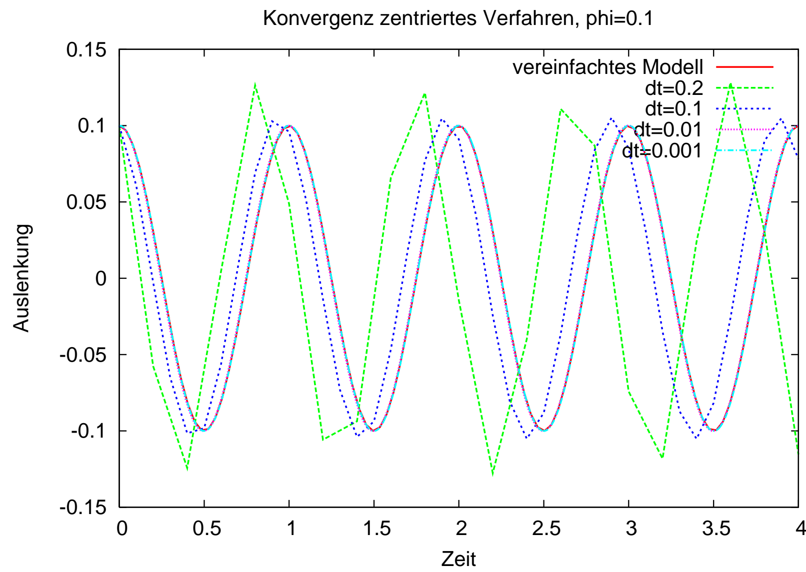



Abbildung 4: Simulation des Fadenpendels (volles Modell) bei $\phi_0 = 0.1 \approx 5.7°$ mit dem zentralen Verfahren.

Abbildung 3 zeigt das Eulerverfahren in Aktion. Für festen Zeitpunkt $t$ und $\Delta t \to 0$ *konvergiert* das Verfahren. Für festes $\Delta t$ und $t \to \infty$ nimmt das Verfahren immer größere Werte an.

Abbildung 4 zeigt zum Vergleich das zentrale Verfahren für die gleiche Anfangsbedingung. Im Unterschied zum expliziten Euler scheint der Fehler bei festem $\Delta t$ und $t \to \infty$ nicht unbeschränkt zu wachsen.

Nun können wir das volle Modell mit dem vereinfachten Modell vergleichen und sehen welche Auswirkungen die Annahme $\sin\phi \approx \phi$ auf das Ergebnis hat. Abbildung 5 zeigt die numerische Simulation. Selbst bei 28.6° ist die Übereinstimmung noch einigermaßen passabel. Für größere Auslenkungen ist das vereinfachte Modell völlig unbrauchbar, die Form der Schwingung ist kein Kosinus mehr.

### Ausblick auf weiterführende Vorlesungen

- Viele Anwendungen führen auf *partielle Differentialgleichungen*. Gesucht sind dann Funktionen in mehreren Raumdimensionen.
- Technische Anwendungen erfordern *Optimierung*.
- Effiziente Programmierung und Visualisierung der Ergebnisse sind sehr wichtig.
- Wir zeigen exemplarisch zwei Anwendungen.

### Eine Geothermieanlage

Wir betrachten die Modellierung und Simulation einer Geothermieanlage. Den schematischen Aufbau zeigt die Abbildung 6. Kaltes Wasser fließt in einer Bohrung nach unten, wird erwärmt und in einem isolierten Innenrohr wieder nach oben geführt.

- Grundwasserströmung gekoppelt mit Wärmetransport.
- Welche Leistung erzielt so eine Anlage?

### Modell für eine Geothermieanlage

- *Strömung des Wassers* in und um das Bohrloch

$$\begin{aligned} \nabla \cdot u &= f, \\ u &= -\frac{K}{\mu}(\nabla p - \varrho_w G) \end{aligned}$$

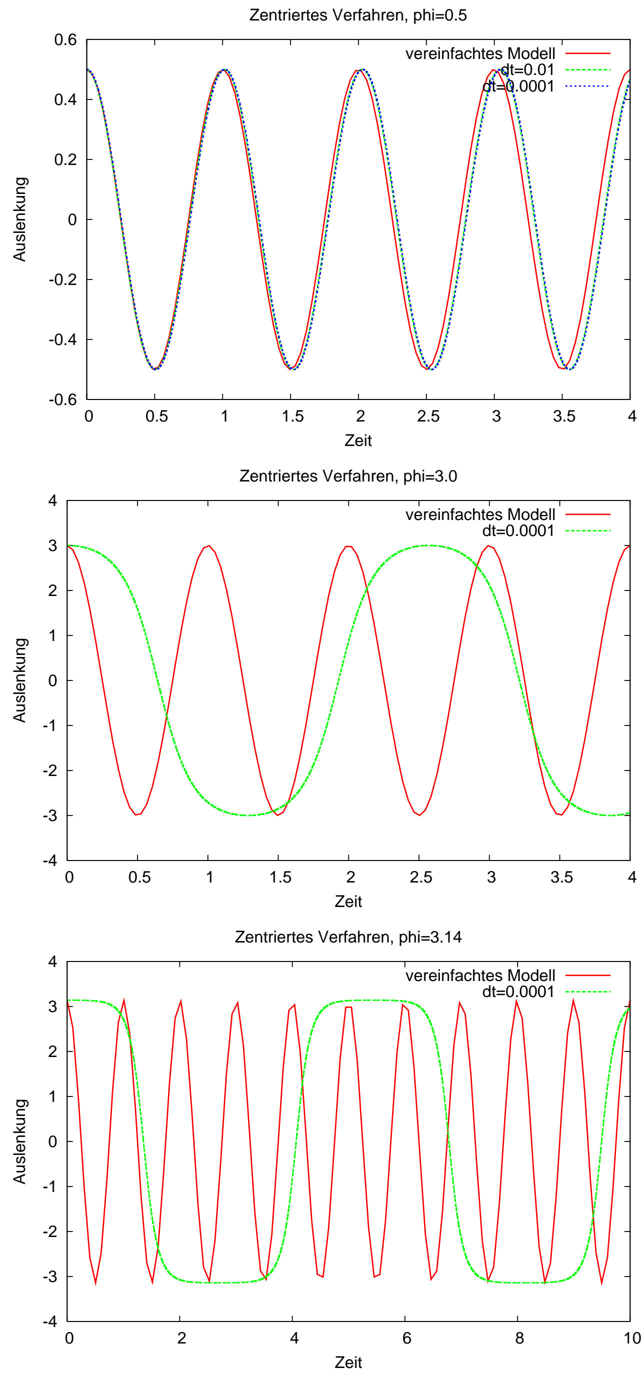



Abbildung 5: Vergleich von vollem und vereinfachtem Modell (jeweils in rot) bei den Winkeln $\phi = 0.5, 3.0, 3.14$ gerechnet mit dem zentralen Verfahren.

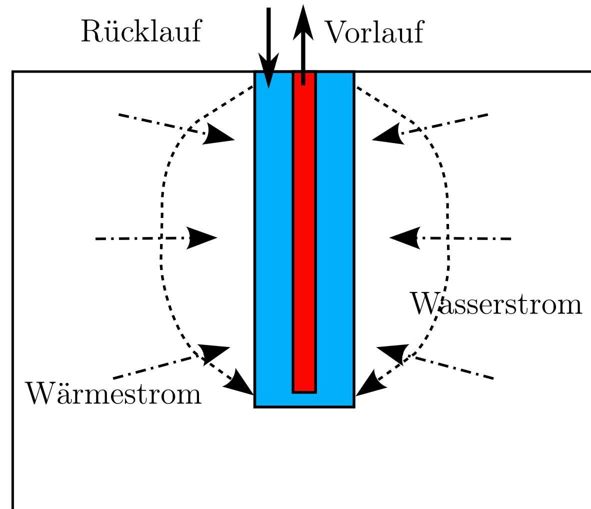



Abbildung 6: Schematischer Aufbau einer Geothermieanlage.

- Transport der Wärme durch *Konvektion* und *Wärmeleitung*

$$\frac{\partial(c_e \varrho_e T)}{\partial t} + \nabla \cdot q + g^- T = g^+,$$
$$q = c_w \varrho_w u T - \lambda \nabla T$$

  in Abhängigkeit diverser *Parameter*: Bodendurchlässigkeit, Wärmekapazität, Dichte, Wärmeleitfähigkeit, Pumprate sowie Rand- und Anfangsbedingungen.

Abbildung 6 zeigt die Entzugsleistung so einer Anlage für die ersten hundert Tage Betriebszeit. Dabei wurden plausible Werte für die Modellparameter gewählt.

### Schadstoffausbreitung

Als weiteres Beispiel nennen wir die Modellierung und Simulation einer Schadstoffausbreitung im Boden, siehe Abbildung 8. Der Schadstoff wird hier als nicht mit Wasser mischbar und schwerer als Wasser angenommen (Bestimmte Reinigungsmittel haben diese Eigenschaften).

- Wo erreicht der Schadstoff welche Konzentrationen?
- Wie bekommt man den Schadstoff wieder weg?
- Wohin bewegt sich gelöster Schadstoff?

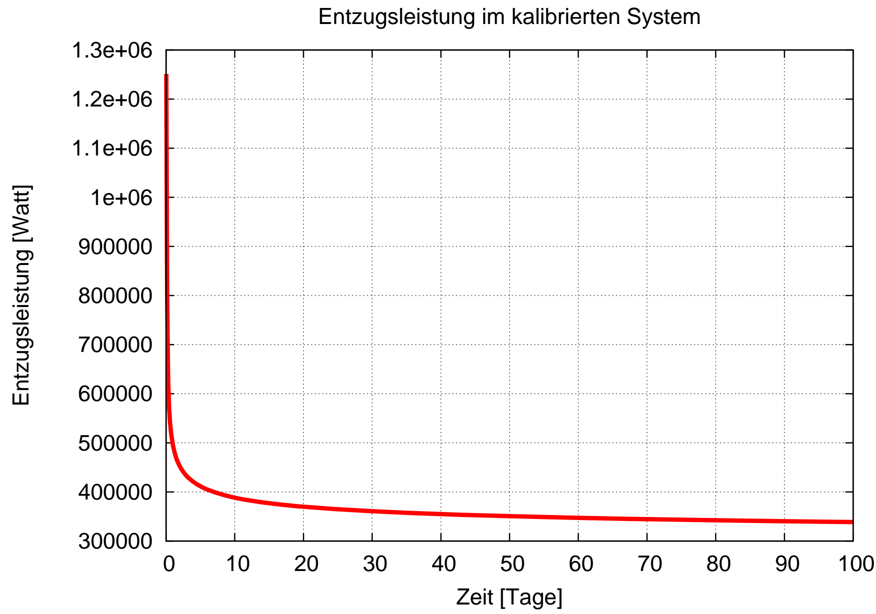



Abbildung 7: Entzugsleistung der Geothermieanlage über die Zeit für bestimmte Systemparameter.

## 1.3 Inhaltsübersicht der Vorlesung

Wir werden in dieser Vorlesung die folgenden Themengebiete behandeln:

- Grundbegriffe, Gleitpunktzahlen, Gleitpunktarithmetik
- Direkte Methoden zur Lösung linearer Gleichungssysteme
- Interpolation und Approximation
- Numerische Integration
- Iterationsverfahren zur Lösung linearer Gleichungssysteme
- Iterationsverfahren zur Lösung nichtlinearer Gleichungssysteme
- Eigenwerte und Eigenvektoren

# 2 Gleitpunktzahlen

### Zahlen im Computer

Alle Programmiersprachen stellen elementare Datentypen zur Repräsentation von Zahlen zur Verfügung. In C/C++:

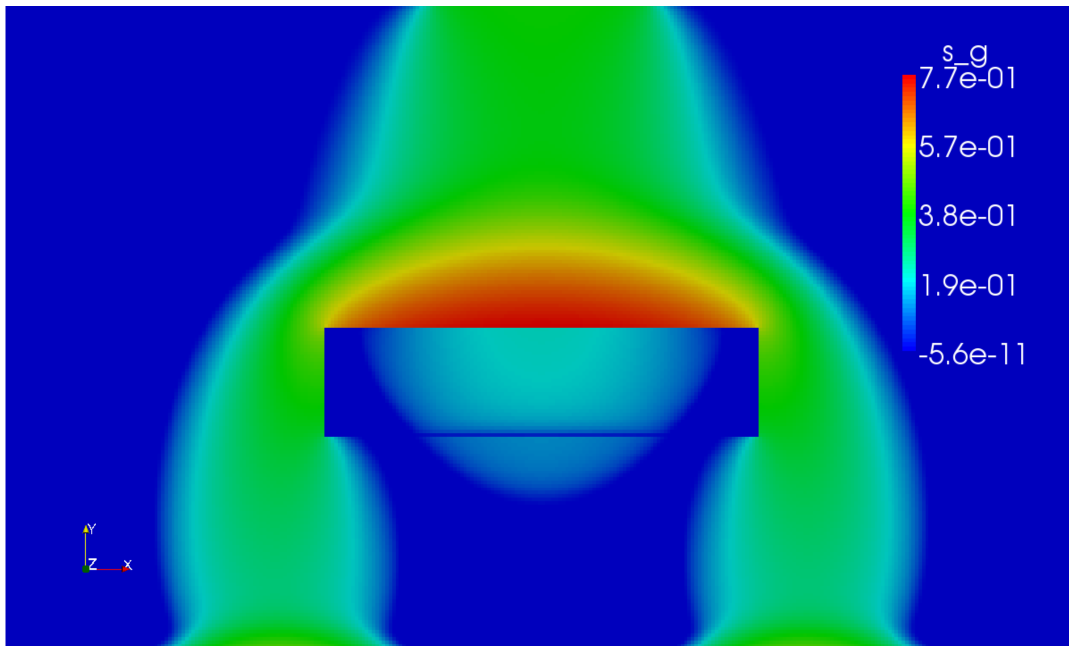



Abbildung 8: Ausbreitung eines nicht Wasser nicht mischbaren Schadstoffes im Boden.

| | |
|---|---|
| unsigned int | $\mathbb{N}_0$ |
| int | $\mathbb{Z}$ |
| float | $\mathbb{R}$ |
| double | $\mathbb{R}$ |
| complex<double> | $\mathbb{C}$ |

Diese sind Idealisierungen der Zahlenmengen $\mathbb{N}_0, \mathbb{Z}, \mathbb{R}, \mathbb{C}$.

Bei *unsigned int* und *int* besteht die Idealisierung darin, dass es eine größte (bzw. kleinste) darstellbare Zahl gibt. Ansonsten sind die Ergebnisse *exakt.*

Bei *float* und *double* kommt hinzu, dass die meisten innerhalb des erlaubten Bereichs liegenden Zahlen nur *näherungsweise* dargestellt werden können.

## 2.1 Beispiel zur Fehlerakkumulation

### Potenzreihe für $e^x$

- $e^x$ lässt sich mit einer Potenzreihe berechnen:

$$e^x = 1 + \sum_{n=1}^{\infty} \frac{x^n}{n!} = 1 + \sum_{n=1}^{\infty} y_n.$$

- Dies formulieren wir rekursiv:

$$y_1 = x, \qquad S_1 = 1 + y_1, \qquad \text{(Anfangswerte)}.$$

Rekursion:

$$y_n = \frac{x}{n} y_{n-1}, \qquad S_n = S_{n-1} + y_n.$$

- Probiere verschiedene Genauigkeiten und $x$.

## Positives Argument

- Für $x = 1$ und `float`-Genauigkeit erhalten wir:

```
1.000000000000000e+00   2  2.500000000000000e+00
5.000000000000000e-01   3  2.666666746139526e+00
1.666666716337204e-01   4  2.708333492279053e+00
4.166666790843010e-02   5  2.716666936874390e+00
8.333333767950535e-03   6  2.718055725097656e+00
1.388888922519982e-03   7  2.718254089355469e+00
1.984127011382952e-04   8  2.718278884887695e+00
2.480158764228690e-05   9  2.718281745910645e+00
2.755731884462875e-06  10  2.718281984329224e+00
2.755731998149713e-07  100 2.718281984329224e+00
0.000000000000000e+00  ex  2.718281828459045E0
```

...also 7 gültige Ziffern.

- Für $x = 5$ ...

```
9.333108209830243e-06  ex  1.484131591025766E2
```

...dito.

## Negatives Argument

- Für $x = -1$ und `float`-Genauigkeit erhalten wir:

```
2.755731998149713e-07   11 3.678793907165527e-01
-2.505210972003624e-08  12 3.678793907165527e-01
2.087675810003020e-09   ex 3.678794411714423E-1
```

...6 gültige Ziffern.

- Für $x = -5$

```
-5.000000000000000e+00   2  8.500000000000000e+00
 1.250000000000000e+01   3 -1.233333396911621e+01
-2.083333396911621e+01   4  1.370833396911621e+01
 2.604166793823242e+01  15  1.118892803788185e-03
-2.333729527890682e-02  16  8.411797694861889e-03
 7.292904891073704e-03  28  6.737461313605309e-03
 1.221854423194557e-10 100  6.737461313605309e-03
 0.000000000000000e+00  ex  6.737946999085467E-3
```

nur noch 4 gültige Ziffern!

## Noch kleineres Argument

- Für $x = -20$ und `float`-Genauigkeit sind ...

```
-2.000000000000000e+01   2  1.810000000000000e+02
 2.000000000000000e+02   3 -1.152333374023438e+03
-1.333333374023438e+03   4  5.514333496093750e+03
 6.666666992187500e+03   5 -2.115233398437500e+04
-2.666666796875000e+04  31 -1.011914250000000e+06
-2.611609750000000e+06  32  6.203418750000000e+05
 1.632256125000000e+06  33 -3.689042500000000e+05
-9.892461250000000e+05  34  2.130052500000000e+05
 5.819095000000000e+05  35 -1.195144687500000e+05
-3.325197187500000e+05  36  6.521870312500000e+04
 1.847331718750000e+05  65  7.566840052604675e-01
-4.473213550681976e-07  66  7.566841244697571e-01
 1.355519287926654e-07  67  7.566840648651123e-01
-4.046326296247571e-08  68  7.566840648651123e-01
 1.190095932912527e-08  ex  2.061153622438557E-9
```

keine Ziffern mehr gültig. Das Ergebnis ist um 8 Größenordnungen daneben!

### Erhöhen der Genauigkeit

- Für $x = -20$ und `double`-Genauigkeit erhält man

```
-1.232613988175268e+07   28 3.623690792934047e+06
 8.804385629823344e+06   94 6.147561828914626e-09
 1.821561256740375e-24   95 6.147561828914626e-09
-3.834865803663947e-25   ex 2.061153622438557E-9
```

  Immer noch um einen Faktor 3 daneben!

- Erst mit „vierfacher Genauigkeit“ erhält man

```
-4.1852929339382073650363741579941E-41 118 2.0611536224385583392700458752947E-9
 7.0937168371834023136209731491427E-42  ex 2.0611536224385578279659403801558E-9
```

  15 gültige Ziffern (bei ca 30 Ziffern „Rechengenauigkeit“).

### Fragen

- Was bedeutet überhaupt „Rechengenauigkeit“?
- Welche Genauigkeit können wir erwarten?
- Wo kommen diese Fehler her?
- Wie werden solche „Kommazahlen“ dargestellt und verarbeitet?

Obige Berechnungen wurden mit den Paketen `qd` und `arprec` (beide `http://crd.lbl.gov/~dhbailey/m` durchgeführt. `qd` erlaubt bis zu vierfache `double` Genauigkeit, `arprec` beliebige Genauigkeit. Die GNU multiprecision library (`http://gmplib.org/`) ist eine Alternative. □

# 3 Interpolation und Approximation

## 3.1 Motivation

Die Abbildungen 9 bis 11 zeigen eine Anwendung von Polynomen bei der Kurvenkompression in der Computergraphik.

Die Lage eines starren Körpers im Raum wird durch 6 Zahlen festgelegt (3 für die Position und 3 für die Orientierung), die sich mit der Zeit ändern können. Eine äquidistante Schrittweite erfordert einen hohen Speicheraufwand, um bei schnellen Positionsänderungen eine gute Genauigkeit erreichen zu können. Bei einer adaptiven Schrittweitenwahl werden möglichst wenig Zeitpunkte ausgewählt, aber so, dass ein vorgegebener Fehler nicht überschritten wird. Diese Anwendung haben Eric Schneider, Manuel Jerger und Benjamin Jillich im Rahmen eines Software-Praktikums im Sommersemester 2008 erarbeitet (Vielen Dank für die tollen Bilder!).

## 3.2 Polynominterpolation

Die Abbildung 12 zeigt die Monome bis zum Grad 6.

Abbildung 13 zeigt die Lagrange-Polynome vom Grad 6 bei äquidistanten Stützstellen auf $[0, 1]$.

Zu interpolieren sei die folgende Wertetabelle mit 4 Einträgen:

| $x_i$ | $y_i$ |
|---|---|
| 0 | 1.0000 |
| 2 | 0.4546 |
| 7 | 0.0938 |
| 10 | −0.0544 |

Abbildung 14 zeigt das zugehörige Interpolationspolynom sowie die skalierten Lagrange-Polynome $y_i L_i^{(3)}$. □

Abbildung 16 illustriert das Wachsen der Lagrange-Polynome weit weg von der Stützstelle $x$ an der $L_i^{(n)(x)} = 1$ gilt.

## 3.3 Spline Interpolation

Wir betrachten die Interpolation der folgenden drei Funktionen

$$f_1(x) = \exp(-x^2) \qquad \text{in } [-10, 10], \tag{9}$$

$$f_2(x) = \begin{cases} \cos^2(x) & |x| < \pi/2 \\ 0 & |x| \geq \pi/2 \end{cases} \qquad \text{in } [-\pi, \pi], \tag{10}$$

$$f_3(x) = \begin{cases} -1 & x < 1/2 \\ +1 & x \geq 1/2 \end{cases} \qquad \text{in } [0, 1], \tag{11}$$

mittels Polynomen, $S_h^{1,0}$ und $S_h^{3,2}$ (mit natürlichen Randbedingungen).

Abbildung 9: Kurvenkompression in der Computergraphik: Die Szene.

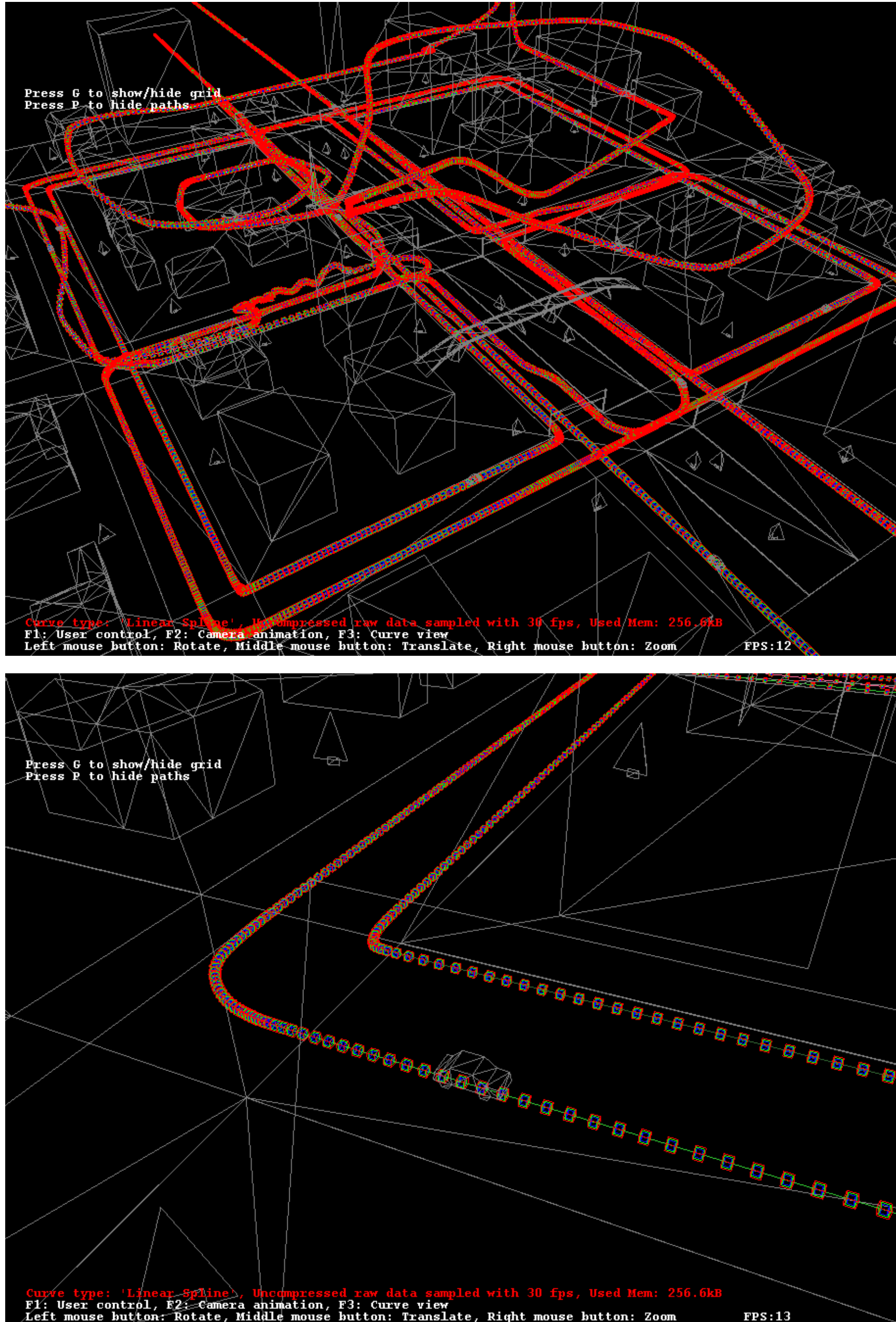



Abbildung 10: Kurvenkompression in der Computergraphik: Stützpunkte der unkomprimierten Kurve.

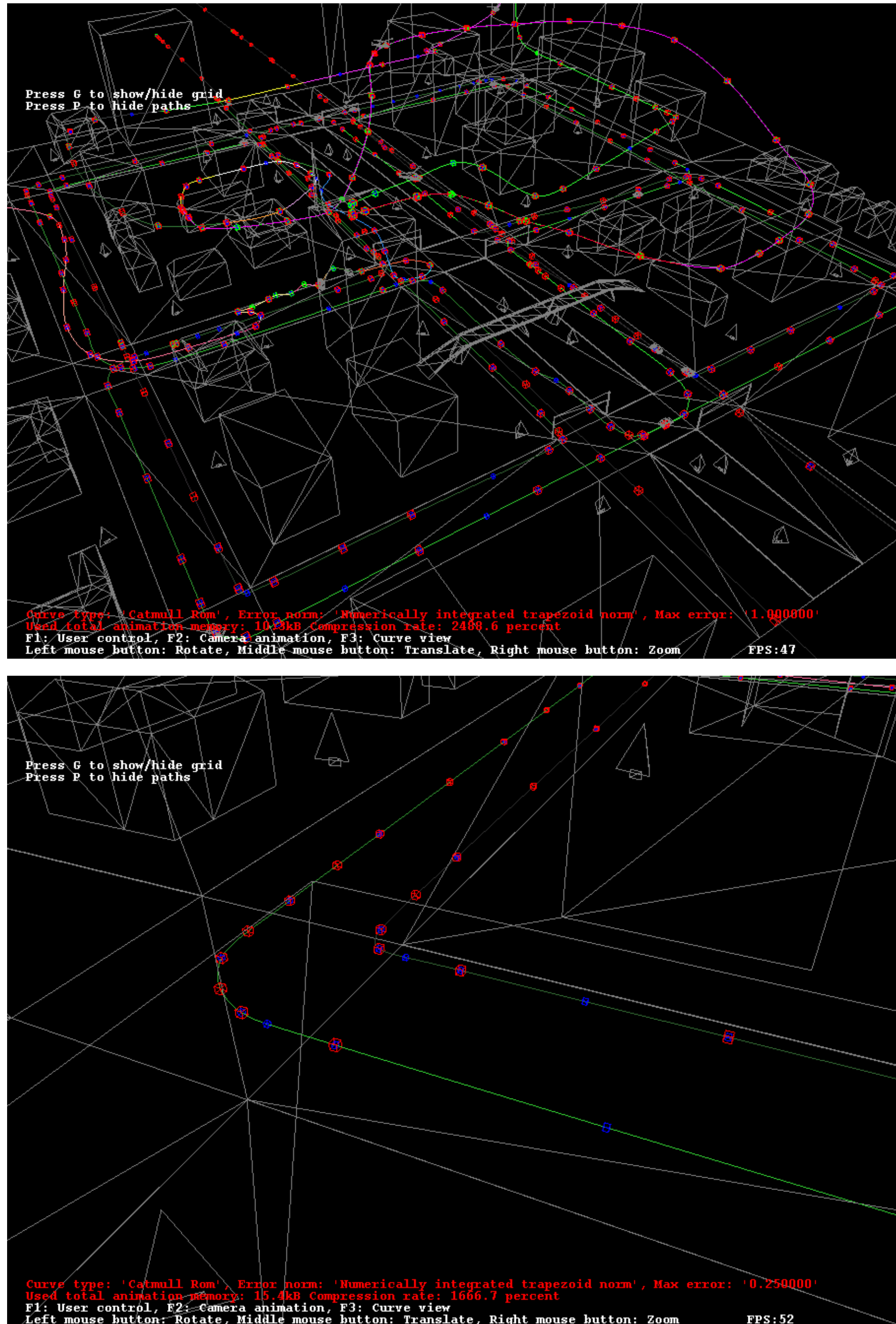



Abbildung 11: Kurvenkompression in der Computergraphik: Stützpunkte der komprimierten Kurve.

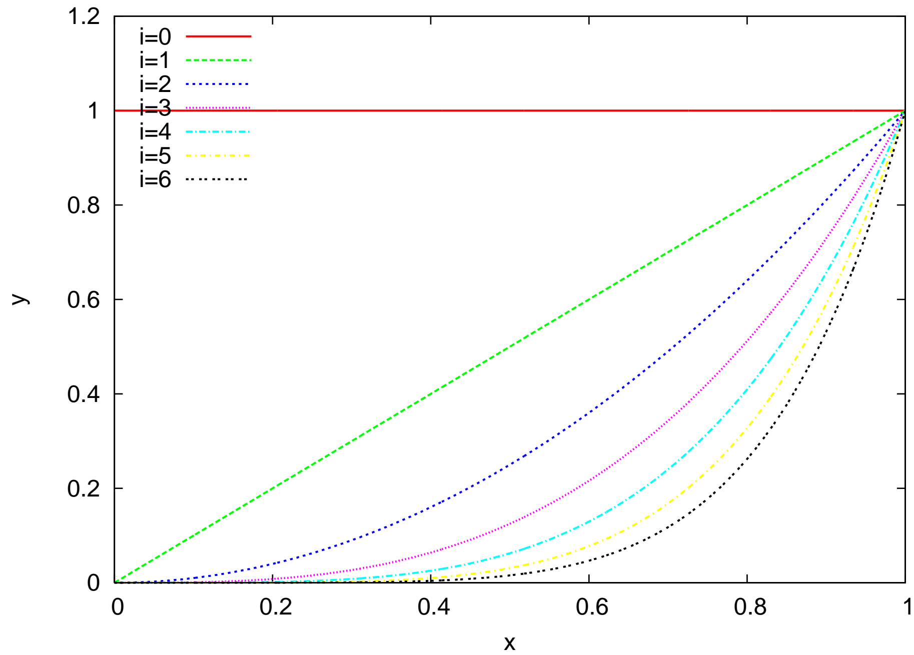
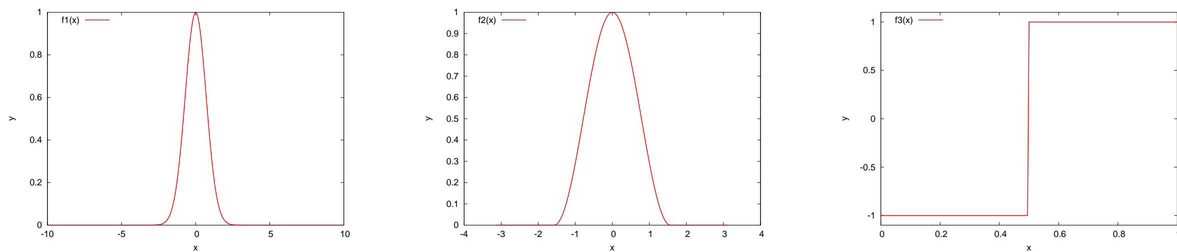



Abbildung 12: Die Monome bis zum Grad 6.




Die Abbildung 17 zeigt die Interpolation der Funktion $f_1(x)$.
Die Abbildung 18 zeigt die Interpolation der Funktion $f_2(x)$.
Die Abbildung 19 zeigt die Interpolation der Funktion $f_3(x)$.
Wir lernen:

- Interpolation mit Polynomen steigenden Grades an äquidistanten Stützstellen schlägt in allen Fällen fehl, d. h. der Interpolationsfehler steigt mit dem Grad an.
- Kubische Splines konvergieren und liefern einen glatten Verlauf. Allerdings kommt es zu möglicherweise „unphysikalischen“ Unter- bzw. Überschwingern. Diese sind aber im Falle von $f_3(x)$ um die Sprungstelle lokalisiert.
- Stückweise lineare Funktionen haben diesen Defekt nicht.

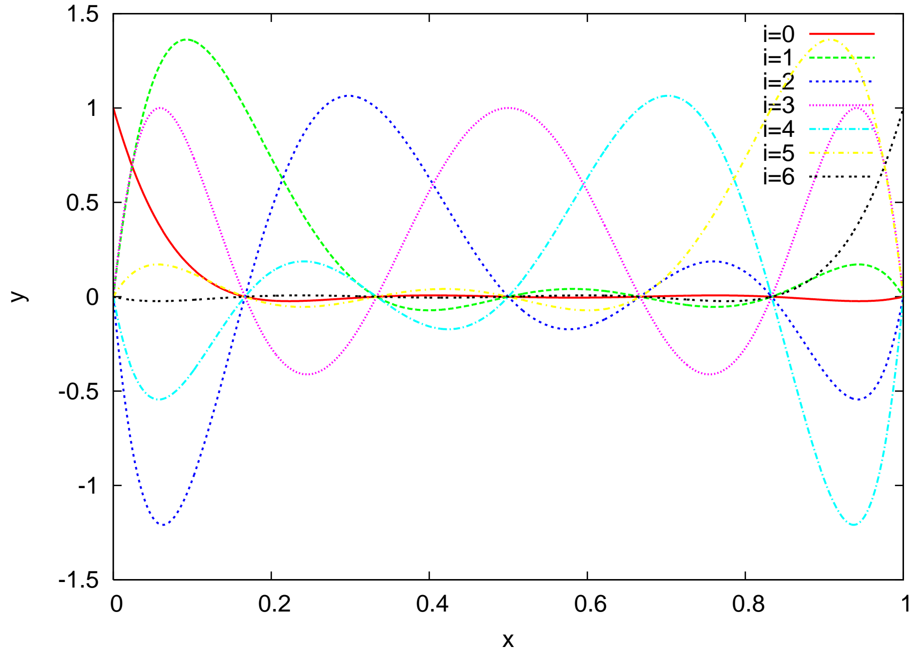



Abbildung 13: Die Lagrange-Polynome $L_i^{(6)}(x)$ vom Grad 6.

Wir wollen nun den Interpolationsfehler noch experimentell bestimmen.
Fehler bei Interpolation der Funktion $f_1(x) = e^{-x^2}$:

| $n$ | $S_h^{1,0}$ | $S_h^{3,2}$ | $P_n$ |
|---:|---:|---:|---:|
| 4 | $6.045_e - 01$ | $7.420_e - 01$ | $8.038_e - 01$ |
| 6 | $4.447_e - 01$ | $5.612_e - 01$ | $9.999_e - 01$ |
| 8 | $3.002_e - 01$ | $3.918_e - 01$ | $2.311_e + 00$ |
| 10 | $1.774_e - 01$ | $2.464_e - 01$ | $5.949_e + 00$ |
| 16 | $1.060_e - 01$ | $2.753_e - 02$ | |
| 32 | $6.946_e - 02$ | $7.083_e - 03$ | |
| 64 | $2.241_e - 02$ | $3.316_e - 04$ | |
| 128 | $5.974_e - 03$ | $1.918_e - 05$ | |
| 256 | $1.517_e - 03$ | $1.173_e - 06$ | |
| 512 | $3.809_e - 04$ | $7.289_e - 08$ | |
| 1024 | $9.533_e - 05$ | $4.549_e - 09$ | |

Angegeben ist der maximale Fehler an einem Punkt. Polynome konvergieren nicht.
Stückweise linear konvergiert mit $h^2$ (d. h. $e_{2n}/e_n = (1/2)^2$), kubische Splines mit $h^4$ (d. h. $e_{2n}/e_n = (1/2)^4$).
In beiden Fällen gilt dies nur, wenn $n$ genügend groß, man spricht von „asymptotischer“

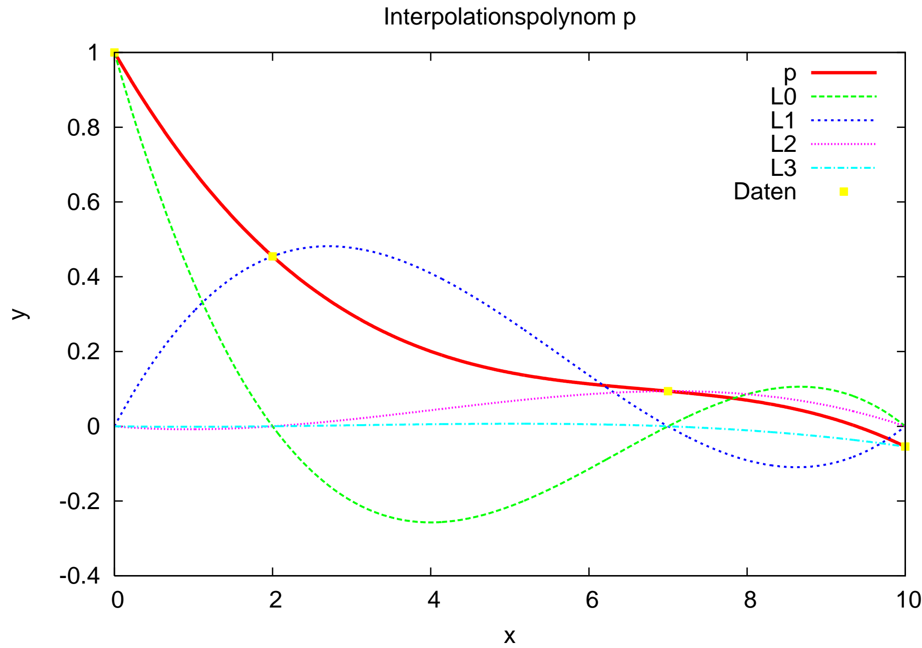



Abbildung 14: Interpolationspolynom zur Wertetabelle aus Beispiel .

Konvergenz.

Fehler bei Interpolation der Funktion $f_2(x) = \begin{cases} \cos^2(x) & x < \pi/2 \\ 0 & x \geq \pi/2 \end{cases}$ :

| $n$ | $S_h^{1,0}$ | $S_h^{3,2}$ |
|---:|---:|---:|
| 4 | $1.052e-01$ | $1.649e-01$ |
| 8 | $1.052e-01$ | $4.498e-02$ |
| 16 | $3.518e-02$ | $8.434e-03$ |
| 32 | $9.423e-03$ | $1.945e-03$ |
| 64 | $2.396e-03$ | $4.764e-04$ |
| 128 | $6.015e-04$ | $1.184e-04$ |
| 256 | $1.505e-04$ | $2.958e-05$ |
| 512 | $3.764e-05$ | $7.394e-06$ |
| 1024 | $9.412e-06$ | $1.848e-06$ |

In diesem Fall konvergiert der maximale Fehler auch im Falle kubischer Splines nur mit $h^2$.

Dies liegt daran, dass $f_2''(x)$ unstetig am Punkt $x = \pi/2$ ist (springt von 2 auf 0).

Die dritte Ableitung existiert nicht mehr. □

Für die Interpolation mit stückweisen Polynomen merken wir uns:

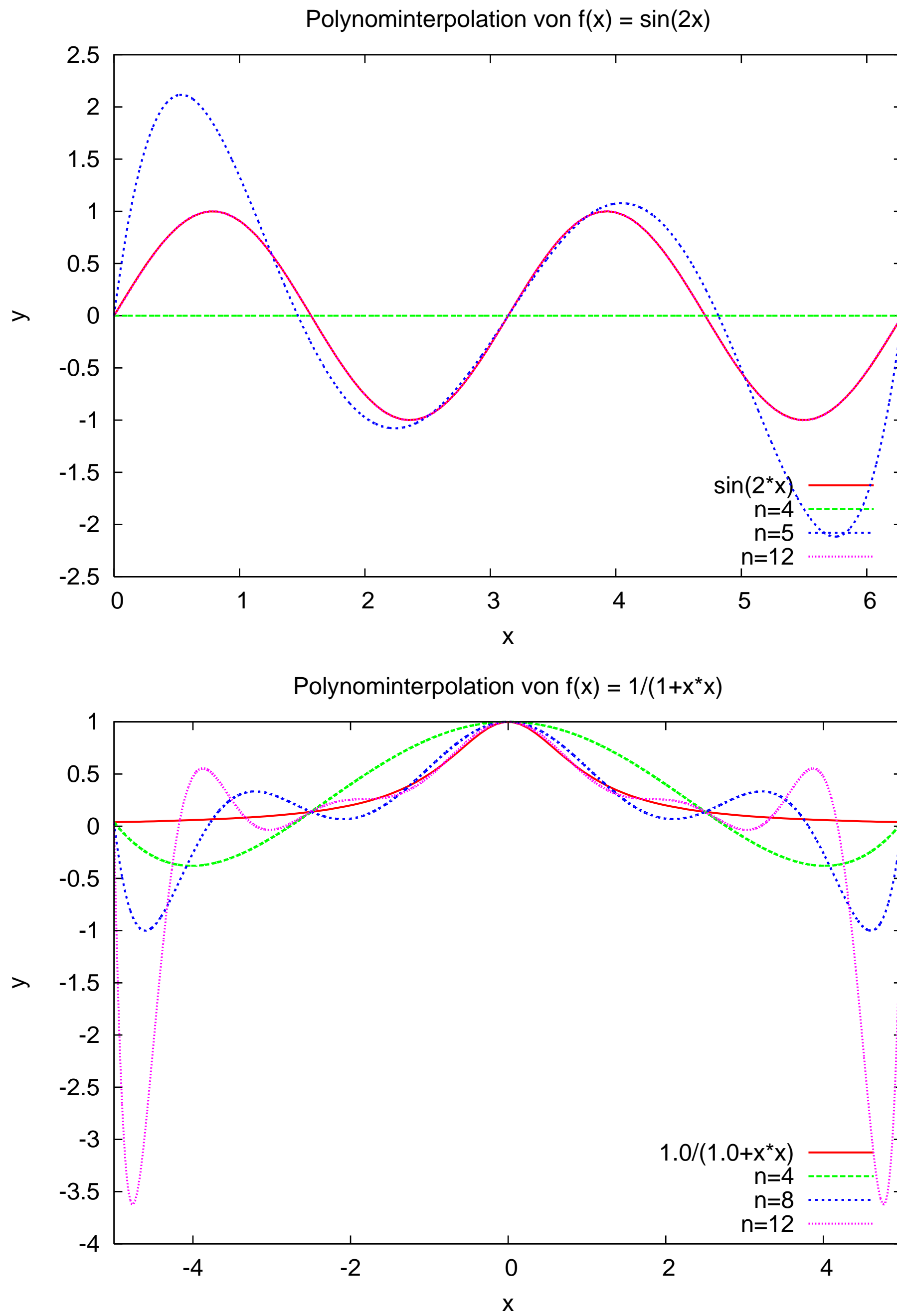



Abbildung 15: Interpolation der Funktionen $\sin(2x)$ (oben) und $\frac{1}{1+x^2}$ (unten) mit äquidistanten Stützstellen und verschiedenen Polynomgraden.

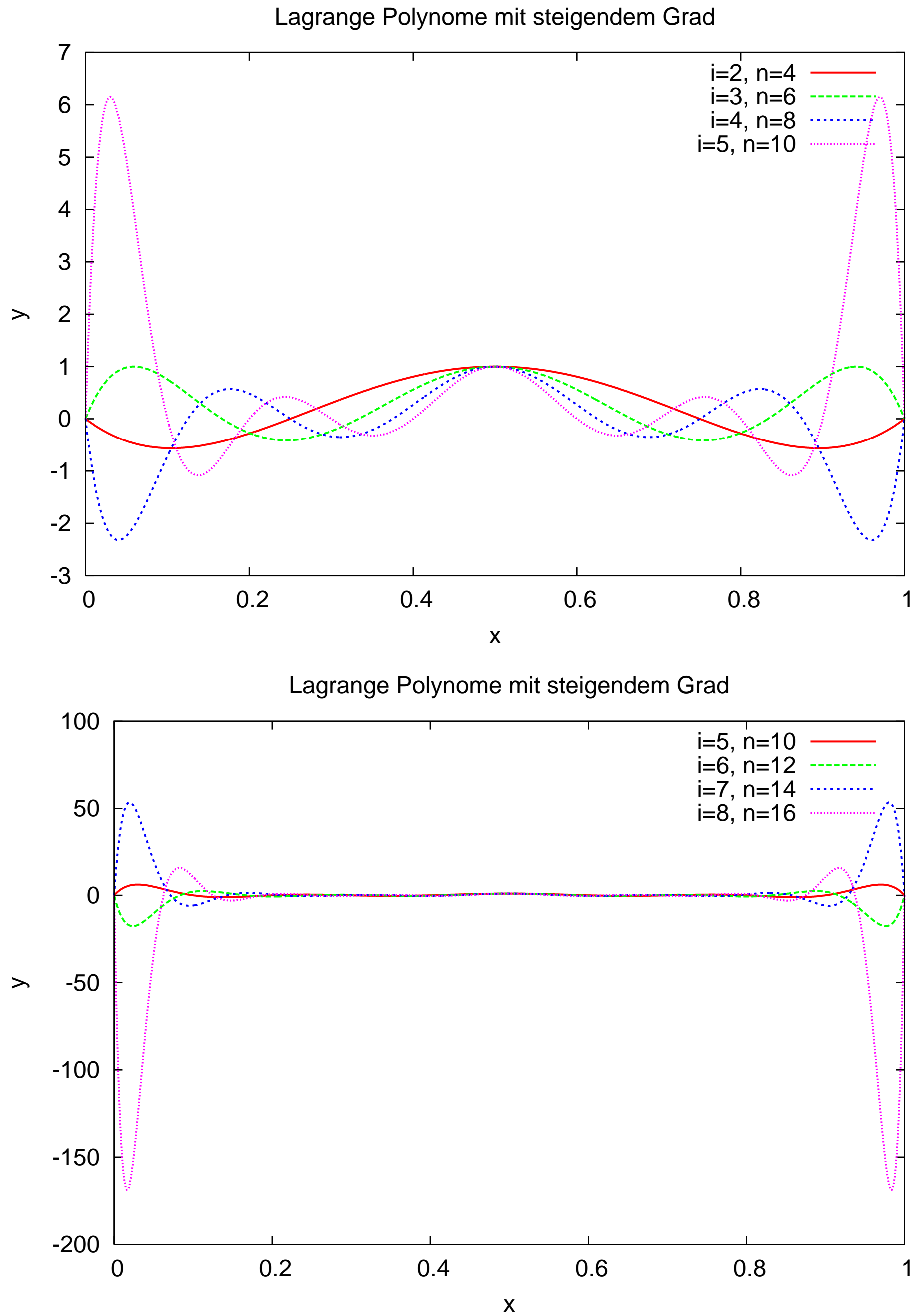



Abbildung 16: Die Lagrange-Polynome $L_{n/2}^{(n)}$ für $n = 4, 6, 8, 10, 12, 14, 16$.

Je höher der (abschnittsweise) Polynomgrad umso schneller konvergiert das Verfahren. Im allgemeinen erhält man $O(h^{k+1})$ Konvergenz für Polynome vom Grad $k$.

Dies gilt allerdings nur dann, wenn die zu interpolierende Funktion genügend of differenzierbar ist. Ist dies nicht der Fall so lohnt also auch die Verwendung von Polynomen hohen Grades nicht. □

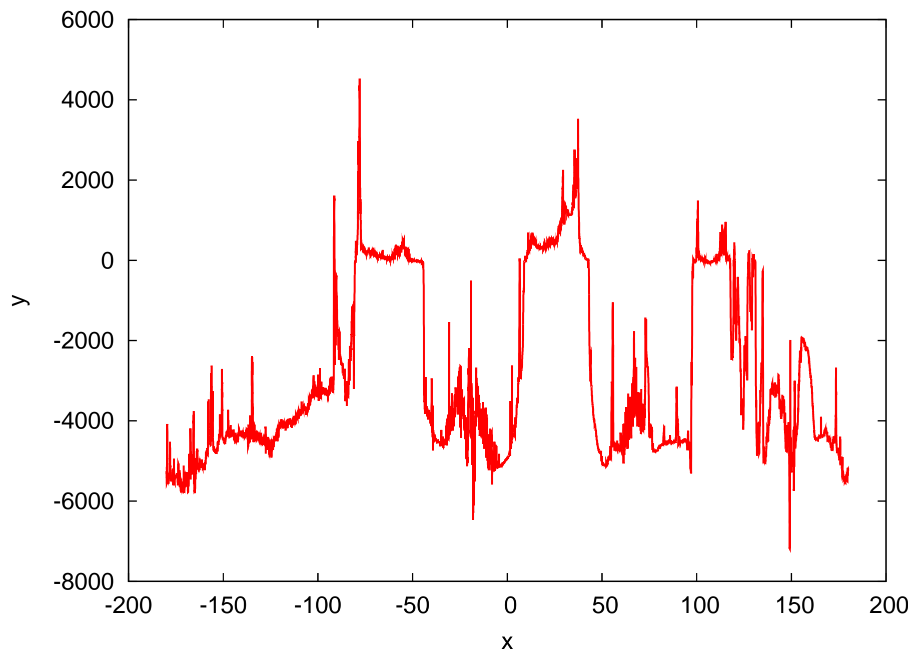
## 3.4 Praktisches zur Diskreten Fourier Analyse

Abbildung 20 zeigt einige Beispiele für Spektren. Die Konstante im Zeitbereich hat einen Puls als Spektrum. Umgedreht hat ein Puls im Ortsbereich ein konstantes Spektrum. Schließlich wird noch das Spektrum eines Dreiecks- bzw. Rechtecksignals gezeigt.

Abbildung 21 zeigt die Interpolation von Dreick- bzw. Rechtecksignal bei Vorgabe von jeweils acht Datenpunkten.

Abbildung 22 illustriert die Verbesserung der Annäherung an die zu interpolierende Funktion bei steigendem Parameter $n$.

Abbildung 22 illustriert die Verbesserung der Annäherung bei unstetigen Funktionen, wenn an der Sprungstelle der Mittelwert vorgeschrieben wird. Wir verwenden einmal $n = 15$ (Sprungstelle ist Interpolationspunkt, Mittelwert wird vorgeschrieben) und $n = 16$ (Sprungstelle ist kein Interpolationspunkt).



# 4 Quadratur

## 4.1 Newton-Cotes Formeln

Wir betrachten folgende bestimmte Integrale:

(*i*) Eine einfache, unendlich oft differenzierbare Funktion:

$$\int_0^{\pi/2} \sin(x)\, dx = 1.$$

(*ii*) Eine glatte Funktion aber mit großen höheren Ableitungen:

$$\int_{-1}^{1} \frac{1}{10^{-5} + x^2}\, dx = 9.914588332462438 \cdot 10^2.$$

(*iii*) Eine nicht unendlich oft differenzierbare Funktion (Halbkreis):

$$\int_{-1}^{1} \sqrt{1 - x^2}\, dx = {}^{\pi}\!/_{2}.$$

Summierte Trapezregel für (*i*).

| *I* | Fehler | #Fktausw. |
|---|---|---|
| 9.480594489685199e-01 | 5.1941e-02 | 3 |
| 9.871158009727754e-01 | 1.2884e-02 | 5 |
| 9.967851718861696e-01 | 3.2148e-03 | 9 |
| 9.991966804850723e-01 | 8.0332e-04 | 17 |
| 9.997991943200188e-01 | 2.0081e-04 | 33 |
| 9.999498000921015e-01 | 5.0200e-05 | 65 |
| 9.999874501175253e-01 | 1.2550e-05 | 129 |
| 9.999968625352869e-01 | 3.1375e-06 | 257 |
| 9.999992156341920e-01 | 7.8437e-07 | 513 |
| ... | | |
| 9.999999999995609e-01 | 4.3909e-13 | 524289 |
| 1.000000000003847e+00 | 3.8467e-12 | 1048577 |

Fehler viertelt sich jeweils, und das von Anfang an. Weniger als $10^{-13}$ wird mit `double` Genauigkeit nicht erreicht.

Summierte Simpsonregel für (*i*).

| *I* | Fehler | #Fktausw. |
|---|---|---|
| 1.000134584974194e+00 | 1.3458e-04 | 5 |
| 1.000008295523968e+00 | 8.2955e-06 | 9 |
| 1.000000516684706e+00 | 5.1668e-07 | 17 |
| 1.000000032265001e+00 | 3.2265e-08 | 33 |
| 1.000000002016129e+00 | 2.0161e-09 | 65 |
| 1.000000000126001e+00 | 1.2600e-10 | 129 |
| 1.000000000007874e+00 | 7.8739e-12 | 257 |
| 1.000000000000491e+00 | 4.9094e-13 | 513 |
| 1.000000000000030e+00 | 2.9976e-14 | 1025 |
| 1.000000000000006e+00 | 5.7732e-15 | 2049 |
| 1.000000000000002e+00 | 1.7764e-15 | 4097 |

Fehler reduziert sich jeweils um den Faktor $16 = ({}^1\!/_2)^4$, und das fast von Anfang an.

Summierte Trapezregel für (*ii*).

| $I$ | Fehler | #Fktausw. |
|---|---|---|
| 1.000009999900001e+05 | 9.9010e+04 | 3 |
| 5.000449983500645e+04 | 4.9013e+04 | 5 |
| 2.501113751079469e+04 | 2.4020e+04 | 9 |
| 1.252430268327760e+04 | 1.1533e+04 | 17 |
| 6.300548144658167e+03 | 5.3091e+03 | 33 |
| 3.227572909110977e+03 | 2.2361e+03 | 65 |
| 1.765586982280199e+03 | 7.7413e+02 | 129 |
| 1.160976493727309e+03 | 1.6952e+02 | 257 |
| 1.003813438906513e+03 | 1.2355e+01 | 513 |
| 9.915347090712996e+02 | 7.5876e-02 | 1025 |
| 9.914588358257512e+02 | 2.5795e-06 | 2049 |
| 9.914588331667655e+02 | 7.9478e-08 | 4097 |
| 9.914588332263689e+02 | 1.9875e-08 | 8193 |
| 9.914588332412698e+02 | 4.9740e-09 | 16385 |

Fehlerverhalten am Anfang unklar, erst spät stellt sich $h^2$ ein.
Summierte Simpsonregel für $(ii)$.

| $I$ | Fehler | #Fktausw. |
|---|---|---|
| 3.333899978334190e+04 | 3.2348e+04 | 5 |
| 1.668001673605744e+04 | 1.5689e+04 | 9 |
| 8.362024407438566e+03 | 7.3706e+03 | 17 |
| 4.225963298451690e+03 | 3.2345e+03 | 33 |
| 2.203247830595247e+03 | 1.2118e+03 | 65 |
| 1.278258340003273e+03 | 2.8680e+02 | 129 |
| 9.594396642096787e+02 | 3.2019e+01 | 257 |
| 9.514257539662473e+02 | 4.0033e+01 | 513 |
| 9.874417991262286e+02 | 4.0170e+00 | 1025 |
| 9.914335447439017e+02 | 2.5289e-02 | 2049 |
| 9.914588322804369e+02 | 9.6581e-07 | 4097 |
| 9.914588332462367e+02 | 7.0486e-12 | 8193 |
| 9.914588332462367e+02 | 7.0486e-12 | 16385 |

Bis 4096 Auswertungen ist Simpson schlechter als Trapez. „Asymptotische Konvergenzrate“ stellt sich erst für genügend kleines $h$ ein.
Summierte Trapezregel für $(iii)$.

| $I$ | Fehler | #Fktausw. |
|---|---|---|
| 1.000000000000000e+00 | 5.7080e-01 | 3 |
| 1.366025403784439e+00 | 2.0477e-01 | 5 |
| 1.497854534051220e+00 | 7.2942e-02 | 9 |
| 1.544909572178587e+00 | 2.5887e-02 | 17 |
| 1.561626518913870e+00 | 9.1698e-03 | 33 |
| 1.567551211438566e+00 | 3.2451e-03 | 65 |
| 1.569648456389842e+00 | 1.1479e-03 | 129 |
| 1.570390396198308e+00 | 4.0593e-04 | 257 |
| 1.570652791478614e+00 | 1.4354e-04 | 513 |
| 1.570745576359828e+00 | 5.0750e-05 | 1025 |
| 1.570778383269506e+00 | 1.7944e-05 | 2049 |
| 1.570789982705718e+00 | 6.3441e-06 | 4097 |
| 1.570794083803873e+00 | 2.2430e-06 | 8193 |
| 1.570795533774854e+00 | 7.9302e-07 | 16385 |

Die Konvergenzordnung $h^2$ wird nicht erreicht, sondern nur ein $h^\alpha$ mit $\alpha < 2$ (siehe unten).
Summierte Simpsonregel für $(iii)$.

| $I$ | Fehler | #Fktausw. |
|---|---|---|
| 1.488033871712585e+00 | 8.2762e-02 | 5 |
| 1.541797577473481e+00 | 2.8999e-02 | 9 |
| 1.560594584887709e+00 | 1.0202e-02 | 17 |
| 1.567198834492299e+00 | 3.5975e-03 | 33 |
| 1.569526108946797e+00 | 1.2702e-03 | 65 |
| 1.570347538040268e+00 | 4.4879e-04 | 129 |
| 1.570637709467796e+00 | 1.5862e-04 | 257 |
| 1.570740256572051e+00 | 5.6070e-05 | 513 |
| 1.570776504653564e+00 | 1.9822e-05 | 1025 |
| 1.570789318906069e+00 | 7.0079e-06 | 2049 |
| 1.570793849184461e+00 | 2.4776e-06 | 4097 |
| 1.570795450836595e+00 | 8.7596e-07 | 8193 |
| 1.570796017098507e+00 | 3.0970e-07 | 16385 |

Die Simpsonregel zeigt *dieselbe* Konvergenzordnung wie die Trapezregel!
Konvergenzabschätzung hat die Form

$$|I(f) - I_h^{(n)}(f)| \leq Ch^{m+1}.$$

Summierte Trapezregel: $m = 1$, $h^2$ - Konvergenz.
Summierte Simpsonregel: $m = 3$, $h^4$ - Konvergenz.
Experimentelle Bestimmung der Konvergenzrate:
Mit dem Ansatz $e_h = |I(f) - I_h^{(n)}(f)| = Ch^\alpha$ gilt

$$\frac{e_{h/2}}{e_h} = \frac{C(h/2)^\alpha}{Ch^\alpha} = (1/2)^\alpha$$

und daraus erhalten wir

$$\alpha = \log\left(\frac{e_{h/2}}{e_h}\right) \Big/ \log\left(\frac{1}{2}\right).$$

Das so bestimmte $\alpha$ heißt *experimental order of convergence* (EOC).
Im letzen Beispiel oben erhalten wir $\alpha = 3/2$.

## 4.2 Gauß- und adaptive Quadratur

Verschiedene Quadraturen für $(i)$ aus letztem Beispiel.

| Methode | $I$ | Fehler | #Fktausw. |
|---|---|---|---|
| Gauss4 | 9.999101667698898e-01 | 8.9833e-05 | 4 |
| | 9.999944679581383e-01 | 5.5320e-06 | 8 |
| | 9.999996555171785e-01 | 3.4448e-07 | 16 |
| | 9.999999784895880e-01 | 2.1510e-08 | 32 |
| Gauss6 | 1.000000118910998e+00 | 1.1891e-07 | 6 |
| | 1.000000001828737e+00 | 1.8287e-09 | 12 |
| | 1.000000000028461e+00 | 2.8461e-11 | 24 |
| | 1.000000000000444e+00 | 4.4409e-13 | 48 |
| Archi | 9.480594489685199e-01 | 5.1941e-02 | 3 |
| | 9.871158009727754e-01 | 1.2884e-02 | 5 |
| | 9.967851718861697e-01 | 3.2148e-03 | 9 |
| | 9.990131153231707e-01 | 9.8688e-04 | 15 |
| | 9.997876171856270e-01 | 2.1238e-04 | 31 |

Das Verfahren hoher Ordnung zahlt sich aus.

| Methode | $I$ | Fehler | #Fktausw. |
|---|---|---|---|
| Trapez | 1.000009999900001e+05 | 9.9010e+04 | 3 |
| | 3.227572909110977e+03 | 2.2361e+03 | 65 |
| | 1.765586982280199e+03 | 7.7413e+02 | 129 |
| | 1.160976493727309e+03 | 1.6952e+02 | 257 |
| | 1.003813438906513e+03 | 1.2355e+01 | 513 |
| | 9.915347090712996e+02 | 7.5876e-02 | 1025 |
| | 9.914588358257512e+02 | 2.5795e-06 | 2049 |
| Archi | 1.767335925226728e+03 | 7.7588e+02 | 25 |
| | 1.004348965298925e+03 | 1.2890e+01 | 37 |
| | 9.946212584262852e+02 | 3.1624e+00 | 81 |
| | 9.922788393957054e+02 | 8.2001e-01 | 173 |
| | 9.916266302474447e+02 | 1.6780e-01 | 361 |
| | 9.914967844457766e+02 | 3.7951e-02 | 769 |
| | 9.914672523966888e+02 | 8.4192e-03 | 1625 |
| | 9.914606991793892e+02 | 1.8659e-03 | 3465 |
| | 9.914592092819358e+02 | 3.7604e-04 | 7629 |

Fehlerreduktion mit Archi ist von Anfang an quadratisch, allerdings „überholt“ die Trapezsumme dann kräftig.

| Methode | $I$ | Fehler | #Fktausw. |
|---|---|---|---|
| Gauss4 | 1.592226038754547e+00 | 2.1430e-02 | 4 |
| | 1.570801362699711e+00 | 5.0359e-06 | 1024 |
| | 1.570798107100650e+00 | 1.7803e-06 | 2048 |
| | 1.570796956200537e+00 | 6.2941e-07 | 4096 |
| | 1.570796549318533e+00 | 2.2252e-07 | 8192 |
| Gauss6 | 1.578036347519909e+00 | 7.2400e-03 | 6 |
| | 1.570801237513435e+00 | 4.9107e-06 | 768 |
| | 1.570798062869299e+00 | 1.7361e-06 | 1536 |
| | 1.570796940567470e+00 | 6.1377e-07 | 3072 |
| | 1.570796543792309e+00 | 2.1700e-07 | 6144 |
| Archi | 1.366025403784439e+00 | 2.0477e-01 | 5 |
| | 1.570774639679624e+00 | 2.1687e-05 | 365 |
| | 1.570791453003758e+00 | 4.8738e-06 | 765 |
| | 1.570795219591928e+00 | 1.1072e-06 | 1605 |
| | 1.570796082320714e+00 | 2.4447e-07 | 3433 |

Hohe Ordnung lohnt sich nicht wegen mangelnder Differenzierbarkeit.

# 5 Ein kleiner Programmierkurs

## 5.1 Hallo Welt

### Programmierumgebung

- Wir benutzen die Programmiersprache C++.
- Wir behandeln nur die Programmierung unter LINUX mit den GNU compilern.
- Windows: On your own.
- Wir setzen Grundfertigkeit im Umgang mit LINUX-Rechnern voraus:
  - Shell, Kommandozeile, Starten von Programmen.
  - Dateien, Navigieren im Dateisystem.
  - Erstellen von Textdatein mit einem Editor ihrer Wahl.
- Idee des Kurses: „Lernen an Beispielen“, keine rigorose Darstellung.
- Blutige Anfänger sollten zusätzlich ein Buch lesen (siehe Literaturliste).

### Workflow

C++ ist eine „kompilierte“ Sprache. Um ein Programm zur Ausführung zu bringen sind folgende Schritte notwendig:

1. Erstelle/Ändere den Programmtext mit einem **Editor**.
2. Übersetze den Programmtext mit dem **C++-Übersetzer** (auch C++-Compiler) in ein Maschinenprogramm.
3. Führe das Programm aus. Das Programm gibt sein Ergebnis auf dem Bildschirm oder in eine Datei aus.

4. Interpretiere Ergebnisse. Dazu benutzen wir weitere Programme wie **gnuplot** oder **grep**.

5. Falls Ergebnis nicht korrekt, gehe nach 1!

### HDNUM

- C++ kennt keine Matrizen, Vektoren, Polynome, ...
- Wir haben C++ erweitert um die **Heidelberg Educational Numerics Library**, kurz **HDNum**.
- Alle in der Vorlesung behandelten Beispiele sind dort enthalten.

### Herunterladen von HDNUM

1. Einloggen
2. Erzeuge neues Verzeichnis mit $ mkdir kurs
3. Wechsle in das Verzeichnis mit $ cd kurs
4. Gehe zur Webseite http://conan.iwr.uni-heidelberg.de/teaching/numerik0_ws2009/
5. Klicke auf **Version 0.10, Stand 12.10.2009** und bestätige
6. Kopiere Datei hdnum-0.10.tgz in das Verzeichnis: $ cp ~/Desktop/hdnum-0.10.tgz .
7. Entpacken der Datei mit $ tar zxvf hdnum-0.10.tgz
8. Wechsle in das Verzeichnis $ cd hdnum/examples
9. Anzeigen der Dateien mittels $ ls

### Wichtige UNIX-Befehle

- ls --color -F - Zeige Inhalt des aktuellen Verzeichnisses
- cd - Wechsle ins Home-Verzeichnis
- cd <verzeichnis> - Wechsle in das angegebene Verzeichnis (im aktuellen Verzeichnis)
- cd .. - Gehe aus aktuellem Verzeichnis heraus
- mkdir < verzeichnis> - Erstelle neues Verzeichnis

- cp <datei1> <datei2> - Kopiere datei1 auf datei2 (datei2 kann durch Verzeichnis ersetzt werden)
- mv <datei1> <datei2> - Benenne datei1 in datei2 um (datei2 kann durch Verzeichnis ersetzt werden, dann wird datei1 dorthin verschoben)
- rm <datei> - Lösche datei
- rm -rf <verzeichnis> - Lösche Verzeichnis mit allem darin

### Hallo Welt !

Öffne die Datei **hallowelt.cc** mit einem Editor: **$ gedit hallowelt.cc**

```
1 // hallowelt.cc (Dateiname als Kommentar)
2 #include <iostream>  // notwendig zur Ausgabe
3
4 int main ()
5 {
6   std::cout << "Numerik 0 ist leicht:" << std::endl;
7   std::cout << "1+1=" << 1+1 << std::endl;
8 }
```

- **iostream** ist eine sog. „Headerdatei“
- **#include** erweitert die „Basissprache“.
- **int main ()** braucht man immer: „Hier geht’s los“.
- **{ ... }** klammert Folge von Anweisungen.
- Anweisungen werden durch Semikolon abgeschlossen.

### Hallo Welt laufen lassen

- Gebe folgende Befehle ein:

  ```
  $ g++ -o hallowelt hallowelt.cc
  $ ./hallowelt
  ```

- Dies sollte dann die folgende Ausgabe liefern:

  ```
  Numerik 0 ist ganz leicht:
  1+1=2
  ```

## 5.2 Variablen und Typen

### (Zahl-) Variablen

- Aus der Mathematik: „$x \in M$“. Variable $x$ nimmt einen beliebigen Wert aus der Menge $M$ an.
- Geht in C++ mit: `M x;`
- **Variablendefinition**: `x` ist eine Variable vom **Typ** `M`.
- Mit **Initialisierung**: `M x(0);`
- Wert von Variablen der „eingebauten“ Typen ist sonst nicht definiert.

```
1 // zahlen.cc
2 #include <iostream>
3 int main ()
4 {
5   unsigned int i; // uninitialisierte natürliche Zahl
6   double x(3.14); // initialisierte Fließkommazahl
7   float y(1.0);   // einfache Genauigkeit
8   short j(3);     // eine "kleine" Zahl
9   std::cout << "(i+x)*(y+j)=" << (i+x)*(y+j) << std::endl;
10 }
```

### Andere Typen

- C++ kennt noch viele weitere Typen.
- Typen können nicht nur Zahlen sondern viele andere Informationen repräsentieren.
- Etwa Zeichenketten: `std::string`
- Oft muss man dazu weitere Headerdateien angeben.

```
1 // string.cc
2 #include <iostream>
3 #include <string>
4 int main ()
5 {
6   std::string m1("Zeichen");
7   std::string leer("␣␣␣");
8   std::string m2("kette");
9   std::cout << m1+leer+m2 << std::endl;
10 }
```

- Jede Variable *muss* einen Typ haben. Strenge Typbindung.

### Mehr Zahlen

```
// mehrzahlen.cc
#include <iostream> // header für Ein-/Ausgabe
#include <gmpxx.h>  // header für GNU multiprecision
#include <complex>  // header für komplexe Zahlen
int main ()
{
  mpf_class x("3.14159265358979323846264338327950288 41",512);
  std::complex<double> y(1.0,3.0);
  std::complex<mpf_class> z(x,x);
  std::cout << x << "␣" << y << "␣" << z << std::endl;
}
```

- GNU Multiprecision Library `http://gmplib.org/` erlaubt Zahlen mit vielen Stellen (hier 512 Stellen zur Basis 2).
- Übersetzen mit: `$ g++ -o mehrzahlen mehrzahlen.cc -lgmpxx -lgmp`
- Komplexe Zahlen sind Paare von Zahlen.
- `complex<>` ist ein Template: Baue komplexe Zahlen aus jedem anderen Zahlentyp auf (später mehr!).

### Mehr Ein- und Ausgabe

```
// eingabe.cc
#include <iostream> // header für Ein-/Ausgabe
#include <iomanip>  // für setprecision etc.
#include <gmpxx.h>  // header für GNU multiprecision
int main ()
{
  mpf_class x("0.0",512);
  std::cout << "Gebe␣eine␣lange␣Zahl␣ein:␣";
  std::cin >> x;
  std::cout << "Wurzel(x)=␣"
            << std::scientific << std::showpoint
            << std::setprecision(15)
            << sqrt(x) << std::endl;
}
```

- Eingabe geht mit `std::cin >> x;`
- Standardmäßig werden nur 6 Nachkommastellen ausgegeben. Das ändert man mit `std::setprecision`.
- Dazu muss man die Headerdatei `iomanip` einbinden.
- Die Wurzel berechnet die Funktion `sqrt`.

### Zuweisung

- Den Wert von Variablen kann man ändern. Sonst wäre es langweilig :-)
- Dies geht mittels Zuweisung:

```
double x(3.14); // Variablendefinition mit Initialisierung
double y;       // uninitialisierte Variable
y = x;          // Weise y den Wert von x zu
x = 2.71;       // Weise x den Wert 2.71, y unverändert
y = (y*3)+4;    // Werte Ausdruck rechts von = aus
                // und weise das Resultat y zu!
```

### Blöcke

- Block: Sequenz von Variablendefinitionen und Zuweisungen in geschweiften Klammern.

```
{
  double x(3.14);
  double y;
  y = x;
}
```

- Blöcke können rekursiv geschachtelt werden.
- Eine Variable ist nur in dem Block *sichtbar* in dem sie definiert ist sowie in allen darin enthaltenen Blöcken:

```
{
  double x(3.14);
  {
    double y;
    y = x;
  }
  y = (y*3)+4; // geht nicht, y nicht mehr sichtbar.
}
```

### Whitespace

- Das Einrücken von Zeilen dient der besseren Lesbarkeit, notwendig ist es (fast) nicht.
- `#include`-Direktiven müssen *immer* einzeln auf einer Zeile stehen.
- Ist das folgende Programm lesbar?

```
1 // whitespace.cc
2 #include <iostream> // includes auf eigener Zeile!
3 #include <iomanip>
4 #include "math.h"
5 int main(){double x(0.0);
6 std::cout<<"Gebe␣eine␣lange␣Zahl␣ein:␣";std::cin >> x;
7 std::cout<<"Wurzel(x)=␣"<<std::scientific<<std::showpoint
8 <<std::setprecision(16)<<sqrt(x)<< std::endl;}
```

## 5.3 Entscheidung

### `If`-Anweisung

- Aus der Mathematik kennt man eine „Zuweisung“ der folgenden Art.

  Für $x \in \mathbb{R}$ setze
  $$y = |x| = \begin{cases} x & \text{falls } x \leq 0 \\ -x & sonst \end{cases}$$

- Dies realisiert man in C++ mit einer `If`-Anweisung:

```
double x(3.14), y;
if (x>=0)
{
  y = x;
}
else
{
  y = -x;
}
```

### Varianten der `If`-Anweisung

- Die geschweiften Klammern kann man weglassen, wenn der Block nur eine Anweisung enthält:

```
double x(3.14), y;
if (x>=0) y = x; else y = -x;
```

- Der `else`-Teil ist optional:

```
double x=3.14;
if (x<0)
  std::cout << "x␣ist␣negativ!" << std::endl;
```

- Weitere Vergleichsoperatoren sind `< <= == >= > !=`

- Beachte: `=` für Zuweisung, aber `==` für den Vergleich zweier Objekte!

## 5.4 Wiederholung

### `While`-Schleife

- Bisher: Sequentielle Abfolge von Befehlen wie im Programm angegeben. Das ist langweilig :-)

- Eine Möglichkeit zur Wiederholung bietet die `While`-Schleife:

  `while (` *Bedingung* `)`

  `{` *Schleifenkörper* `}`

- Beispiel:

```
int i=0; while (i<10) { i=i+1; }
```

- Bedeutung:
  1. Teste Bedingung der `While`-Schleife
  2. Ist diese *wahr* dann führe Anweisungen im Schleifenkörper aus, sonst gehe zur ersten Anweisung nach dem Schleifenkörper.
  3. Gehe nach 1.

- Anweisungen im Schleifenkörper beeinflussen normalerweise den Wahrheitswert der Bedingung.

- Endlosschleife: Wert der Bedingung wird nie *falsch*.

### Pendel (analytische Lösung; `while`-Schleife)

- Die Auslenkung des Pendels mit der Näherung $\sin(\phi) \approx \phi$ und $\phi(0) = \phi_0$, $\phi'(0) = 0$ lautet:

$$\phi(t) = \phi_0 \cos\left(\sqrt{\frac{g}{l}}\right).$$

- Das folgende Programm gibt diese Lösung zu den Zeiten $t_i = i\Delta t$, $0 \leq t_i \leq T$, $i \in \mathbb{N}_0$ aus:

```
1  // pendelwhile.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <math.h>   // mathematische Funktionen
4  int main ()
5  {
6    double l(1.34);    // Pendellänge in Meter
7    double phi0(0.2);  // Amplitude im Bogenmaß
8    double dt(0.05);   // Zeitschritt in Sekunden
9    double T(30.0);    // Ende in Sekunden
10   double t(0.0);     // Anfangswert
11
12   while ( t<=T )
13     {
14       std::cout << t << " "
15                 << phi0*cos(sqrt(9.81/l)*t)
16                 << std::endl;
17       t = t + dt;
18     }
19 }
```

### Wiederholung (for-Schleife)

- Möglichkeit der Wiederholung: `for`-Schleife:

  `for (` *Anfang*; *Bedingung*; *Inkrement* `)`

  `{` *Schleifenkörper* `}`

- Beispiel:

```
for (int i=0; i<=5; i=i+1)
{
  std::cout << "Wert␣von␣i␣ist␣" << i << std::endl;
}
```

- Enthält der Block nur eine Anweisung dann kann man die geschweiften Klammern weglassen.
- Die *Schleifenvariable* ist so nur innerhalb des Schleifenkörpers sichtbar.
- Die `for`-Schleife kann auch mittels einer *while*-Schleife realisiert werden.

### Pendel (analytische Lösung, `for`-Schleife)

```
1  // pendel.cc
2  #include <iostream> // header für Ein-/Ausgabe
3  #include <math.h>   // mathematische Funktionen
4  int main ()
5  {
6    double l(1.34);    // Pendellänge in Meter
7    double phi0(0.2);  // Amplitude im Bogenmaß
8    double dt(0.05);   // Zeitschritt in Sekunden
9    double T(30.0);    // Ende in Sekunden
10   for (double t=0.0; t<=T; t=t+dt)
11     std::cout << t << "␣"
12               << phi0*cos(sqrt(9.81/l)*t)
13               << std::endl;
14 }
```

### Visualisierung mit `Gnuplot`

- `Gnuplot` erlaubt einfache Visualisierung von Funktionen $f : \mathbb{R} \to \mathbb{R}$ und $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.
- Für $f : \mathbb{R} \to \mathbb{R}$ genügt eine zeilenweise Ausgabe von Argument und Funktionswert.
- Umlenken der Ausgabe eines Programmes in eine Datei: `$ ./pendel > pendel.dat$`
- Starte `gnuplot` `gnuplot> plot "pendel.dat"with lines`

### Geschachtelte Schleifen

- Ein Schleifenkörper kann selbst wieder eine Schleife enthalten, man spricht von *geschachtelten* Schleifen.
- Beispiel:

```
for (int i=1; i<=10; i=i+1)
  for (int j=1; j<=10; j=j+1)
    if (i==j)
      std::cout << "i␣gleich␣j:␣" << std::endl;
    else
      std::cout << "i␣ungleich␣j!" << std::endl;
```

### Numerische Lösung des Pendels

- Volles Modell für das Pendel aus der Einführung:

$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{l}\sin(\phi(t)) \qquad \forall t > 0,$$
$$\phi(0) = \phi_0, \qquad \frac{d\phi}{dt}(0) = u_0.$$

- Umschreiben in System erster Ordnung:

$$\frac{d\phi(t)}{dt} = u(t), \qquad \frac{d^2\phi(t)}{dt^2} = \frac{du(t)}{dt} = -\frac{g}{l}\sin(\phi(t)).$$

- Eulerverfahren für $\phi^n = \phi(n\Delta t)$, $u^n = u(n\Delta t)$:

$$\begin{aligned} \phi^{n+1} &= \phi^n + \Delta t\, u^n & \phi^0 &= \phi_0 \\ u^{n+1} &= u^n - \Delta t\,(g/l)\,\sin(\phi^n) & u^0 &= u_0 \end{aligned}$$

### Pendel (expliziter Euler)

```
1 // pendelnumerisch.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <math.h>   // mathematische Funktionen
4
5 int main ()
6 {
7   double l(1.34);  // Pendellänge in Meter
8   double phi(3.0); // Anfangsamplitude in Bogenmaß
9   double u(0.0);   // Anfangsgeschwindigkeit
10  double dt(1E-4); // Zeitschritt in Sekunden
11  double T(30.0);  // Ende in Sekunden
12  double t(0.0);   // Anfangszeit
13
14  std::cout << t << "␣" << phi << std::endl;
15  while (t<T)
16    {
17      t = t + dt;        // inkrementiere Zeit
```

```
18        double phialt(phi);// merke phi
19        double ualt(u);    // merke u
20        phi = phialt + dt*ualt;             // neues phi
21        u = ualt - dt*(9.81/l)*sin(phialt);// neues u
22        std::cout << t << " " << phi << std::endl;
23     }
24 }
```

## 5.5 Funktionen

### Funktionsaufruf und Funktionsdefinition

- In der Mathematik gibt es das Konzept der *Funktion*.
- In C++ auch.
- Sei $f : \mathbb{R} \to \mathbb{R}$, z.B. $f(x) = x^2$.
- Wir unterscheiden den *Funktionsaufruf*

  ```
  double x,y;
  y = f(x);
  ```

- und die *Funktionsdefinition*. Diese sieht so aus:

  *Ergebnistyp Funktionsname* ( *Argumente* )

  { *Funktionsrumpf* }

- Beispiel:

  ```
  double f (double x)
  {
    return x*x;
  }
  ```

### Komplettbeispiel zur Funktion

```
1 // funktion.cc
2 #include <iostream>
3
4 double f (double x)
5 {
6   return x*x;
7 }
8
9 int main ()
10 {
11   double x(2.0);
12   std::cout << "f(" << x << ")=" << f(x) << std::endl;
13 }
```

- Funktionsdefinition muss vor Funktionsaufruf stehen.

- Formales Argument in der Funktionsdefinition entspricht einer Variablendefinition.
- Beim Funktionsaufruf wird das Argument (hier) *kopiert*.
- `main` ist auch nur eine Funktion.

### Weiteres zum Verständnis der Funktion

- Der Name des formalen Arguments in der Funktionsdefinition ändert nichts an der Semantik der Funktion (Sofern es überall geändert wird):

```
double f (double y)
{
  return y*y;
}
```

- Das Argument wird hier kopiert, d.h.:

```
double f (double y)
{
  y = 3*y*y;
  return y;
}

int main ()
{
  double x(3.0),y;
  y = f(x); // ändert nichts an x !
}
```

### Weiteres zum Verständnis der Funktion

- Argumentliste kann leer sein (wie in der Funktion `main`):

```
double pi ()
{
  return 3.14;
}

y = pi(); // Klammern sind erforderlich!
```

- Der Rückgabetyp `void` bedeutet „keine Rückgabe“

```
void hello ()
{
  std::cout << "hello" << std::endl;
}

hello();
```

- Mehrere Argument werden durch Kommata getrennt:

```
double g (int i, double x)
{
  return i*x;
}
std::cout << g(2,3.14) << std::endl;
```

### Pendelsimulation als Funktion

```
1 // pendelmitfunktion.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <math.h>   // mathematische Funktionen
4
5 void simuliere_pendel (double l, double phi, double u) {
6   double dt   = 1E-4;
7   double T    = 30.0;
8   double t    = 0.0;
9
10  std::cout << t << "␣" << phi << std::endl;
11  while (t<T) {
12      t = t + dt;
13      double phialt(phi),ualt(u);
14      phi = phialt + dt*ualt;
15      u = ualt - dt*(9.81/l)*sin(phialt);
16      std::cout << t << "␣" << phi << std::endl;
17    }
18 }
19
20 int main () {
21   double l(1.34); // Pendellänge in Meter
22   double phi(3.0);// Anfangsamplitude in Bogenmaß
23   double u(0.0);  // Anfangsgeschwindigkeit
24   simuliere_pendel(l,phi,u); }
```

### Funktionsschablonen

- Oft macht eine Funktion mit Argumenten verschiedenen Typs einen Sinn.
- `double f (double x) {return x*x;}` macht auch mit `float`, `int` oder `mpf_class` Sinn.
- Man könnte die Funktion für jeden Typ definieren. Das ist natürlich sehr umständlich. (Es darf mehrere Funktionen gleichen Namens geben, sog. *overloading*).
- In C++ gibt es mit Funktionsschablonen (engl.: *function templates*) eine Möglichkeit den Typ variabel zu lassen:

```
template<typename T>
T f (T y)
{
  return y*y;
}
```

- `T` steht hier für einen beliebigen Typ.

### Pendelsimulation mit Templates

```
1 // pendelmitfunktionstemplate.cc
2 #include <iostream> // header für Ein-/Ausgabe
3 #include <math.h>   // mathematische Funktionen
4
5 template<typename Number>
6 void simuliere_pendel (Number l, Number phi, Number u) {
```

```
7     Number dt(1E-4);
8     Number T(30.0);
9     Number t(0.0);
10    Number g(9.81/l);
11
12    std::cout << t << "␣" << phi << std::endl;
13    while (t<T) {
14        t = t + dt;
15        Number phialt(phi),ualt(u);
16        phi = phialt + dt*ualt;
17        u = ualt - dt*g*sin(phialt);
18        std::cout << t << "␣" << phi << std::endl;
19      }
20  }
21
22  int main () // geht leider nicht mit GNU MP :-(
23  {
24    float l1(1.34);  // Pendellänge in Meter
25    float phi1(3.0); // Anfangsamplitude in Bogenmaß
26    float u1(0.0);   // Anfangsgeschwindigkeit
27    simuliere_pendel(l1,phi1,u1);
28
29    double l2(1.34);  // Pendellänge in Meter
30    double phi2(3.0); // Anfangsamplitude in Bogenmaß
31    double u2(0.0);   // Anfangsgeschwindigkeit
32    simuliere_pendel(l2,phi2,u2);
33  }
```

#### Referenzargumente

- Das Kopieren der Argumente einer Funktion kann verhindert werden indem man das Argument als *Referenz* definiert:

  ```
  void f (double x, double& y)
  {
    y = x*x;
  }

  double x(3), y;
  f(x,y); // y hat nun den Wert 9, x ist unverändert.
  ```

- Statt eines Rückgabewertes kann man auch ein (zusätzliches) Argument modifizieren.
- Insbesondere kann man so den Fall mehrerer Rückgabewerte realisieren.
- Referenzargumente bieten sich auch an wenn Argumente „sehr groß“ sind und damit das kopieren sehr zeitaufwendig ist.
- Der aktuelle Parameter im Aufruf *muss* dann eine Variable sein.

# 6 Heidelberg Educational Numerics Library

## 6.1 Einführung

#### Was ist HDNUM

- HDNUM ist eine kleine Sammlung von C++ Klassen, die die Implementierung numerischer Algorithmen aus der Vorlesung erleichtern soll.
- Die aktuelle Version gibt es unter

  `http://conan.iwr.uni-heidelberg.de/teaching/numerik0_ws2009/`

- Einige Ziele bei der Entwicklung von HDNUM waren:
  - Einfache Installation: Es mur nur eine Header-Datei eingebunden werden.
  - Einfache Benutzung der Klassen: Z.B. keine dynamische Speicherverwaltung.
  - Möglichkeit der Rechnung mit verschiedenen Zahl-Datentypen.
  - Effiziente Realisierung der Verfahren möglich: Z.B. Block-Algorithmen in der linearen Algebra.

### Installation

- Datei `hdnum-x.yy.tgz` (komprimiertes tar archive) herunterladen.
- Archiv mit `tar zxf hdnum-x.yy.tgz` entpacken.
- Das Verzeichnis enthält unter anderem:
  - Das Verzeichnis `src` mit dem Quellcode der Klassen (muss Sie nicht interessieren).
  - Das Verzeichnis `examples` mit den Beispielanwendungen (die sollten Sie sich ansehen).
  - Die Datei `hdnum.hh`, die zentrale Header-Datei, die in alle Anwendungen eingebunden werden muss.
- Das Verzeichnis `hdnum/examples` enthält ein simples Makefile zum Übersetzen der Programme.
- Die Beispiele erfordern die Installation der GNU multiprecision library `http://gmplib.org/`. Ist diese nicht vorhanden müssen Makefiles entsprechend angepasst werden.

### Typisches HDNUM Programm

```
1  // hallohdnum.cc
2  #include <iostream>     // notwendig zur Ausgabe
3  #include "hdnum.hh"     // hdnum header
4  using namespace hdnum; // Namen ohne hdnum:: verwenden
5
6  int main ()
7  {
8    Array<float> a(10,3.14); // Feld mit 10 init. Elementen
9    a[3] = 1.0;              // Zugriff auf Element 3
10 }
```

- Übersetzen im Verzeichnis examples mit GMP installiert: g++ -I.. -o hallohdnum hallohdnum.cc -lm -
- und ohne GMP: g++ -I.. -o hallohdnum hallohdnum.cc -lm
- oder einfach make
- oder falls kein GMP installiert ist make nogmp

## 6.2 Vektoren

Vector<T>

- Vector<T> ist ein Klassen-Template.
- Es macht aus einem beliebigen (Zahl-)Datentypen T einen Vektor.
- Auch komplexe und hochgenaue Zahlen sind möglich.
- Vektoren verhalten sich so wie man es aus der Mathematik kennt:
  - Bestehen aus $n$ Komponenten.
  - Diese sind von 0 bis $n-1$ (!) durchnummeriert.
  - Addition und Multiplikation mit Skalar.
  - Skalarprodukt und Norm (noch nicht implementiert).
  - Matrix-Vektor-Multiplikation
- Die folgenden Beispiele findet man in vektoren.cc

### Konstruktion und Zugriff

- Konstruktion mit und ohne Initialisierung

```
Vector<float> x(10);          // Vektor mit 10 Elementen
Vector<double> y(10,3.14);    // 10 Elemente initialisiert
Vector<float> a;              // ein leerer Vektor
```

- Speziellere Vektoren

```
Vector<std::complex<double> >
  cx(7,std::complex<double>(1.0,3.0));
mpf_set_default_prec(1024); // Setze Genauigkeit für mpf_class
Vector<mpf_class> mx(7,mpf_class("4.44"));
```

- Zugriff auf Element

```
for (std::size_t i=0; i<x.size(); i=i+1)
  x[i] = i;                   // Zugriff auf Elemente
```

- Vektorobjekt wird am Ende des umgebenden Blockes gelöscht.

### Kopie und Zuweisung

- Copy-Konstruktor und Zuweisung haben **Referenzsemantik**!

```
Vector<float> z(x); // Kopie ist eine Referenz auf gleiche Daten
z[2] = 1.24;        // hat den gleichen Effekt wie x[2] = 1.24 !

a = z;              // a referenziert die Daten von z
a[2] = -0.33;       // ändert auch z[2] und x[2];
a = 5.4;            // Zuweisung an alle Elemente
```

- Erstellen einer echten Kopie

```
Vector<float> b(copy(a)); // b ist echte Kopie von a,x,z
a = copy(x);              // a hält echte Kopie der Daten von x
```

- Ausschnitte von Vektoren

```
Vector<float> w(x.sub(7,3));// w referenziert x[7],...,x[9]
z = x.sub(3,4);             // z referenziert x[3],...,x[6]
```

### Rechnen und Ausgabe

- Vektorraumoperationen und Skalarprodukt

```
w += z;             // w = w+z
w -= z;             // w = w-z
w *= 1.23;          // skalare Multiplikation
w /= 1.23;          // skalare Division
w.update(1.23,z);   // w = w + a*z
float s;
s = w*z;            // Skalarprodukt
```

- Ausgabe auf die Konsole

```
std::cout << w << std::endl;// schöne Ausgabe
w.iwidth(2);                // Stellen in Indexausgabe
w.width(20);                // Anzahl Stellen gesamt
w.precision(16);            // Anzahl Nachkommastellen
std::cout << w << std::endl;// nun mit mehr Stellen
std::cout <<cx << std::endl;// geht auch für complex
std::cout <<mx << std::endl;// geht auch für mpf_class
```

### Beispielausgabe

```
[   0]    1.204200e+01
[   1]    1.204200e+01
[   2]    1.204200e+01
[   3]    1.204200e+01
```

```
[ 0] 1.2042000770568848e+01
[ 1] 1.2042000770568848e+01
[ 2] 1.2042000770568848e+01
[ 3] 1.2042000770568848e+01
```

### Hilfsfunktionen

```
zero(w);                          // das selbe wie w=0.0
fill(w,(float)1.0);               // das selbe wie w=1.0
fill(w,(float)0.0,(float)0.1); // w[0]=0, w[1]=0.1, w[2]=0.2, ...
unitvector(w,2);                  // kartesischer Einheitsvektor
gnuplot("test.dat",w);            // gnuplot Ausgabe: i w[i]
gnuplot("test2.dat",w,z);         // gnuplot Ausgabe: w[i] z[i]
```

### Funktionen

- Beispiel: Summe aller Komponenten

```
double sum (Vector<double> x) {
  double s(0.0);
  for (std::size_t i=0; i<x.size(); i=i+1)
    s = s + x[i];
  return s;
}
```

- Mit **Funktionentemplate**:

```
template<class T>
T sum (Vector<T> x) {
  T s(0.0);
  for (std::size_t i=0; i<x.size(); i=i+1)
    s = s + x[i];
  return s;
}
```

- **Vorsicht**: Call-by-value erzeugt **keine** Kopie!

## 6.3 Matrizen

`Matrix<T>`

- `Matrix<T>` ist ein Klassen-Template.
- Es macht aus einem beliebigen (Zahl-)Datentypen `T` eine Matrix.
- Auch komplexe und hochgenaue Zahlen sind möglich.
- Matrizen verhalten sich so wie man es aus der Mathematik kennt:

- Bestehen aus $m \times n$ Komponenten.
- Diese sind von 0 bis $m-1$ bzw. $n-1$ (!) durchnummeriert.
- $m \times n$-Matrizen bilden einen Vektorraum.
- Matrix-Vektor und Matrizenmultiplikation.

- Die folgenden Beispiele findet man in `matrizen.cc`

### Konstruktion und Zugriff

- Konstruktion mit und ohne Initialisierung

```
Matrix<float> A;                // leere Matrix mit Größe 0x0
Matrix<float> B(10,10);         // 10x10 Matrix uninitialisiert
Matrix<float> C(10,10,0.0);     // 10x10 Matrix initialisiert
```

- Zugriff auf Elemente

```
for (int i=0; i<B.rowsize(); ++i)
  for (int j=0; j<B.colsize(); ++j)
    B[i][j] = 0.0;              // jetzt ist B initialisiert
```

- Matrixobjekt wird am Ende des umgebenden Blockes gelöscht.

### Kopie und Zuweisung

- Copy-Konstruktor und Zuweisung haben **Referenzsemantik**!

```
Matrix<float> D(B); // D identisch mit B! Keine Kopie
A = D;              // A ist nun identisch mit D!
A[0][0] = 3.14;     // ändert auch B[0][0], D[0][0]
```

- Erstellen einer echten Kopie

```
Matrix<float> E(copy(B)); // E ist echte Kopie
A = copy(B);              // und auch A
```

- Ausschnitte von Matrizen (Untermatrizen)

```
Matrix<float> F(A.sub(1,2,3,4));// 3x4 Mat ab (1,2)
```

### Rechnen mit Matrizen

- Vektorraumoperationen

```
A += B;             // A = A+B
A -= B;             // A = A-B
A *= 1.23;          // Multiplikation mit Skalar
A /= 1.23;          // Division durch Skalar
A.update(1.23,B);   // A = A + s*B
```

- Matrix-Vektor und Matrizenmultiplikation

```
Vector<float> x(10,1.0); // make two vectors
Vector<float> y(10,2.0);
A.mv(y,x);                // y = A*x
A.umv(y,x);               // y = y + A*x
A.umv(y,(float)-1.0,x);   // y = y + s*A*x
C.mm(A,B);                // C = A*B
C.umm(A,B);               // C = C + A*B
```

## Ausgabe und Hilfsfunktionen

- Ausgabe von Matrizen

```
std::cout << A.sub(0,0,3,3) << std::endl;// schöne Ausgabe
A.iwidth(2);                    // Stellen in Indexausgabe
A.width(10);                    // Anzahl Stellen gesamt
A.precision(4);                 // Anzahl Nachkommastellen
std::cout << A << std::endl;// nun mit mehr Stellen
```

- einige Hilfsfunktionen

```
identity(A);
spd(A);
fill(x,(float)1,(float)1);
vandermonde(A,x);
```

## Beispielausgabe

```
             0            1            2            3
 0   4.0000e+00  -1.0000e+00  -2.5000e-01  -1.1111e-01
 1  -1.0000e+00   4.0000e+00  -1.0000e+00  -2.5000e-01
 2  -2.5000e-01  -1.0000e+00   4.0000e+00  -1.0000e+00
 3  -1.1111e-01  -2.5000e-01  -1.0000e+00   4.0000e+00
```

## Funktion mit Matrixargument

Beispiel einer Funktion, die eine Matrix $A$ und einen Vektor $b$ initialisiert.

```
template<class T>
void initialize (Matrix<T> A, Vector<T> b)
{
  if (A.rowsize()!=A.colsize() || A.rowsize()==0)
    HDNUM_ERROR("need square and nonempty matrix");
  if (A.rowsize()!=b.size())
    HDNUM_ERROR("b must have same size as A");
  for (int i=0; i<A.rowsize(); ++i)
    {
      b[i] = 1.0;
      for (int j=0; j<A.colsize(); ++j)
```

```
        if (j<=i) A[i][j]=1.0; else A[i][j]=0.0;
    }
}
```

## Lehrbücher Numerik


[DH02] Deuflhard, P. und A. Hohmann: *Numerische Mathematik I, Eine algorithmisch orientierte Einführung.* de Gruyter, 2002.

[EEHJ96] Erikson, K., D. Estep, P. Hansbo und C. Johnson: *Computational Differential Equations.* Cambridge University Press, 1996.

[GL89] Golub, G. H. und C. F. Van Loan: *Matrix Computations.* Johns Hopkins University Press, 2nd Auflage, 1989.

[GO96] Golub, G. und J. M. Ortega: *Scientific Computing.* Teubner, 1996.

[QSS07] Quarteroni, A., R. Sacco und F. Saleri: *Numerical Mathematics.* Springer, 2nd Auflage, 2007.

[Ran06] Rannacher, R.: *Einführung in die Numerische Mathematik (Numerik 0).* http://numerik.iwr.uni-heidelberg.de/~lehre/notes, 2006.

[SB05] Stoer, J. und R. Bulirsch: *Numerische Mathematik II.* Springer, 5. Auflage, 2005.

[SK05] Schwarz, H.-R. und N. Köckler: *Numerische Mathematik.* Teubner, 5. Auflage, 2005.

[Sto05] Stoer, J.: *Numerische Mathematik I.* Springer, 9. Auflage, 2005.

[SW05] Schaback, R. und H. Wendland: *Numerische Mathematik.* Springer, 5th Auflage, 2005.


## Lehrbücher C++


[EA00] Eckel, B. und C. Allison: *Thinking in C++: Volume I: Introduction to Standard C++.* Prentice Hall Ptr, 2000. http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

[Erl00] Erlenkötter, H.: *C++ Objektorientiertes Programmieren von Anfang an.* Rowohlt Taschenbuchverlag GmbH, Reinbek bei Hamburg, 2000.

## Weiterführende Literatur

[Knu98] Knuth, D. E.: *The Art of Computer Programming*, Band 2. Addison-Wesley, 3. Auflage, 1998.

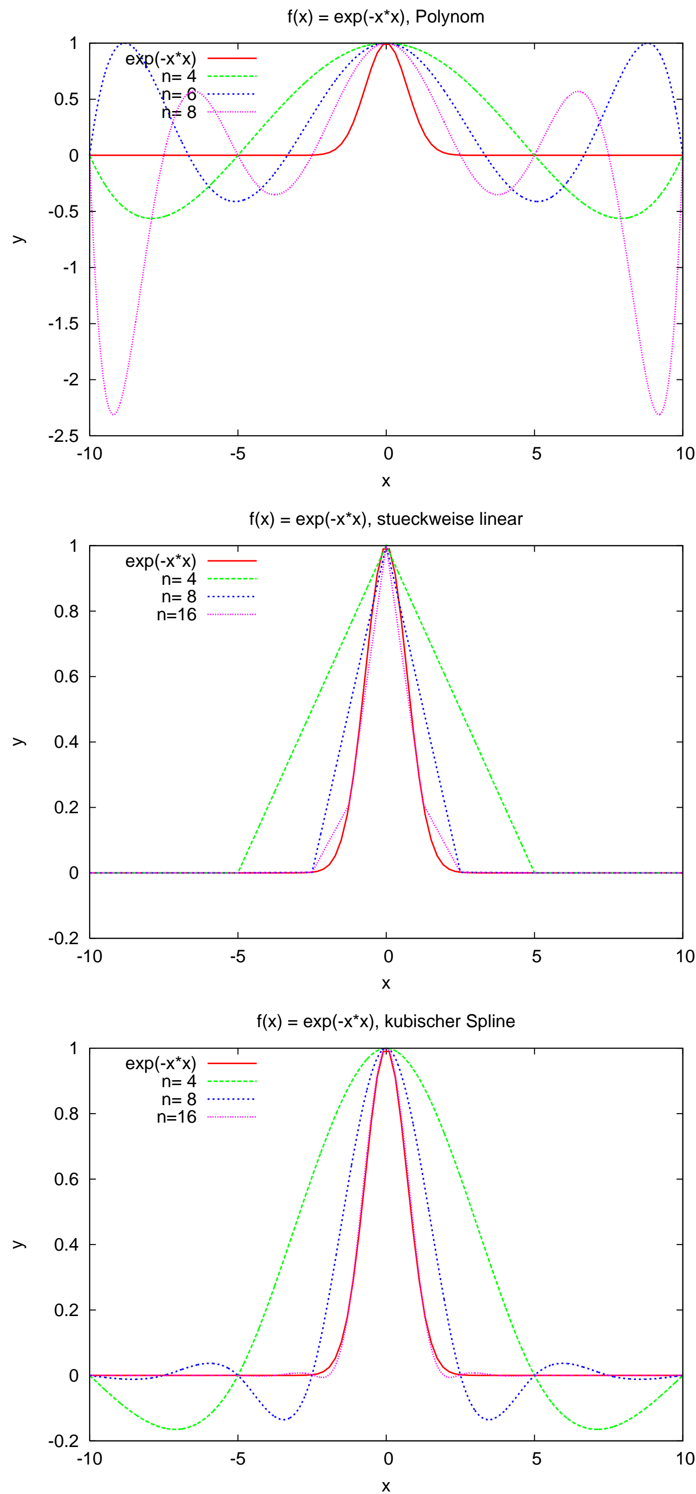



Abbildung 17: Interpolation der Funktion $f_1(x)$ mit Lagrange-Polynomen, stückweise linearen Funktionen und kubischen Splines.

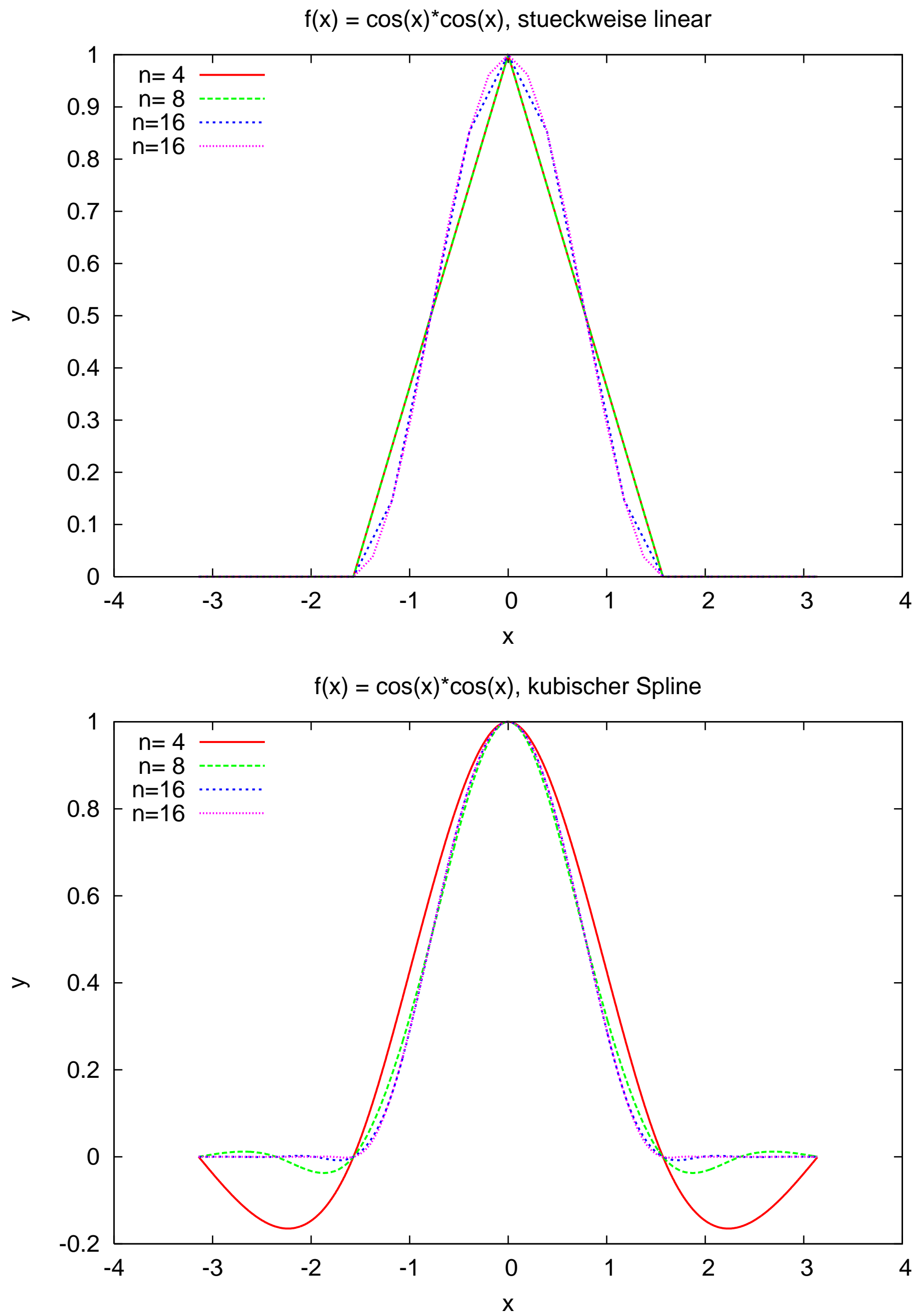



Abbildung 18: Interpolation der Funktion $f_2(x)$ mit Lagrange-Polynomen, stückweise linearen Funktionen und kubischen Splines.

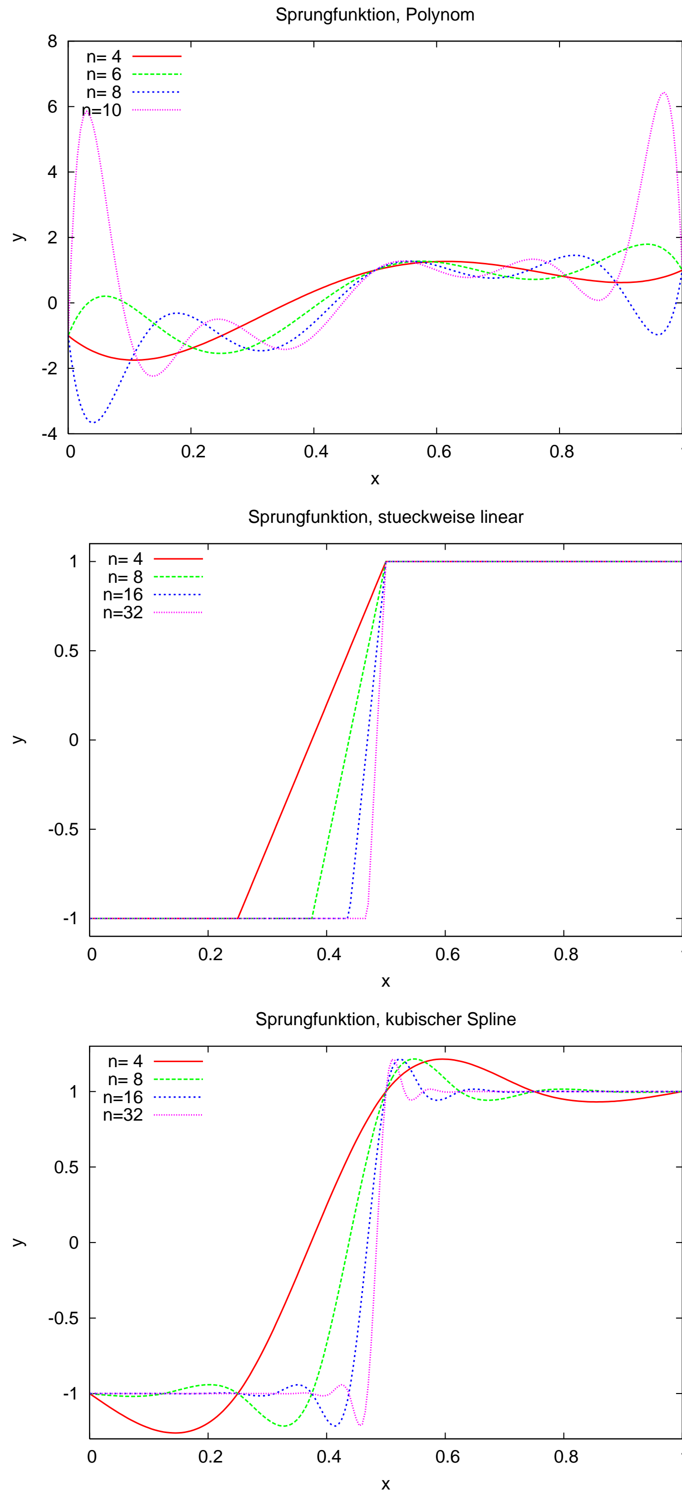



Abbildung 19: Interpolation der Funktion $f_3(x)$ mit Lagrange-Polynomen, stückweise linearen Funktionen und kubischen Splines.

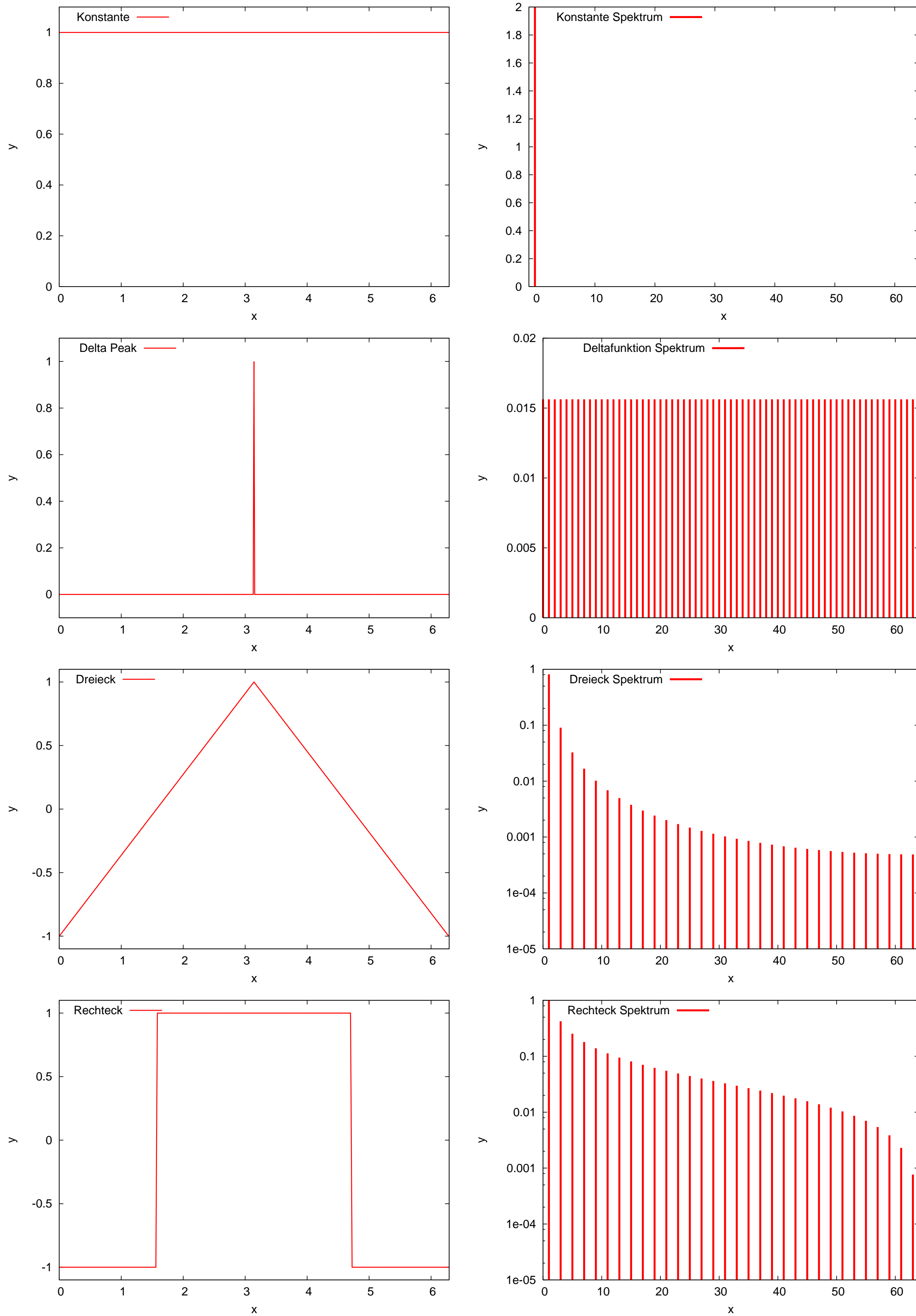
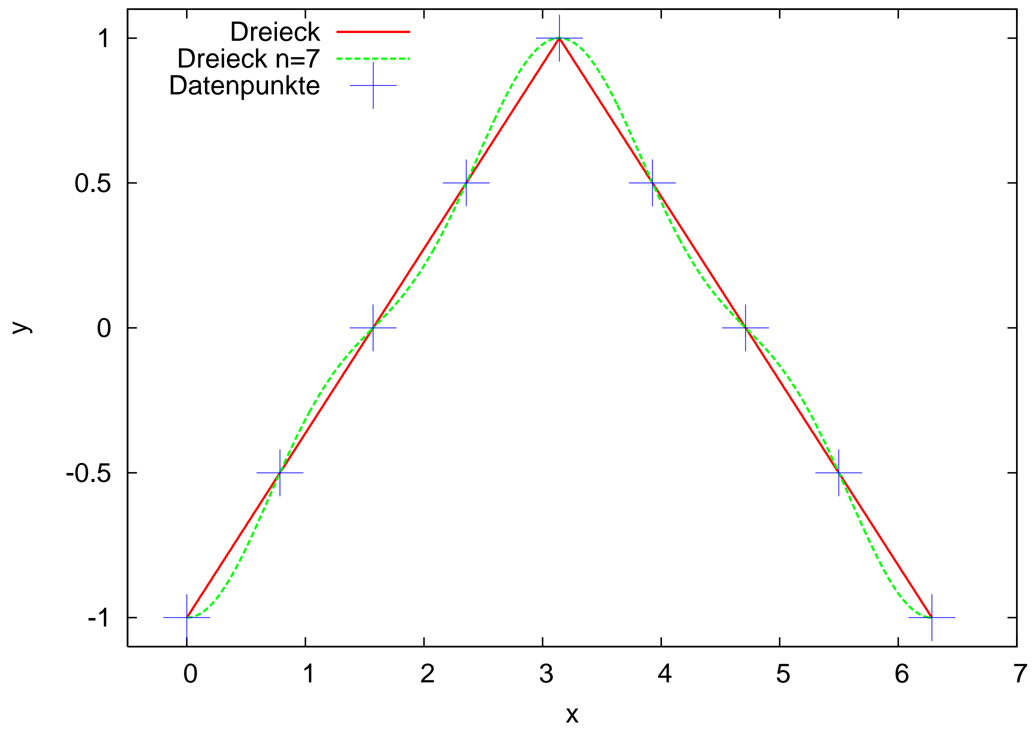



Abbildung 20: Spektren zu verschiedenen Funktionen.



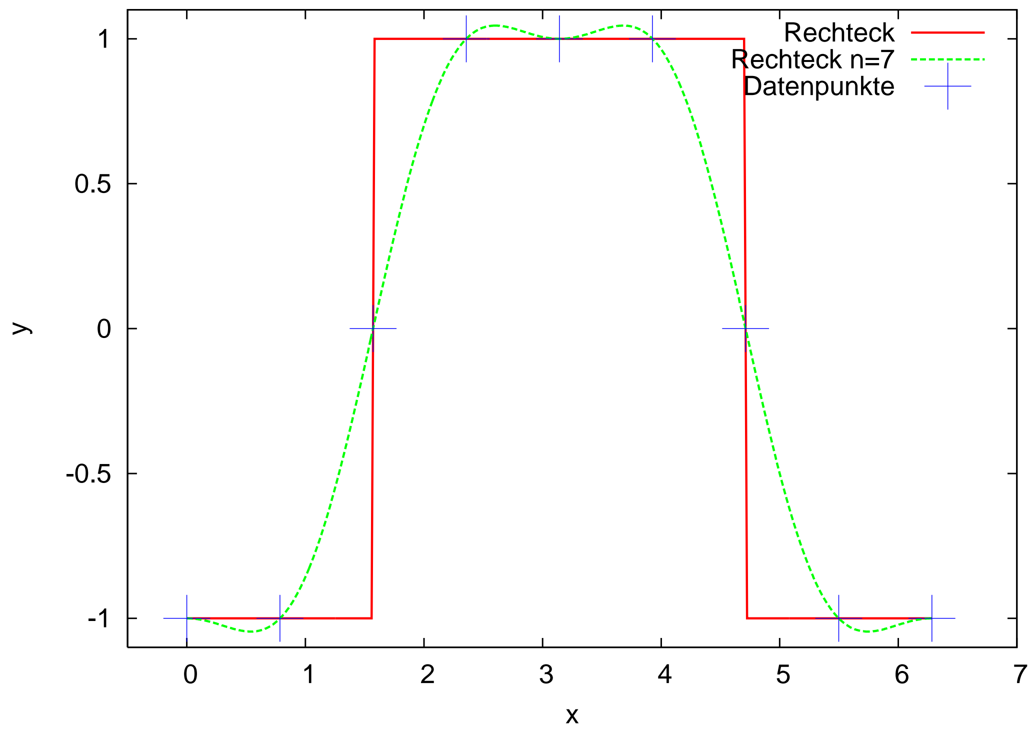
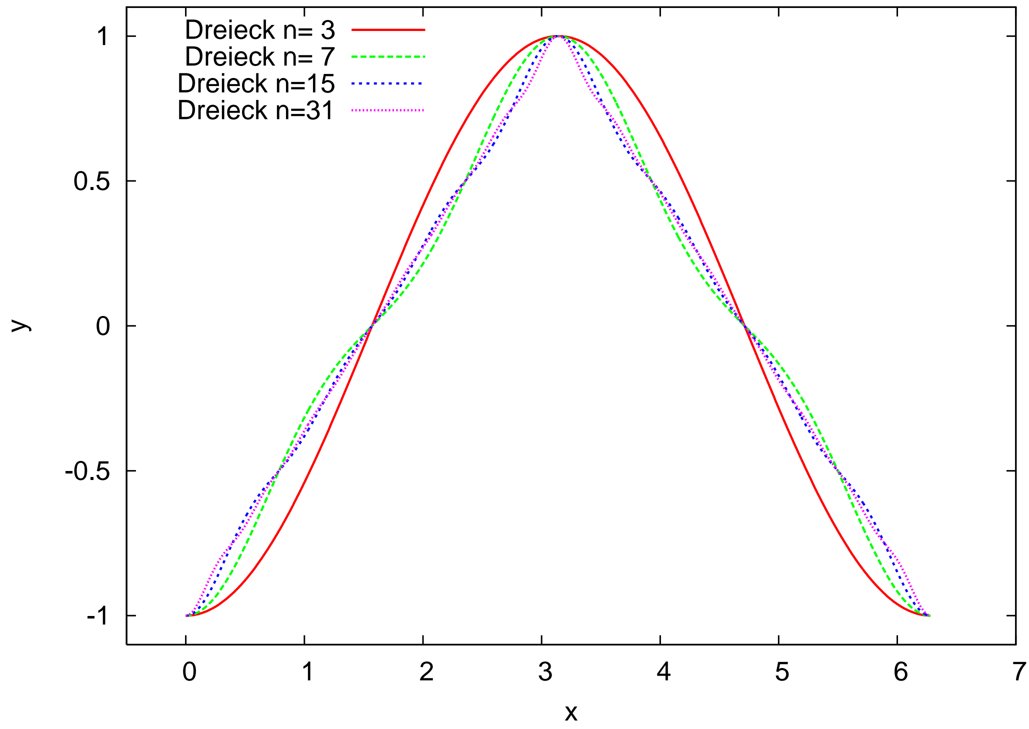




Abbildung 21: Interpolation verschiedener Funktionen.



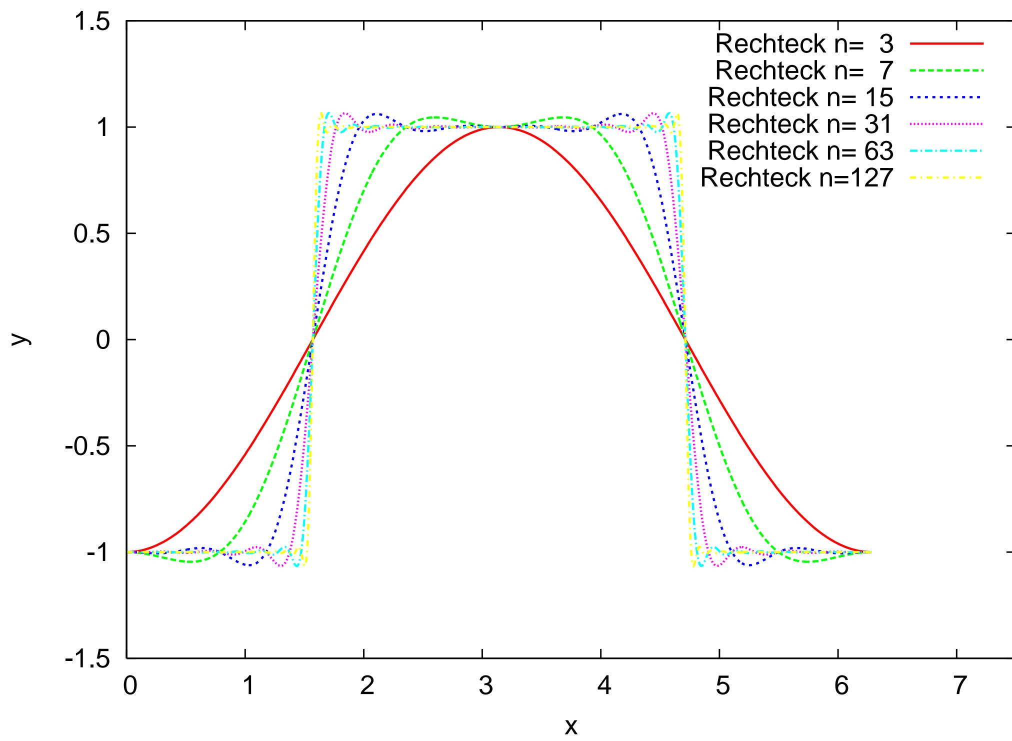




Abbildung 22: Approximation verschiedener Funktionen bei steigendem $n$.

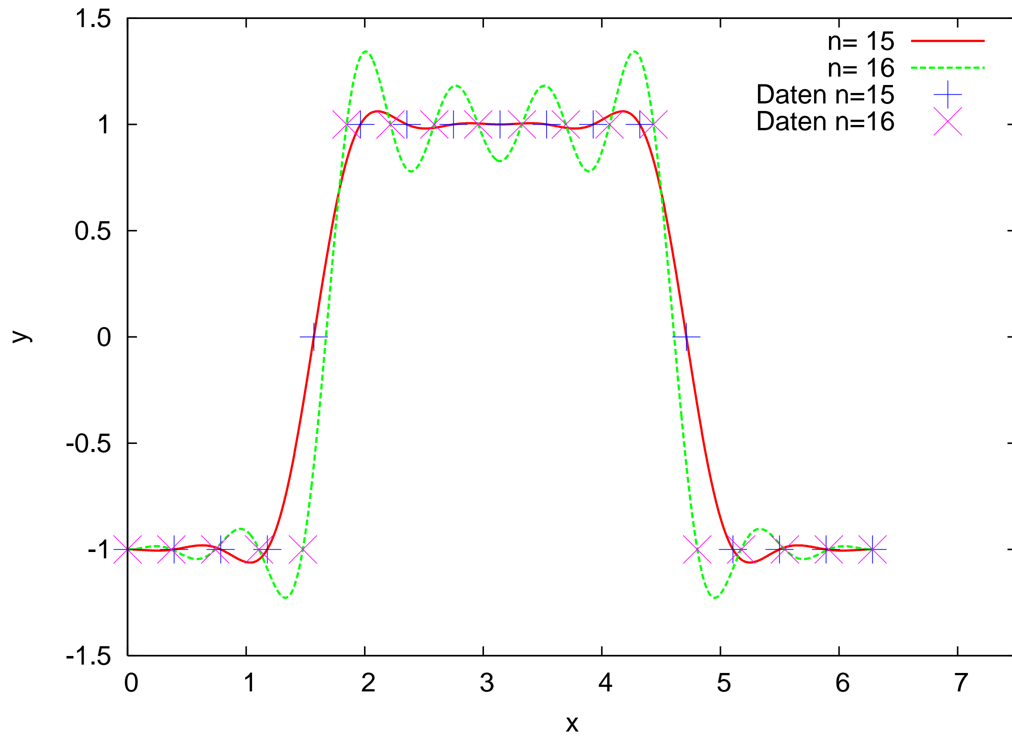



Abbildung 23: Interpolation einer unstetigen Funtion.

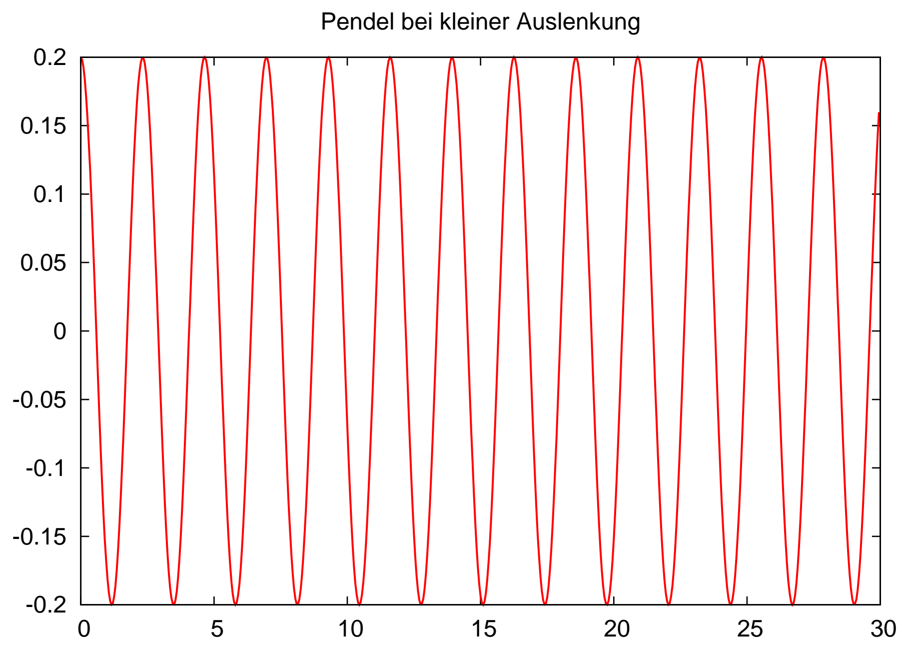



Abbildung 24: Das Pendel in Aktion. Gnuplot-Ausgabe des Programmes `pendel.cc`.