Praktisches Material zur Vorlesung Einführung in die Numerik

Olaf Ippisch
Universität Heidelberg
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Im Neuenheimer Feld 368, D-69120 Heidelberg
email: olaf.ippisch@iwr.uni-heidelberg.de

12. Juli 2011

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 Sommersemester 2011. Es basiert im wesentlichen auf Arbeiten von Peter Bastian (IWR, Universität Heidelberg).

Inhaltsverzeichnis

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!

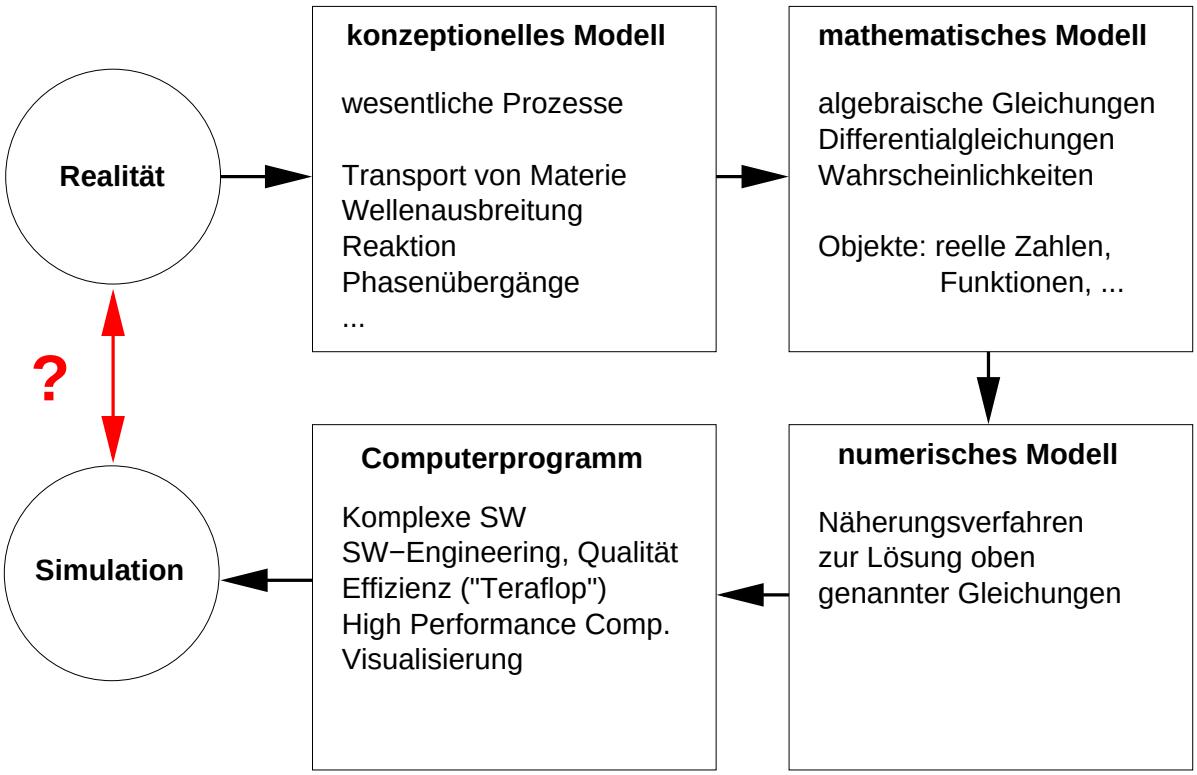


Abbildung 1: Prinzipielles Vorgehen im Wissenschaftlichen Rechnen.

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

Kräfte

- Pendel läuft auf Kreisbahn: Nur *Tangentialkraft* ist relevant.
- Tangentialkraft bei Auslenkung ϕ:

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

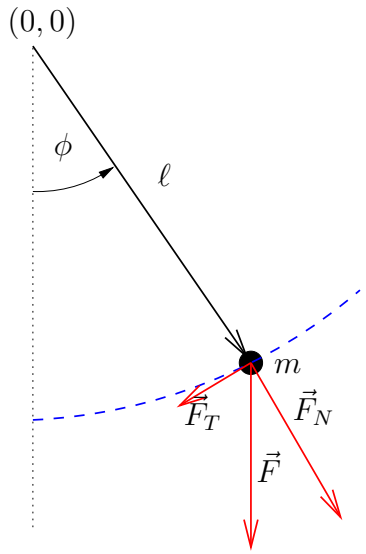


Abbildung 2: Das Fadenpendel.

- Also etwa:

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

- 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:

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

Weg, Geschwindigkeit, 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 $s(t) = \ell\phi(t)$.
- Also für die Geschwindigkeit

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

- und die Beschleunigung

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

Bewegungsgleichung

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

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

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

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

- 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 ϕ gilt
$$\sin(\phi) \approx \phi,$$
z.B. $\sin(0,1) = 0,099833417$.
- Diese *Näherung* reduziert die Gleichung auf
$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{\ell}\phi(t).$$
- 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
$$\phi(t) = \phi_0 \cos\left(\sqrt{\frac{g}{\ell}}t\right) \tag{3}$$

Volles Modell; Verfahren 1

- Löse das volle Modell mit zwei numerischen Verfahren.
- Ersetze Gleichung zweiter Ordnung durch *zwei* Gleichungen *erster* Ordnung:
$$\frac{d\phi(t)}{dt} = u(t), \qquad \frac{d^2\phi(t)}{dt^2} = \frac{du(t)}{dt} = -\frac{g}{\ell}\sin(\phi(t)).$$
- Ersetze Ableitungen durch Differenzenquotienten:
$$\begin{aligned}\frac{\phi(t+\Delta t) - \phi(t)}{\Delta t} &\approx \frac{d\phi(t)}{dt} = u(t),\\ \frac{u(t+\Delta t) - u(t)}{\Delta t} &\approx \frac{du(t)}{dt} = -\frac{g}{\ell}\sin(\phi(t)).\end{aligned}$$
- Mit $\phi^n = \phi(n\Delta t)$, $u^n = u(n\Delta t)$ erhält man Rekursion (*Euler*):
$$\phi^{n+1} = \phi^n + \Delta t\, u^n \qquad \phi^0 = \phi_0 \tag{4}$$
$$u^{n+1} = u^n - \Delta t\,(g/\ell)\,\sin(\phi^n) \qquad u^0 = u_0 \tag{5}$$

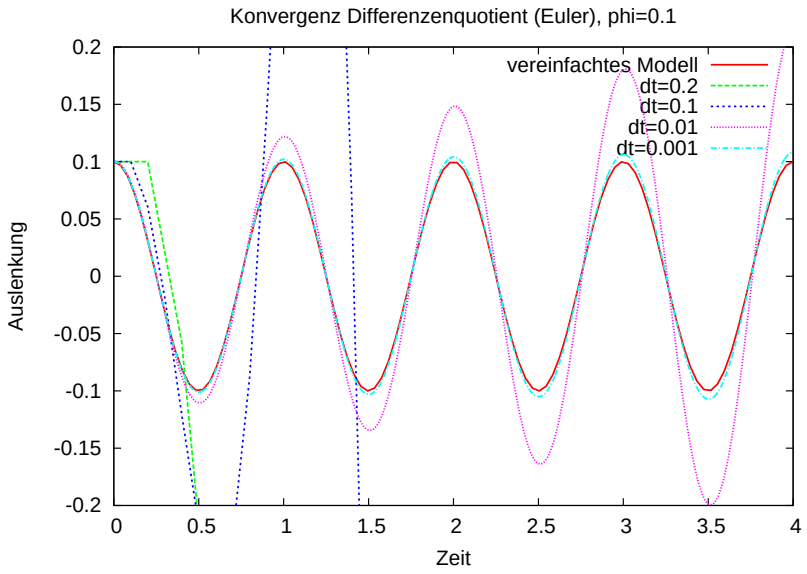


Abbildung 3: Simulation des Fadenpendels (volles Modell) bei $\phi_0 = 0.1 \approx 5.7°$ mit dem Eulerverfahren.

Volles Modell; Verfahren 2

- Nutze Näherungsformel für die zweite Ableitung, sog. *Zentraler Differenzenquotient*:
$$\frac{\phi(t+\Delta t) - 2\phi(t) + \phi(t-\Delta t)}{\Delta t^2} \approx \frac{d^2\phi(t)}{dt^2} = -\frac{g}{\ell}\sin(\phi(t)).$$
- Auflösen nach $\phi(t+\Delta t)$ ergibt Rekursionsformel ($n \geq 2$):
$$\phi^{n+1} = 2\phi^n - \phi^{n-1} - \Delta t^2\,(g/\ell)\,\sin(\phi^n) \tag{6}$$
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 zeigt das Eulerverfahren in Aktion. Für festen Zeitpunkt t und $\Delta t \to 0$ *konvergiert* das Verfahren. Für festes Δt 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 Δt und $t \to \infty$ nicht unbeschränkt zu wachsen.

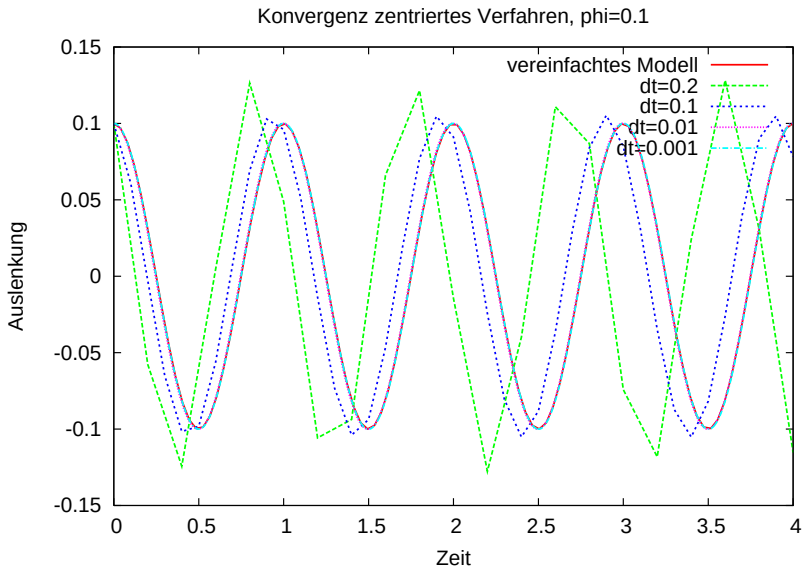


Abbildung 4: Simulation des Fadenpendels (volles Modell) bei $\phi_0 = 0.1 \approx 5.7°$ mit dem zentralen Verfahren.

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.

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

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

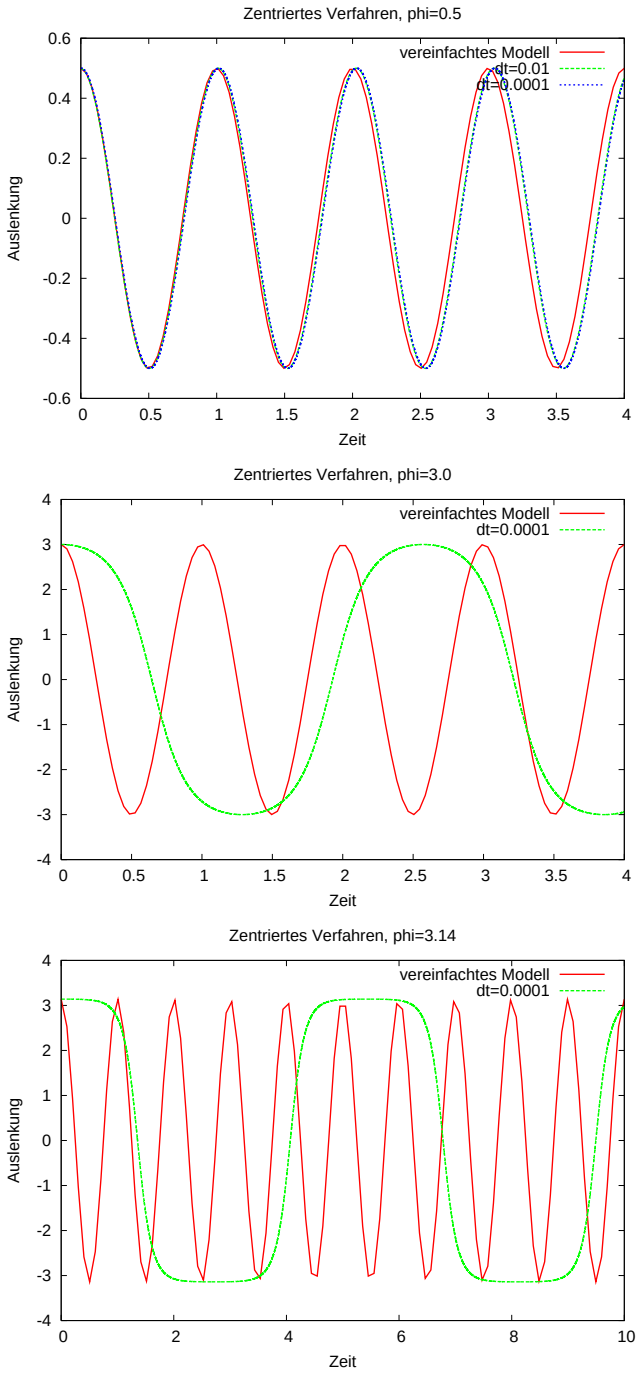


Abbildung 5: Vergleich von vollem und vereinfachtem Modell (jeweils in rot) bei den Winkeln $\phi = 0.5, 3.0, 3.14$ gerechnet mit dem zentralen Verfahren.

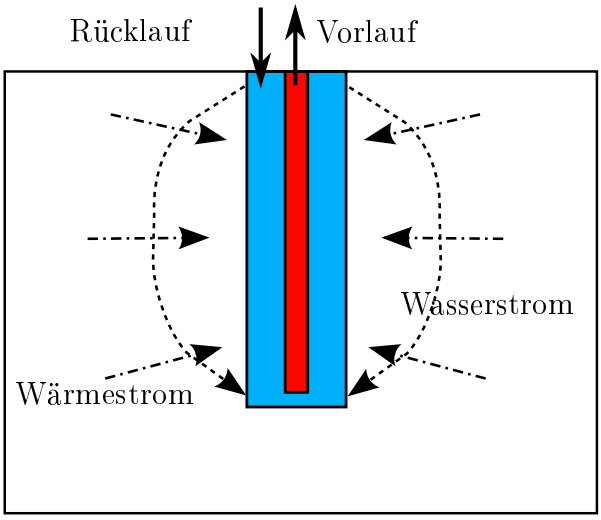


Abbildung 6: Schematischer Aufbau einer Geothermieanlage.

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

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

  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?

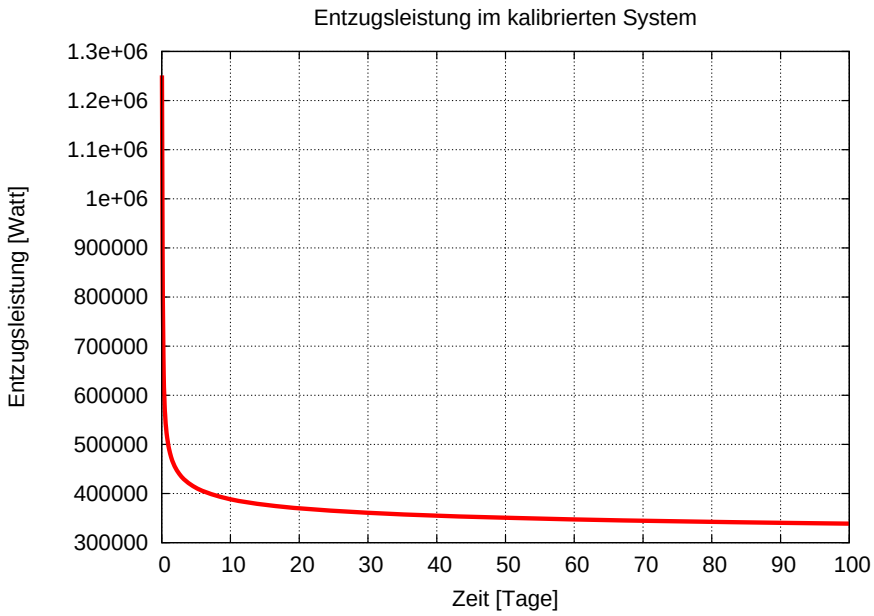


Abbildung 7: Entzugsleistung der Geothermieanlage über die Zeit für bestimmte Systemparameter.

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

Ausblick auf weiterführende Vorlesungen

- Gewöhnliche Differentialgleichungen beschreiben viele zeitabhängige Prozesse (Numerik 1).

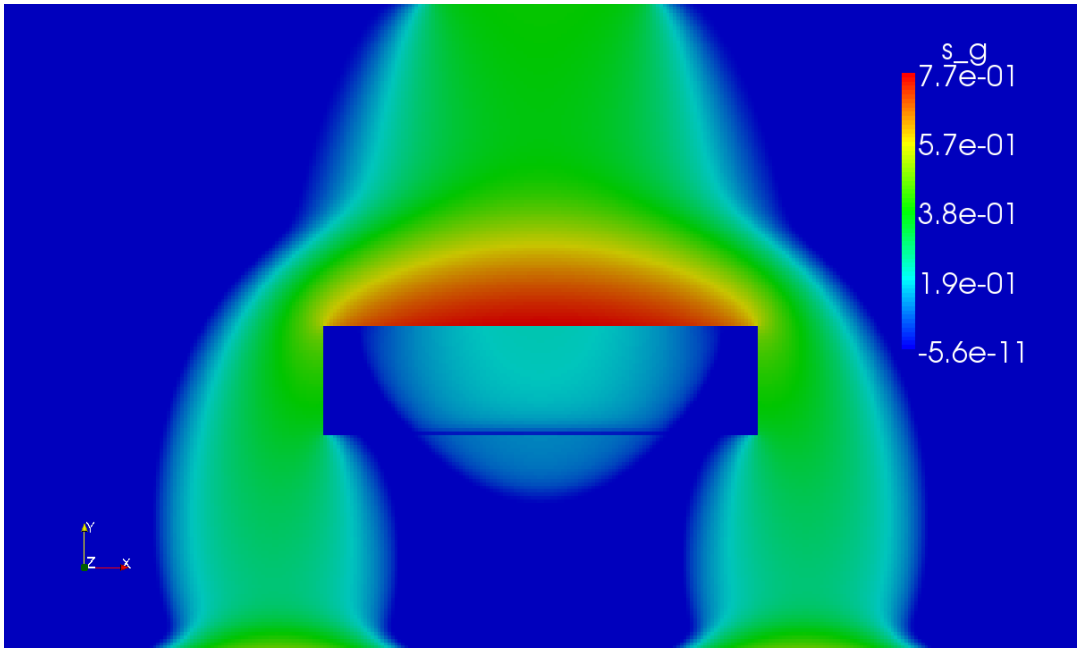


Abbildung 8: Ausbreitung eines nicht Wasser nicht mischbaren Schadstoffes im Boden.

- Praktische Anwendungen führen oft auch auf *partielle Differentialgleichungen*. Gesucht sind dann Funktionen in mehreren Raumdimensionen (Numerik 2).
- Technische Anwendungen erfordern *Optimierung* (Numerik 3).
- Effiziente Programmierung und Visualisierung der Ergebnisse sind sehr wichtig (z.B. Objektorientiertes Programmieren im Wissenschaftlichen Rechnen).

2 Gleitpunktzahlen

Zahlen im Computer

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

`unsigned int, unsigned short, unsigned long`	$\mathbb{N}_0$
`int, short, long`	$\mathbb{Z}$
`float, double`	$\mathbb{R}$
`complex<float>, complex<double>`	$\mathbb{C}$

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

Bei `unsigned int`, `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 Werte von x.

Positives Argument

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

```
1.000000000000000e+00   1  2.000000000000000e+00
5.000000000000000e-01   2  2.500000000000000e+00
1.666666716337204e-01   3  2.666666746139526e+00
4.166666790843010e-02   4  2.708333492279053e+00
8.333333767950535e-03   5  2.716666936874390e+00
1.388888922519982e-03   6  2.718055725097656e+00
1.984127011382952e-04   7  2.718254089355469e+00
2.480158764228690e-05   8  2.718278884887695e+00
2.755731884462875e-06   9  2.718281745910645e+00
2.755731998149713e-07  10  2.718281984329224e+00
0.000000000000000e+00  100 2.718281984329224e+00
                        ex 2.718281828459045e+00
```

  ... also 7 gültige Ziffern.

- Für $x = 5$ ...

```
9.333108209830243e-06  21  1.484131774902344e+02
                       ex  1.484131591025766e+02
```

  ... dito.

Negatives Argument

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

```
 2.755731998149713e-07    10 3.678794205188751e-01
-2.505210972003624e-08    11 3.678793907165527e-01
 2.087675810003020e-09    12 3.678793907165527e-01
                          ex 3.678794411714423e-01
```

  ... 6 gültige Ziffern.

- Für $x = -5$

```
-5.000000000000000e+00    1 -4.000000000000000e+00
 1.250000000000000e+01    2  8.500000000000000e+00
-2.083333396911621e+01    3 -1.233333396911621e+01
 2.604166793823242e+01    4  1.370833396911621e+01
-2.333729527890682e-02   15  1.118892803788185e-03
 7.292904891073704e-03   16  8.411797694861889e-03
 1.221854423194557e-10   28  6.737461313605309e-03
 0.000000000000000e+00  100  6.737461313605309e-03
                         ex  6.737946999085467e-03
```

  nur noch 4 gültige Ziffern!

Noch kleineres Argument

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

```
-2.000000000000000e+01    1 -1.900000000000000e+01
 2.000000000000000e+02    2  1.810000000000000e+02
-1.333333374023438e+03    3 -1.152333374023438e+03
 6.666666992187500e+03    4  5.514333496093750e+03
-2.666666796875000e+04    5 -2.115233398437500e+04
-2.611609750000000e+06   31 -1.011914250000000e+06
 1.632256125000000e+06   32  6.203418750000000e+05
-9.892461250000000e+05   33 -3.689042500000000e+05
 5.819095000000000e+05   34  2.130052500000000e+05
-3.325197187500000e+05   35 -1.195144687500000e+05
 1.847331718750000e+05   36  6.521870312500000e+04
-4.473213550681976e-07   65  7.566840052604675e-01
 1.355519287926654e-07   66  7.566841244697571e-01
-4.046326296247571e-08   67  7.566840648651123e-01
 1.190095932912527e-08   68  7.566840648651123e-01
                         ex  2.061153622438557e-09
```

  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    27 -5.180694836889297e+06
 8.804385629823344e+06    28  3.623690792934047e+06
 1.821561256740375e-24    94  6.147561828914626e-09
-3.834865803663947e-25    95  6.147561828914626e-09
                          ex  2.061153622438557e-09
```

  Immer noch um einen Faktor 3 daneben!

- Erst mit „vierfacher Genauigkeit“ erhält man

```
7.0937168371834023136209731491427e-42  118  2.0611536224385583392700458752947e-09
                                       ex 2.0611536224385578279659403801558e-09
```

  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/mpdist/`) durchgeführt. `qd` erlaubt bis zu vierfache `double` Genauigkeit, `arprec` beliebige Genauigkeit. Die GNU multiprecision library (`http://gmplib.org/`) ist eine Alternative. □

3 Gaußelimination

Algorithmus 4.1 (Gauß-Elimination)
Input: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$

for $(k = 1; k < n; k = k + 1)$ **do**
  Finde $r \in$ {k,...,n} so dass $a_{rk} \neq 0$; (sonst Fehler);
  if $(r \neq k)$ **then** {tausche Zeile k mit Zeile r}
    for $(j = k; j \leq n; j = j + 1)$ **do**
      $t = a_{kj}; a_{kj} = a_{rj}; a_{rj} = t;$
    end for
    $t = b_k; b_k = b_r; b_r = t;$
  end if
  {Update der unteren Zeilen}
  for $(i = k + 1; i \leq n; i = i + 1)$ **do**
    $q = a_{ik}/a_{kk};$
    for $(j = k + 1; j \leq n; j = j + 1)$ **do**
      $a_{ij} = a_{ij} - q \cdot a_{kj};$
    end for
    $b_i = b_i - q \cdot b_k;$
  end for
end for

{Rückwärtseinsetzen}
for $(k = n;\ k \geq 1;\ k = k - 1)$ **do**
  $t = 0$;
  for $(j = k + 1;\ j \leq n;\ j = j + 1)$ **do**
    $t = t + a_{kj} \cdot x_j$;
  end for
  $x_k = (b_k - t)/a_{kk}$
end for

Beispiel 4.2
Hier sind keine Zeilenvertauschungen notwendig. Das Pivotelement ist jeweils durch einen Kasten gekennzeichnet.

$$\left[\begin{array}{cccc|c} \boxed{2} & 4 & 6 & 8 & 40 \\ 16 & 33 & 50 & 67 & 330 \\ 4 & 15 & 31 & 44 & 167 \\ 10 & 29 & 63 & 97 & 350 \end{array}\right] \quad \rightarrow \left[\begin{array}{cccc|c} 2 & 4 & 6 & 8 & 40 \\ 0 & \boxed{1} & 2 & 3 & 10 \\ 0 & 7 & 19 & 28 & 87 \\ 0 & 9 & 33 & 57 & 150 \end{array}\right]$$

$$\rightarrow \left[\begin{array}{cccc|c} 2 & 4 & 6 & 8 & 40 \\ 0 & 1 & 2 & 3 & 10 \\ 0 & 0 & \boxed{5} & 7 & 17 \\ 0 & 0 & 15 & 30 & 60 \end{array}\right] \quad \rightarrow \left[\begin{array}{cccc|c} 2 & 4 & 6 & 8 & 40 \\ 0 & 1 & 2 & 3 & 10 \\ 0 & 0 & 5 & 7 & 17 \\ 0 & 0 & 0 & 9 & 9 \end{array}\right]$$

Schließlich liefert Rückwärtseinsetzen:

$$x_4 = 9/9 = \boxed{1}, \qquad x_3 = (17 - 7 \cdot 1)/5 = \boxed{2},$$
$$x_2 = (10 - 2 \cdot 2 - 3 \cdot 1)/1 = \boxed{3}, \qquad x_1 = (40 - 4 \cdot 3 - 6 \cdot 2 - 8 \cdot 1)/2 = \boxed{4}.$$

□

Algorithmus 4.7.1 zur LR-Zerlegung
Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$ (wird überschrieben)
Output: $L \in \mathbb{R}^{n \times n}$ in $a_{ij}, j < i, l_{ii} = 1$ implizit
$R \in \mathbb{R}^{n \times n}$ in $a_{ij}, j \geq i$
$p: \{0, \ldots, n-1\} \rightarrow \{0, \ldots, n-1\}$
for $(k = 1;\ k < n;\ k = k + 1)$ **do**
  Finde $r \in \{$k,...,n$\}$ so dass $a_{rk} \neq 0$; (sonst Fehler);
  if $(r \neq k)$ **then** {tausche Zeile k mit Zeile r}
    for $(j = 1;\ j \leq n;\ j = j + 1)$ **do**
      $t = a_{kj}; a_{kj} = a_{rj}; a_{rj} = t$;
    end for
  end if
  $p_k = r$; {merke Permutation}
  {Update der unteren Zeilen}

for $(i = k+1;\ i \leq n;\ i = i+1)$ **do**
$a_{ik} = a_{ik}/a_{kk}$;
for $(j = k+1;\ j \leq n;\ j = j+1)$ **do**
$a_{ij} = a_{ij} - a_{ik} \cdot a_{kj}$;
end for
end for
end for

{Permutation von b}
for $(k = 1;\ k < n;\ k = k+1)$ **do**
if $(p_k \neq k)$ **then**
$t = b_k; b_k = b_{p_k}; b_{p_k} = t$;
end if
end for
{Vorwärtseinsetzen}
for $(k = 1;\ k \leq n;\ k = k+1)$ **do**
$t = 0$;
for $(j = 1;\ j < k;\ j = j+1)$ **do**
$t = t + a_{kj} \cdot x_j$;
end for
$x_k = b_k - t$;
end for
{Rückwärtseinsetzen}
for $(k = n;\ k \geq 1;\ k = k-1)$ **do**
$t = 0$;
for $(j = k+1;\ j \leq n;\ j = j+1)$ **do**
$t = t + a_{kj} \cdot x_j$;
end for
$x_k = (x_k - t)/a_{kk}$;
end for

Algorithmus 4.7.2 zur LR-Zerlegung (Andere Permutation)
Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$ (wird überschrieben)
Output: $L \in \mathbb{R}^{n \times n}$ in $a_{ij}, j < i, l_{ii} = 1$ implizit
$R \in \mathbb{R}^{n \times n}$ in $a_{ij}, j \geq i$
$p: \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$
for $(k = 1;\ k \leq n;\ k = k+1)$ **do**
$p_k = k$;
end for
for $(k = 1;\ k < n;\ k = k+1)$ **do**
Finde $r \in$ {k,...,n} so dass $a_{rk} \neq 0$; (sonst Fehler);
if $(r \neq k)$ **then** {tausche Zeile k mit Zeile r}

```
      for (j = 1; j ≤ n; j = j + 1) do
        t = a_kj;a_kj = a_rj;a_rj = t;
      end for
      t = p_k;p_k = p_r;p_r = t;
    end if
    {Update der unteren Zeilen}
    for (i = k + 1; i ≤ n; i = i + 1) do
      a_ik = a_ik/a_kk;
      for (j = k + 1; j ≤ n; j = j + 1) do
        a_ij = a_ij − a_ik · a_kj;
      end for
    end for
  end for
  {Permutation von b}
  for (k = 1; k ≤ n; k = k + 1) do
    x_k = b_{p_k};
  end for
  {Vorwärtseinsetzen}
  for (k = 1; k ≤ n; k = k + 1) do
    t = 0;
    for (j = 1; j < k; j = j + 1) do
      t = t + a_kj · x_j;
    end for
    x_k = x_k − t;
  end for
  {Rückwärtseinsetzen}
  for (k = n; k ≥ 1; k = k − 1) do
    t = 0;
    for (j = k + 1; j ≤ n; j = j + 1) do
      t = t + a_kj · x_j;
    end for
    x_k = (x_k − t)/a_kk;
  end for
```

Algorithmus 4. zu LS with QR-Zerlegung

```
 Input:   x ∈ ℝ^n, j ∈ ℕ^+
 Output:  v ∈ ℝ^n
  HouseholderVector(x,j)
  begin
  n = length(x);
  v = 0;
  μ = 0;
```

for $(k = j;\ k \leq n;\ k = k + 1)$ **do**
$\mu = \mu + x_k * x_k$;
$v_k = x_k$;
end for
if $(\mu \neq 0)$ **then**
$\beta = x_1 + \mathsf{sign}(x_1)\mu$;
for $(k = j + 1;\ k \leq n;\ k = k + 1)$ **do**
$v_k = v_k/\beta$;
end for
end if
$v_1 = 1$
return v;
end

Algorithmus 4. zu LS with QR-Zerlegung
Input: $A \in \mathbb{R}^{m \times n}$, $v \in \mathbb{R}^m$
Output: $A \in \mathbb{R}^{m \times n}$ (wird überschrieben)
HouseholderMultiplikation(A,v)
begin
$\beta = -2/(v^T v)$;
$w = \beta A^T v$;
$A = A + vw^T$;
end

Algorithmus 4. zu LS with QR-Zerlegung
Input: $A \in \mathbb{R}^{m \times n}$
Output: n Vektoren $v^{(k)} \in \mathbb{R}^{m \times n}$ in $a_{ij}, j < i, v_1^{(k)} = 1$ implizit
$R \in \mathbb{R}^{m \times n}$ in $a_{ij}, j \geq i$
HouseholderQR(A)
begin
for $j = 1; j \leq n; j = j + 1$ **do**
v_k
$w = \beta A^T v$;
$A = A + vw^T$;
end for
end

4 Interpolation und Approximation

4.1 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!).

4.2 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.

Beispiel 5.8 zur numerischen Differentiation
Wir wollen die zweite Ableitung von $f(x) = \sinh(x)$ für $x = 0.6$ mit dem zweiten Differenzenquotient ermitteln:

$$\frac{d^2}{dx^2}\sinh(x) \approx \frac{\sinh(x+h) - 2\sinh(x) + \sinh(x-h)}{h^2}$$

Abbildung 9: Kurvenkompression in der Computergraphik: Die Szene.

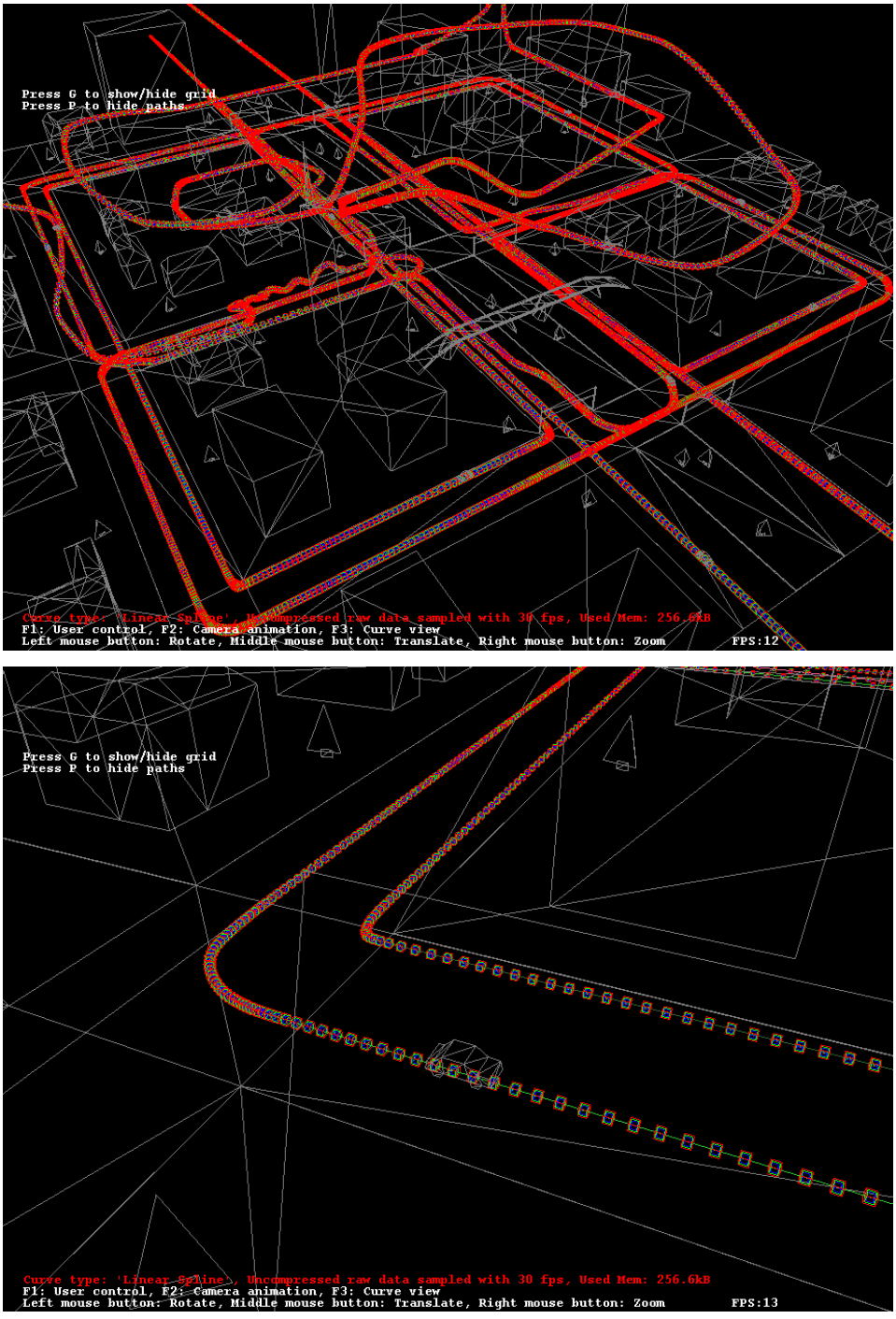


Abbildung 10: Kurvenkompression in der Computergraphik: Stützpunkte der unkomprimierten Kurve.

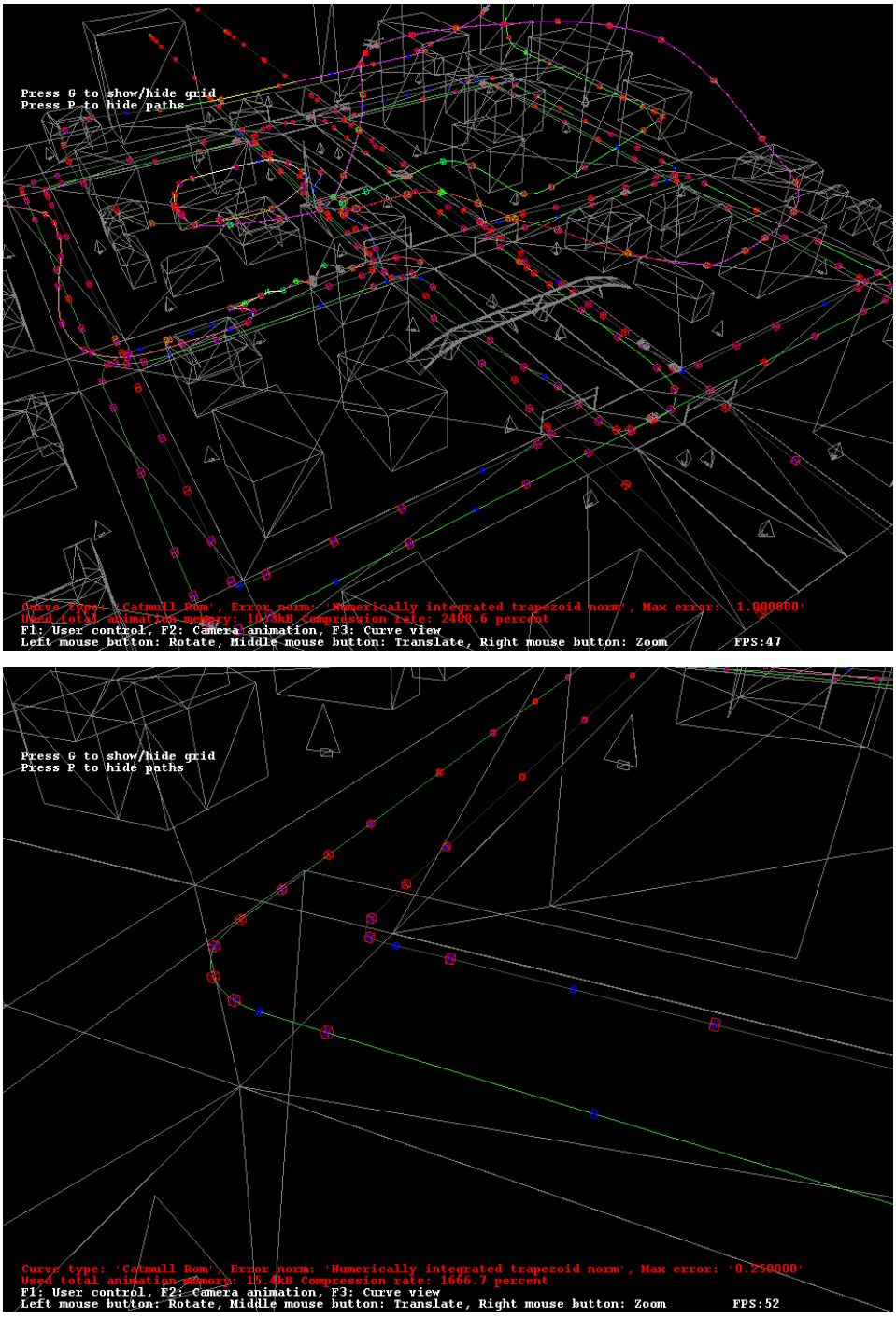


Abbildung 11: Kurvenkompression in der Computergraphik: Stützpunkte der komprimierten Kurve.

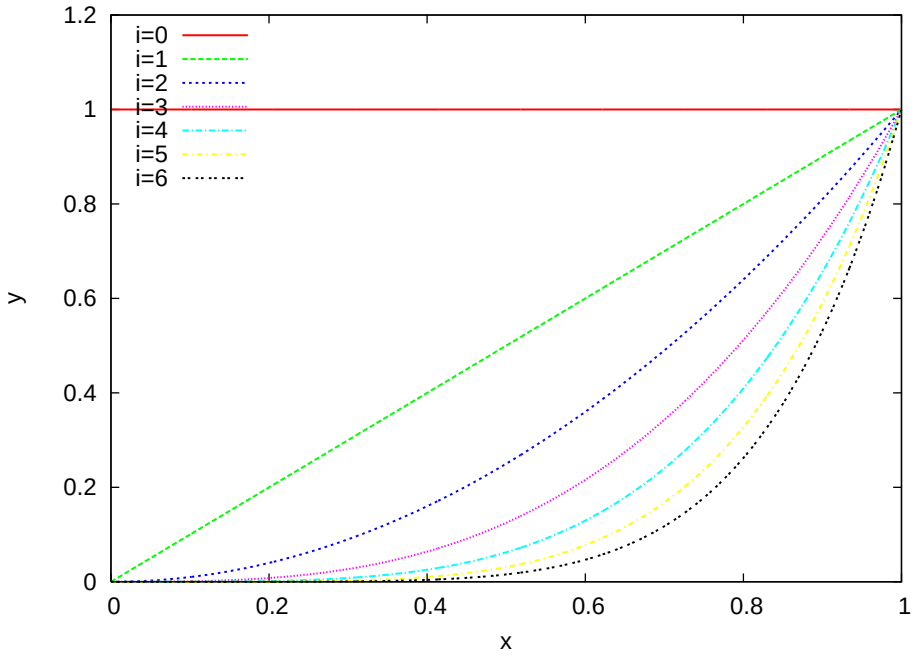


Abbildung 12: Die Monome bis zum Grad 6.

zur Erinnerung:

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x}),$$
$$\frac{d}{dx}\sinh = \cosh = \frac{1}{2}(e^x + e^{-x}),$$
$$\frac{d^2}{dx^2}\sinh(x) = \sinh(x).$$

Mit `double` Genauigkeit erhält man den Wert

$$\sinh(0.6) = 6,366535821482 \cdot 10^{-1}.$$

Dagegen liefert die numerische Differentiation die folgende Tabelle

h	Differenzenquotient		
$1 \cdot 10^{-1}$	6.371	$\cdot 10^{-1}$	
$1 \cdot 10^{-2}$	6.3665888	$\cdot 10^{-1}$	
$1 \cdot 10^{-3}$	6.366536352	$\cdot 10^{-1}$	
$1 \cdot 10^{-4}$	6.3665358540	$\cdot 10^{-1}$	
$1 \cdot 10^{-5}$	6.3665017	$\cdot 10^{-1}$	Auslöschung
$1 \cdot 10^{-6}$	6.3671	$\cdot 10^{-1}$	
$\vdots$			
$1 \cdot 10^{-10}$	1.1102	$\cdot 10^{4}$	!

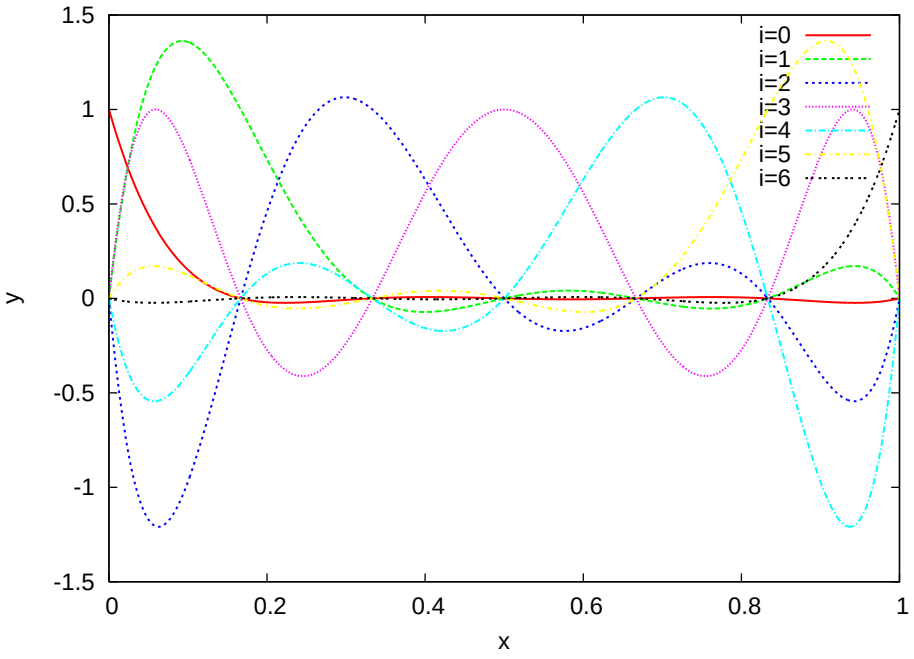


Abbildung 13: Die Lagrange-Polynome $L_i^{(6)}(x)$ vom Grad 6.

Numerische Differentation ist sehr anfällig gegenüber Rundungsfehlern. Mögliche Abhilfe bietet die „Extrapolation“. □

4.3 Spline Interpolation

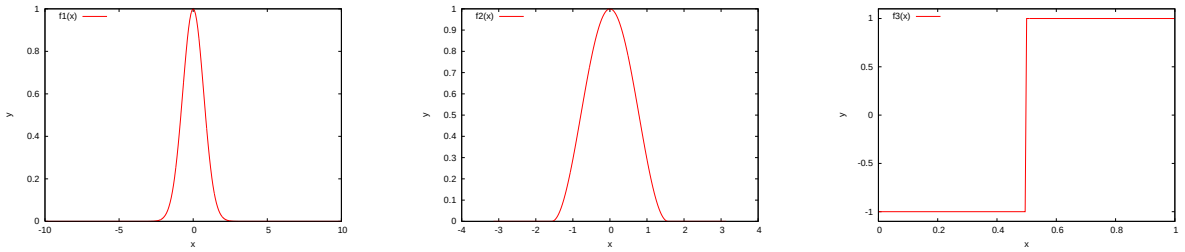
Wir betrachten die Interpolation der folgenden drei Funktionen

$$f_1(x) = \exp(-x^2) \qquad \text{in } [-10, 10], \tag{8}$$

$$f_2(x) = \begin{cases} \cos^2(x) & |x| < \pi/2 \\ 0 & |x| \geq \pi/2 \end{cases} \qquad \text{in } [-\pi, \pi], \tag{9}$$

$$f_3(x) = \begin{cases} -1 & x < 1/2 \\ +1 & x \geq 1/2 \end{cases} \qquad \text{in } [0, 1], \tag{10}$$

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



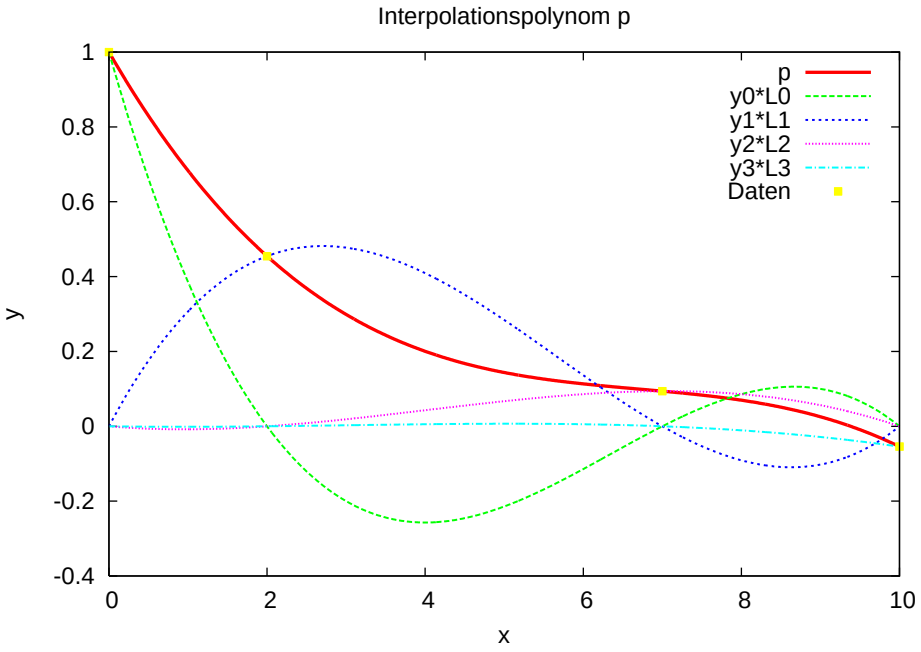


Abbildung 14: Interpolationspolynom zur Wertetabelle aus Beispiel .

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.

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

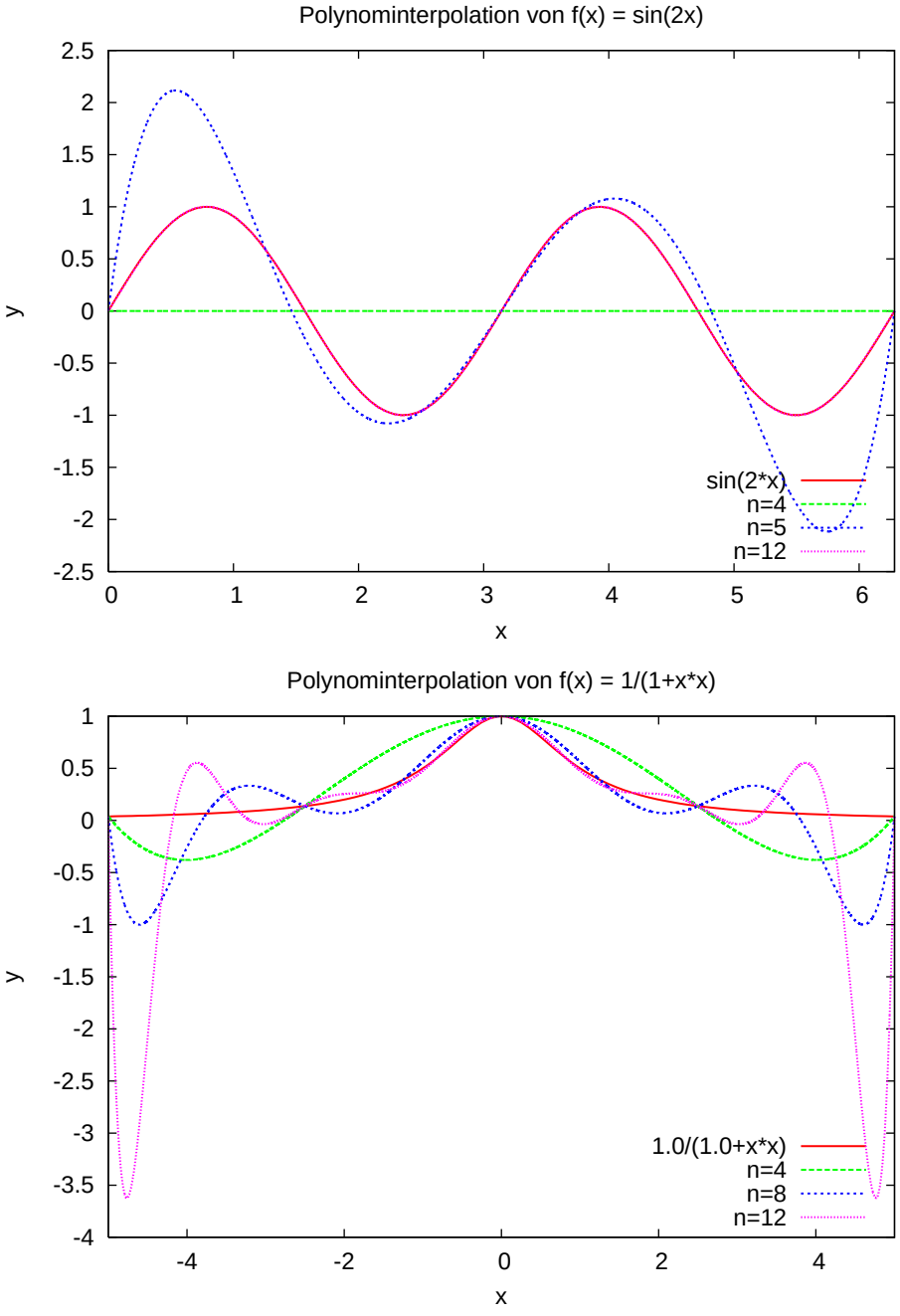


Abbildung 15: Interpolation der Funktionen $\sin(2x)$ (oben) und $\frac{1}{1+x^2}$ (unten) mit äquidistanten Stützstellen und verschiedenen Polynomgraden.

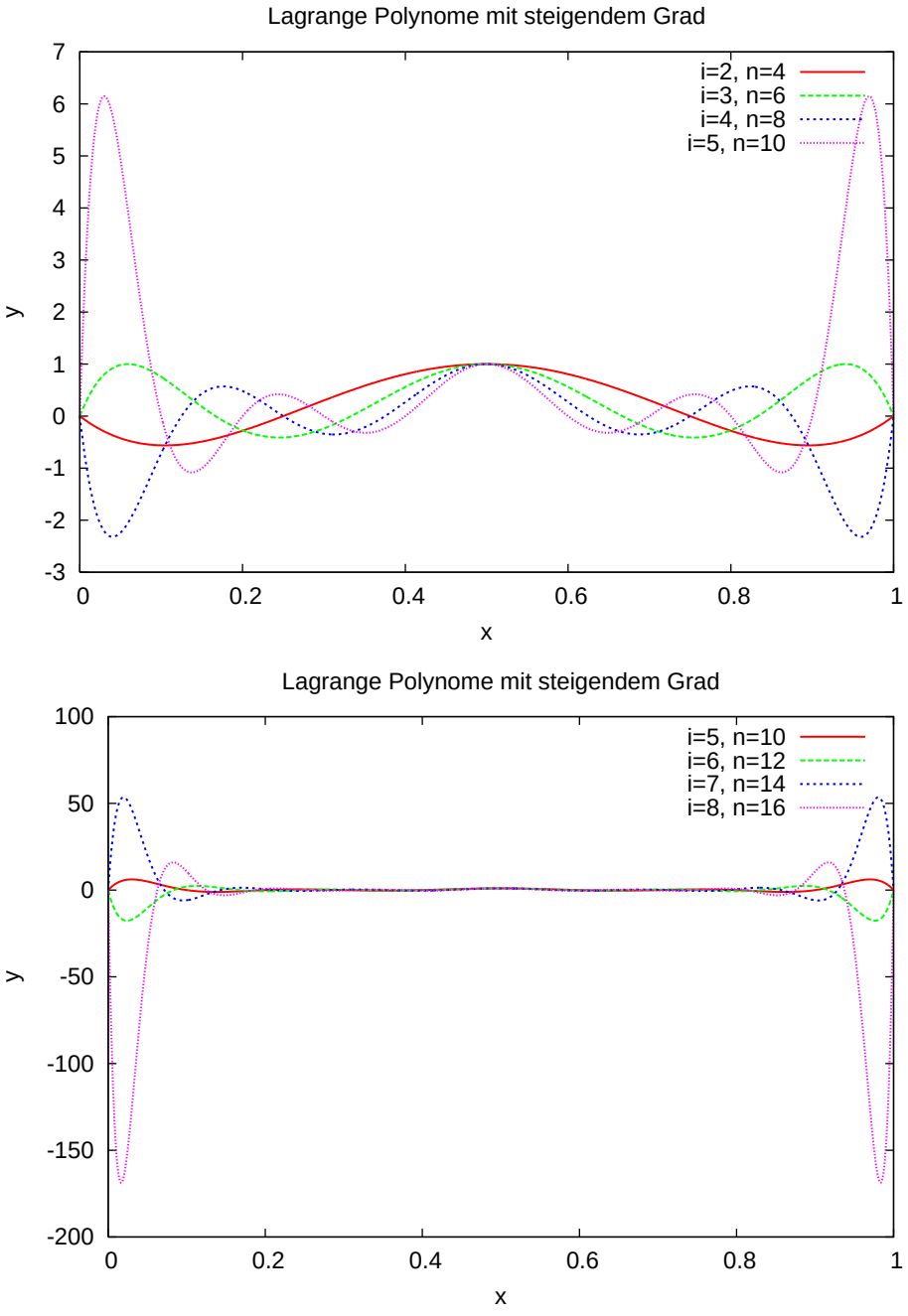


Abbildung 16: Die Lagrange-Polynome $L_{n/2}^{(n)}$ für $n = 4, 6, 8, 10, 12, 14, 16$.

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“ 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:

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. □

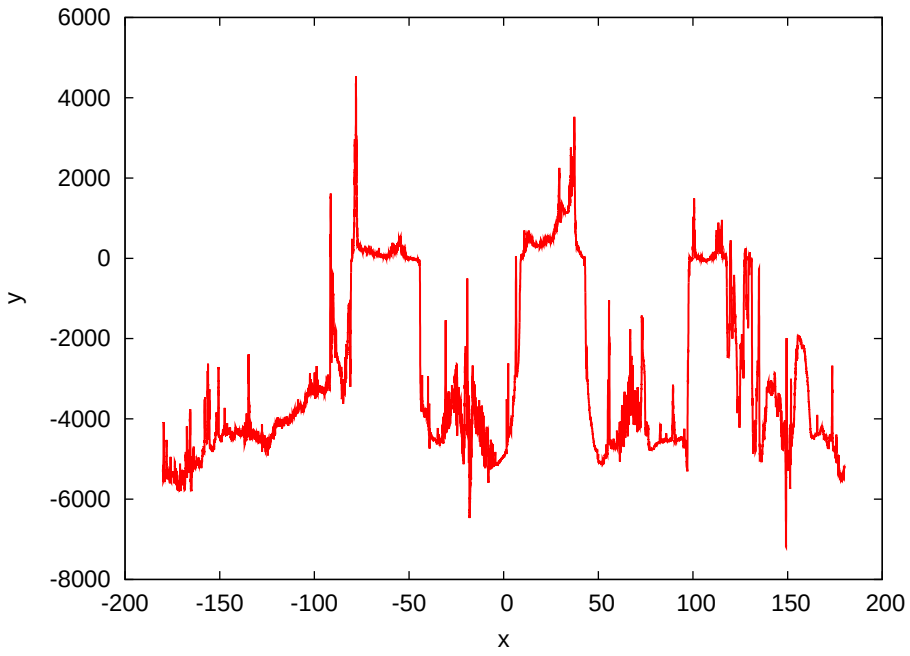
4.4 Praktisches zur Diskreten Fourier Analyse

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).



4.5 Gauß-Approximation

Beispiel 5.23: Fourierreihe

Für $N = 2m + 1, \quad m \in \mathbb{N}$ ist

$$\Psi_F = \left\{ \tfrac{1}{\sqrt{2\pi}}, \ \frac{1}{\sqrt{\pi}}\cos(x), \ldots, \frac{1}{\sqrt{\pi}}\cos(mx), \frac{1}{\sqrt{\pi}}\sin(x), \ldots, \frac{1}{\sqrt{\pi}}\sin(mx) \right\}$$

eine Orthonormalbasis von Funktionen auf dem Intervall $[-\pi, \pi]$.

Damit gilt dann:

$$g(x) = \frac{a_0}{2} + \sum_{k=1}^{m} \{a_k \cos(kx) + b_k \sin(kx)\}$$

und

$$\begin{aligned} a_k &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(kx) dx, \qquad k = 0, \ldots, m \\ b_k &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(kx) dx, \qquad k = 1, \ldots, m \end{aligned}$$

Für unendlich viele Glieder ($m = \infty$) nennt man die Reihe Fourier-Reihe. Diese konvergiert gegen ein Element aus $L^2(-\pi, \pi)$.

Beispiel 5.24: Finite-Elemente Diskretisierung

Die numerische Lösung der partiellen Differentialgleichung

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f \qquad \text{in } \Omega$$

(Poisongleichung) mit Hilfe der Methode der Finiten-Elemente führt auf die Aufgabe:

Finde $u \in S$, so dass

$$\int_{\Omega} \nabla u \cdot \nabla \varphi dx = \int_{\Omega} f \varphi dx \qquad \forall \varphi \in S$$

5 Quadratur

5.1 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^α 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

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

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 α heißt *experimental order of convergence* (EOC).

Im letzen Beispiel oben erhalten wir $\alpha = 3/2$.

5.2 Fehlerkontrolle

Wir haben in Satz 6.4 gezeigt, wie der Fehler mit mehr Stützstellen abnimmt. Dies nennt man eine *a-priori Fehlerschranke*.

So erhalten wir etwa für die summierte Trapezregel:

$$|I(f) - I_h^{(1)}(f)| = | - \frac{b-a}{12} f''(\xi) h^2| \leq \underbrace{\frac{b-a}{12} \max_{\xi \in [a,b]} |f''(\xi)|}_{=:C} h^2.$$

C ist allerdings im allgemeinen schwer zu bestimmen.

In der Praxis würde man aber gerne wissen, bei wievielen Stützstellen (bei welchem h) der Fehler kleiner als eine vorgegebene *Toleranz* ist.

Dazu wollen wir eine Methode zur *a-posteriori Fehlerschätzung* vorstellen.

Idee: Die Simpson-Summe konvergiert schneller als die Trapezsumme (f genügend glatt), hat also für genügend kleines h einen kleineren Fehler.

Wir wollen den Fehler in der Trapezsumme zum Gitter $h/2$ abschätzen. Dazu „schieben“ wir die Auswertung der Simpsonsumme dazwischen:

$$|I(f) - I_{\frac{h}{2}}^{(1)}(f)| = |I(f) - I_h^{(2)}(f) + I_h^{(2)}(f) - I_{\frac{h}{2}}^{(1)}(f)|.$$

Nun nutze die Dreiecksungleichung:

$$|I(f) - I_{\frac{h}{2}}^{(1)}(f)| \leq |I(f) - I_h^{(2)}(f)| + |I_h^{(2)}(f) - I_{\frac{h}{2}}^{(1)}(f)|.$$

Nun *nimmt man an*, dass die Simpsonsumme genauer ist als die Trapezsumme: $|I(f) - I_h^{(2)}(f)| \leq \omega |I(f) - I_{\frac{h}{2}}^{(1)}(f)|$ mit $0 < \omega < 1$:

$$|I(f) - I_{\frac{h}{2}}^{(1)}(f)| \leq \omega |I(f) - I_{\frac{h}{2}}^{(1)}(f)| + |I_h^{(2)}(f) - I_{\frac{h}{2}}^{(1)}(f)|.$$

Auflösen nach dem Fehler in der Trapezsumme liefert:

$$|I(f) - I_{\frac{h}{2}}^{(1)}(f)| \leq \frac{1}{1-\omega} |I_h^{(2)}(f) - I_{\frac{h}{2}}^{(1)}(f)|.$$

Algorithmische Realisierung der Fehlerkontrolle

Besonders ökonomisch lässt sich die Fehlerkontrolle realisieren, wenn man folgende Beziehungen zwischen Simpson-Rgel, Mittelpunktregel und Trapezregel benutzt (deshalb haben wir oben die Trapezregel zu $h/2$ und die Simpsonsumme zu h verwendet):

Die summierte Simpson-Regel lässt sich aus summierter Trapez- und Mittelpunktregel zusammensetzen:

$$I_h^{(2)}(f) = \frac{1}{3} I_h^{(1)}(f) + \frac{2}{3} I_h^{(0)}(f).$$

Die summierte Trapezregel zu dem nächst feineren Gitter erhält man aus summierter Trapez- und Mittelpunktregel des gröberen Gitters:

$$I^{(1)}_{\frac{h}{2}} = \frac{1}{2} I^{(1)}_h(f) + \frac{1}{2} I^{(0)}_h(f)$$

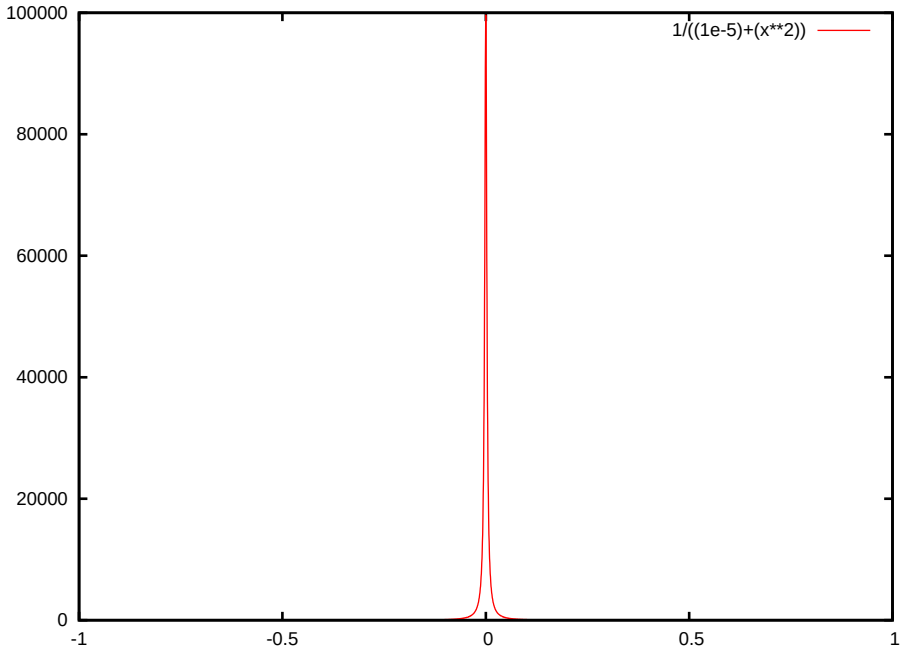
$h = b - a$; $N = 1$; $I1 = h(f(a) + f(b))/2$;
while $(h > \varepsilon)$ **do**
  $I0 = 0$;
  for $(i = 0, i < N, i = i + 1)$ **do**
    $I0 = I0 + hf(a + (i + 0.5)h)$; {Mittelpunktsumme}
  end for
  $I2 = \frac{1}{3} I1 + \frac{2}{3} I0$; {Simpson-Summe zu h}
  $I1 = \frac{1}{2} I1 + \frac{1}{2} I0$; {Trapez-Summe zu $h/2$}
  $h = \frac{1}{2} h$; $N = 2N$;
  if $(\frac{1}{1-\omega}(I2 - I1) \leq TOL)$ **then**
    return $I1$; {Liefere Trapez-Summe zu h}
  end if
end while

Die Fehlerkontrolle lässt sich so ohne zusätzlichen Aufwand erledigen.

5.3 Gauß- und adaptive Quadratur

Adaptive Quadratur

Quadratur mit konstanter Schrittweite ist bei manchen Integranden ineffizient. Betrachte z. B. $f(x) = \frac{1}{10^{-5} + x^2}$.



In so einem Fall möchte man die Schrittweite „adaptiv“, d. h. angepasst an den speziellen Integranden wählen.

Dazu bedient man sich eines lokalen „Fehlerschätzers“ (oder Indikators), der angibt, an welcher Stelle die Schrittweite weiter verkleinert werden muss.

Es bietet sich an, dies noch mit einer Fehlerkontrolle zu kombinieren. Grob ergibt sich das folgende Vorgehen:

(1) Wähle eine Unterteilung $G_0 = \{x_i^{(0)} \mid i = 0, \ldots, N_0\}$. Berechne das Integral I_0 bezüglich der Unterteilung G_0. Setze $k = 0$.

(2) Berechne eine Schätzung für den Fehler E_k in I_k. Falls $E_k < TOL$ sind wir fertig.

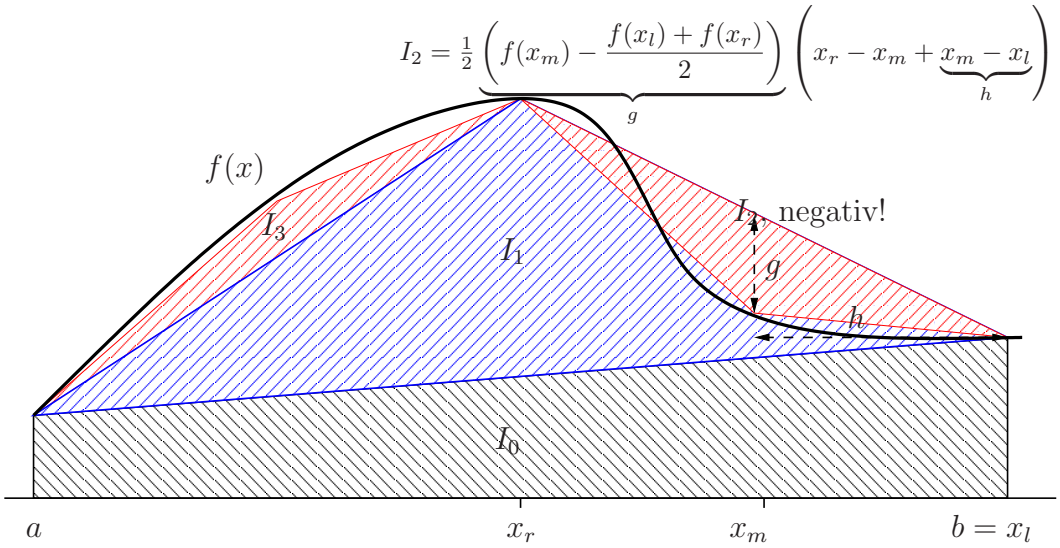
(3) Verfeinere die Unterteilung G_k zu G_{k+1} durch Hinzufügen von Punkten angepasst an den Integranden und setze $k = k + 1$.

(4) Berechne I_k zu G_k und gehe nach (2).

Prinzip von Archimedes

Wir betrachten das Prinzip von Archimedes[2]:

[2] Archimedes von Syrakus, 287 v. Chr.-212 v. Chr., griechischer Mathematiker, Physiker und Ingenieur.



1. Berechne rekursiv $I(f) = I_0 + I_1 + I_2 + I_3 + \ldots$
2. Breche rekursiven Ast ab, falls $|I_j|$ klein genug.

Beispiel 6.8 zur numerischen 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.

5.4 Mehrdimensionale Quadratur

Für Rechtecke ($d = 2$), Quader ($d = 3$), ...lassen sich die Formeln für die eindimensionale Integration leicht erweitern:

$$\begin{aligned}\int_c^d \int_a^b f(x,y)\,dxdy &\approx \int_c^d \sum_{i=0}^{n} w_i f(x_i,y)\,dy \\ &= \sum_{i=0}^{n} w_i \int_c^d f(x_i,y)\,dx \\ &\approx \sum_{i=0}^{n} w_i \left(\sum_{j=0}^{n} w_j f(x_i,y_j) \right) \\ &= \sum_{i=0}^{n}\sum_{j=0}^{n} \underbrace{w_i w_j}_{w_{ij}} f(x_i,y_j)\end{aligned}$$

Koordinatentransformation

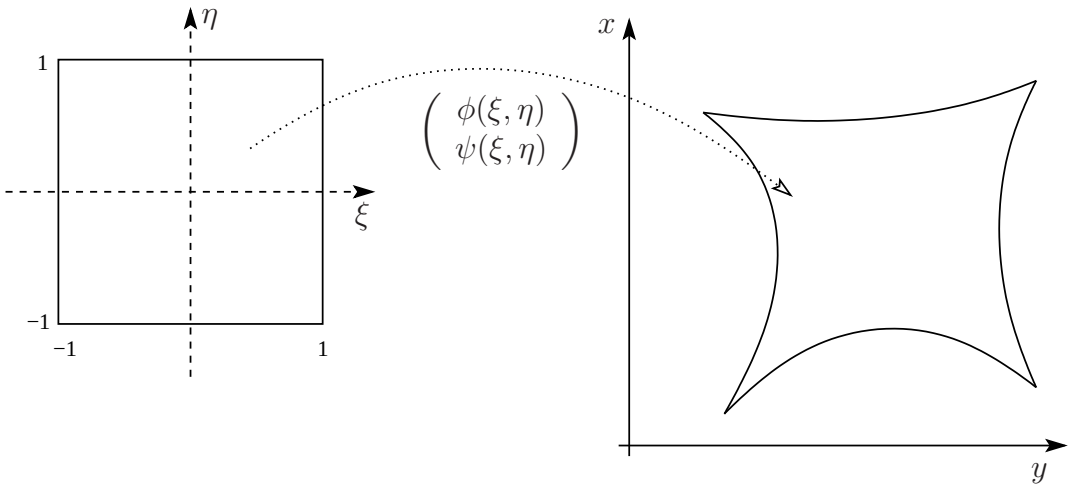
Allerdings sind nicht alle Gebiete Rechtecke (anders als in 1D!). Dann lassen sich Integrale mittels Koordinatentransformation berechnen:

$$\int_\Omega f(x,y)dx\,dy = \int_{-1}^{1}\int_{-1}^{1} f\Big(\varphi(\xi,\eta),\psi(\xi,\eta)\Big) \left|\frac{\partial(\varphi,\psi)}{\partial(\xi,\eta)}\right| d\xi d\eta$$

wobei die Transformation

$$\begin{pmatrix} \varphi(\xi,\eta) \\ \psi(\xi,\eta) \end{pmatrix} : [-1,1]\times[-1,1] \to \Omega$$

das Gebiet $[-1,1]\times[-1,1]$ auf Ω abbildet.



Weiter ist

$$\left|\frac{\partial(\varphi,\psi)}{\partial(\xi,\eta)}\right| = det \begin{pmatrix} \frac{\partial\varphi}{\partial\xi}(\xi,\eta) & \frac{\partial\psi}{\partial\xi}(\xi,\eta) \\ \frac{\partial\varphi}{\partial\eta}(\xi,\eta) & \frac{\partial\psi}{\partial\eta}(\xi,\eta) \end{pmatrix} \neq 0$$

die Determinante der (transponierten) Jacobimatrix[3] der Transformation.

Es lassen sich auch Simplizes (=Dreieck, Tetraeder) zur Unterteilung verwenden und für diese direkte Integrationsformeln herleiten.

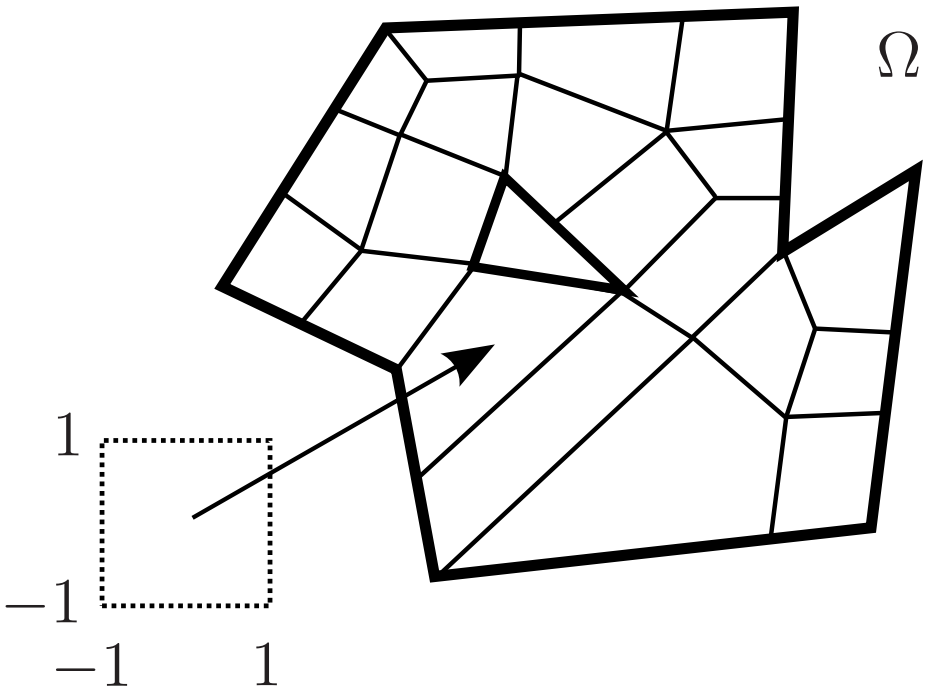
Komplexe Gebiete

Bei komplexen Gebieten, z. B. Gebieten mit Löchern oder einer komplizierten Geometrie reicht das nicht.

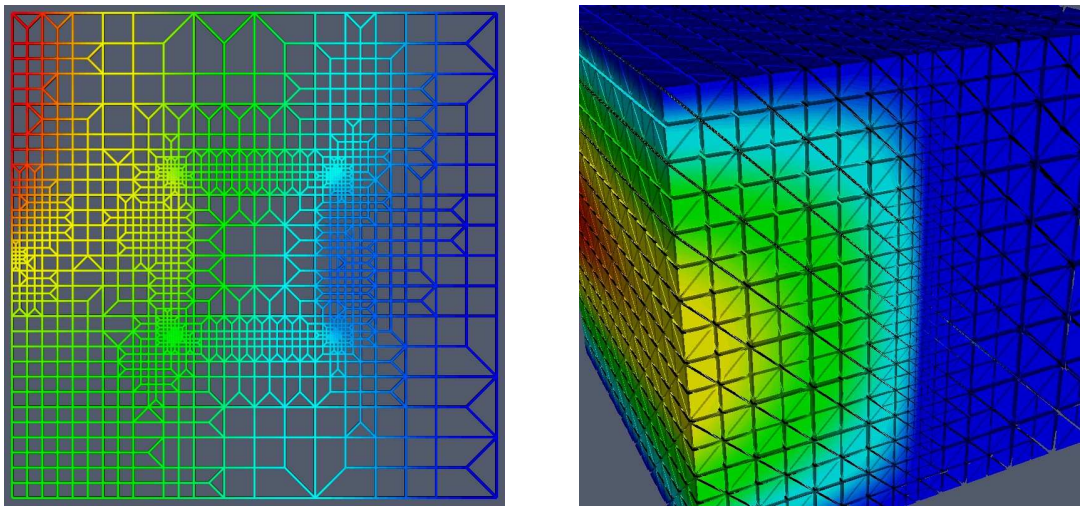
⇒ Zerlegung in Teilgebiete, die sich auf Rechtecke transformieren lassen, aber:

- "Gittergenerierung" nicht trivial und schwierig "automatisch" zu machen.
- Erfordert Beschreibung des Gebietes Ω.
- Zusätzlicher Geometriefehler durch nicht exakte Darstellung des Gebietes.

[3] Carl Gustav Jacob Jacobi, 1804-1851, dt. Mathematiker



Auch in mehr als einer Raumdimensionen kann man hierarchisch adaptiv verfeinern:



Links: Adaptives Dreiecks- und Viereckgitter. Rechts: Adaptives Tetraedergitter mit Bisektionsverfeinerung.

Fluch der Dimension

- Ist d sehr groß, so sind die hier behandelten Methoden nicht brauchbar.

- Betrachte $\Omega = [0,1]^d$. Zerlegt man $[0,1]$ in zwei Teilintervalle je Richtung, so hat man den d-dimensionalen Würfel in 2^d Teilwürfel zerlegt. $\Rightarrow$ Der Aufwand steigt exponentiell in d an. Dies bezeichnet man als "Flux der Dimension".
- Eine Möglichkeit ist dann die Monte-Carlo Integration

$$I(f) \approx \frac{C}{N} \sum_{i=1}^{N} f(\xi_i)$$

  mit Zufallszahlen $\xi_i \in \Omega$.

6 Ein kleiner Programmierkurs

6.1 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 Textdateien 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**.
5. Falls Ergebnis nicht korrekt, gehe nach 1!

Wichtige UNIX-Befehle

`ls`	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 &`

```
// hallowelt.cc (Dateiname als Kommentar)
#include <iostream>  // notwendig zur Ausgabe

int main ()
{
  std::cout << "Numerik␣0␣ist␣leicht:" << std::endl;
  std::cout << "1+1=" << 1+1 << std::endl;
}
```

- `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
```

6.2 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);`
- Ohne Initialisierung ist der Wert von Variablen der „eingebauten“ Typen nicht definiert (hängt davon ab was gerade zufällig an der Stelle im Speicher stand).

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

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.

```
// string.cc
#include <iostream>
#include <string>
int main ()
{
  std::string m1("Zeichen");
  std::string leer("␣␣␣");
  std::string m2("kette");
  std::cout << m1+leer+m2 << std::endl;
}
```

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

Mehr Ein- und Ausgabe

```
// eingabe.cc
#include <iostream> // header für Ein-/Ausgabe
#include <iomanip>  // für setprecision
#include <cmath>    // für sqrt
int main ()
{
  double x(0.0);
  std::cout << "Gebe␣eine␣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`. `std::scientific` sorgt für eine Ausgabe im Format `mantisseeexponent` (z.B. `1.0e+04`, `std::showpoint` erzwingt die Ausgabe von Nullen am Ende.
- Für die Verwendung der Manipulatoren mit Argument muss die Headerdatei `iomanip` eingebunden werden.
- Die Funktion `sqrt` berechnet die Wurzel, dafür ist die Headerdatei `cmath` notwendig.

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 zu, y unverändert
y = (y*3)+4;    // Werte Ausdruck rechts von = aus
                // und weise das Resultat y zu!
```

Blöcke

- Block: Sequenz von Variablendefinitionen und Anweisungen 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, es ist (fast) nicht vorgeschrieben, aber extrem nützlich.
- `#include`-Direktiven müssen *immer* einzeln auf einer Zeile stehen.
- Ist das folgende Programm lesbar?

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

6.3 Entscheidung

`If`-Anweisung

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

  Für $x \in \mathbb{R}$ setze

$$y = |x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \textit{sonst} \end{cases}$$

- 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 Variablen/Werte!

6.4 Wiederholung

`While`-Schleife

- Bisher: Sequentielle Abfolge von Befehlen wie im Programm angegeben.
- 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:
$$\phi(t) = \phi_0 \cos\left(\sqrt{\frac{g}{l}}t\right).$$
- 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:

```
// pendelwhile.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen
int main ()
{
  double l(1.34);   // Pendellänge in Meter
  double phi0(0.2); // Amplitude im Bogenmaß
  double dt(0.05);  // Zeitschritt in Sekunden
  double T(30.0);   // Ende in Sekunden
  double t(0.0);    // Anfangswert

  while ( t<=T )
  {
    std::cout << t << "␣"
              << phi0*cos(sqrt(9.81/l)*t)
              << std::endl;
    t = t + dt;
  }
}
```

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.
- Wird die *Schleifenvariable* direkt in der `for`-Anweisung definiert, so ist sie nur innerhalb des Schleifenkörpers sichtbar.
- Die `for`-Schleife kann auch mittels einer `while`-Schleife realisiert werden.

Pendel (analytische Lösung, `for`-Schleife)

```
// pendel.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen
int main ()
{
  double l(1.34);   // Pendellänge in Meter
  double phi0(0.2); // Amplitude im Bogenmaß
  double dt(0.05);  // Zeitschritt in Sekunden
  double T(30.0);   // Ende in Sekunden
  for (double t=0.0; t<=T; t=t+dt)
  {
    std::cout << t << "␣"
              << phi0*cos(sqrt(9.81/l)*t)
              << std::endl;
  }
}
```

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 Programms in eine Datei: `$ ./pendel > pendel.dat`
- Starte `gnuplot $ 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;
```

besser:

```
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 $\phi^n = \phi(n\Delta t)$, $u^n = u(n\Delta t)$:

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

Pendel (expliziter Euler)

```
// pendelnumerisch.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

int main ()
{
  double l(1.34);  // Pendellänge in Meter
  double phi(3.0); // Anfangsamplitude in Bogenmaß
  double u(0.0);   // Anfangsgeschwindigkeit
  double dt(1E-4); // Zeitschritt in Sekunden
  double T(30.0);  // Ende in Sekunden
  double t(0.0);   // Anfangszeit

  std::cout << t << "␣" << phi << std::endl;
  while (t<T)
  {
    t = t + dt;         // inkrementiere Zeit
    double phialt(phi);// merke phi
    double ualt(u);    // merke u
    phi = phialt + dt*ualt;            // neues phi
    u = ualt - dt*(9.81/l)*sin(phialt);// neues u
    std::cout << t << "␣" << phi << std::endl;
  }
}
```

Übungsaufgaben

1. Tippen und Übersetzen Sie das “Hallo Welt”-Programm.

2. Schreiben Sie ein Programm, das Sie nach Ihrem Vornamen, Nachnamen und Alter fragt und anschließend etwas in der folgenden Art auf den Bildschirm schreibt:

```
Ihr Name ist Olaf Ippisch.
Sie sind 41 Jahre alt.
```

3. Verändert Sie das Programm so, dass es ausrechnet wievielen Monaten, Tagen, Stunden, Minuten und Sekunden Ihr Alter entspricht (Sie dürfen Schaltjahre vernachlässigen). Die Ausgabe sollte in etwa so aussehen:

```
Ihr Name ist Olaf Ippisch.
Sie sind 41 Jahre alt.
Das entspricht
492 Monaten
oder 14965 Tagen
oder 359160 Stunden
oder 21549600 Minuten
oder 1292976000 Sekunden.
```

4. Bei der Fibonacci Folge: 1 1 2 3 5 8 13 21 34 $\cdots$ erhält man das nächste Folgenglied durch Addieren der jeweils letzten zwei Glieder der Folge:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n) &= Fib(n-1) + Fib(n-2) \end{aligned}$$

   Schreiben Sie ein Programm, das vom Benutzer die Anzahl von Folgengliedern abfragt und dann entsprechend viele Elemente der Fibonacci Folge ausgibt.

6.5 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:

Rückgabetyp Funktionsname (*Argumentliste*) { *Funktionsrumpf* }

- Beispiel:

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

Komplettbeispiel zur Funktion

```
// funktion.cc
#include <iostream>

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

int main ()
{
  double x(2.0);
  std::cout << "f(" << x << ")=" << f(x) << std::endl;
}
```

- 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!
}
```

- 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 Argumente werden durch Kommata getrennt:

```
double g (int i, double x)
{
  return i*x;
}

std::cout << g(2,3.14) << std::endl;
```

Referenzargumente

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

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

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

- Statt eines Rückgabewertes kann man also 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.

Pendelsimulation als Funktion

```
// pendelmitfunktion.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

void simuliere_pendel (double l, double phi, double u)
{
  double dt   = 1E-4;
  double T    = 30.0;
  double t    = 0.0;

  std::cout << t << "␣" << phi << std::endl;
  while (t<T)
  {
    t = t + dt;
    double phialt(phi),ualt(u);
    phi = phialt + dt*ualt;
    u = ualt - dt*(9.81/l)*sin(phialt);
    std::cout << t << "␣" << phi << std::endl;
  }
}

int main ()
{
  double l(1.34); // Pendellänge in Meter
  double phi(3.0);// Anfangsamplitude in Bogenmaß
  double u(0.0);  // Anfangsgeschwindigkeit
  simuliere_pendel(l,phi,u);
}
```

6.6 Funktionsschablonen

- Oft macht eine Funktion mit Argumenten verschiedenen Typs einen Sinn.
- z.B. braucht man die Funktion

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

oft auch mit `float` oder `int`.

- Man könnte die Funktion für jeden Typ definieren, z.B.

```
float Square (float x)
{
  return x*x;
}
```

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 Square(T y)
{
  return y*y;
}
```

- `T` steht hier für einen beliebigen Typ.

Pendelsimulation mit Templates

```
// pendelmitfunktionstemplate.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen

template<typename Number>
void simuliere_pendel (Number l, Number phi, Number u)
{
  Number dt(1E-4);
  Number T(30.0);
  Number t(0.0);
  Number g(9.81/l);

  std::cout << t << "␣" << phi << std::endl;
  while (t<T)
  {
    t = t + dt;
    Number phialt(phi),ualt(u);
    phi = phialt + dt*ualt;
    u = ualt - dt*g*sin(phialt);
    std::cout << t << "␣" << phi << std::endl;
  }
}

int main ()
{
  float l1(1.34);  // Pendellänge in Meter
  float phi1(3.0); // Anfangsamplitude in Bogenmaß
  float u1(0.0);   // Anfangsgeschwindigkeit
```

```
  simuliere_pendel(l1,phi1,u1);

  double l2(1.34);  // Pendellänge in Meter
  double phi2(3.0); // Anfangsamplitude in Bogenmaß
  double u2(0.0);   // Anfangsgeschwindigkeit
  simuliere_pendel(l2,phi2,u2);
}
```

6.7 HDNUM

- C++ kennt keine Matrizen und Vektoren, ...
- 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_ss2011/`
5. Klicke auf **Version 0.20, Stand 26.04.2011** und bestätige
6. Kopiere Datei `hdnum-0.20.tgz` in das Verzeichnis: `$ cp ~/Desktop/hdnum-0.20.tgz .`
7. Entpacken der Datei mit `$ tar xzvf hdnum-0.20.tgz`
8. Wechsle in das Verzeichnis `$ cd hdnum/examples`
9. Anzeigen der Dateien mittels `$ ls`

Bestandteile von HDNUM

- Vektoren
- Matrizen
- Timer

Vektoren in C++

- Die Klasse `std::vector` aus der Standard Template Library ist der komfortabelste Weg in C++ ein Feld von Werte anzulegen.
- Ein solcher Vektor ist eine Menge von Werten. Die Elemente können über einen Index in eckigen Klammern angesprochen werden. Der erste Index ist *Null*.
- Um einen Standardvektor verwenden zu können muss die Headerdatei `vector` mit `#include <vector>` eingebunden werden.
- Vektoren werden wie normale Variablen angelegt. Der Variablentyp ist `std::vector<typ>`, wobei `typ` der Variablentyp der Elemente ist (Vektoren sind also eher Schablonen für Felder).

```
std::vector<int> intVector;
```

- Die Länge des Vektors (die Anzahl der Elemente) kann in runden Klammern nach dem Variablennamen angegeben werden.

```
std::vector<int> intVector(7);
```

Verwendung von `std::vector`

- Nach der Größe lässt sich ein Defaultwert für die Elemente angeben. Andernfalls ist der Wert der Elemente nicht definiert.

```
std::vector<int> intVector(7,0);
```

- Vektoren können als Kopie eines existierenden Vektors angelegt werden:

```
std::vector<int> intVector(7,0);
std::vector<int> secondVector(intVector);
```

- Die einzelnen Elemente werden durch Angabe des Index in eckigen Klammern nach dem Variablennamen ausgewählt. Der Index des ersten Elements ist `0`, der Index des letzten Elements ist `size-1`

```
intVector[1] = 3; // setzt den Wert des zweiten Elements auf 3
```

Verwendung der Methoden von `std::vector`

Vektoren haben spezielle Funktionen (Methoden), die mittels Variablenname gefolgt von einem Punkt und dem Namen der Funktion aufgerufen werden, z.B.

```
int size = intVector.size();    // size() liefert die Länge
                                // des Vektors zurück
```

Methodenname	Zweck
`size()`	gibt Länge des Vektors zurück
`resize(int newSize)`	ändert die Länge des Vektors. Zusätzliche Elemente werden nicht initialisiert. Ist der neue Vektor kürzer, wird der Rest abgeschnitten.
`front()`	liefert eine Referenz auf das erste Element
`back()`	liefert eine Referenz auf das letzte Element
`push_back(value)`	fügt ein Element am Ende hinzu (und erhöht die Länge um eins)
`clear()`	Löscht alle Elemente (Länge ist anschließend Null)

HDNUM Vektor

- `std::vector` sieht keine mathematischen Operationen mit Vektoren vor. Wir haben deshalb eine erweiterte Klasse `hdnum::Vector` geschaffen.
- Sie steht nach Einbinden des Headers `hdnum.hh` mit `#include "hdnum.hh"` zur Verfügung.
- Beim Übersetzen des Programms muss der Pfad zu dem Verzeichnis `hdnum` hinter der Option `-I` angeben werden, damit der Compiler die Headerdateien findet, z.B. zur Übersetzung des Programms `vektoren.cc` im Unterverzeichnis `examples` von `hdnum`

```
g++ -I.. -o vektoren vektoren.cc
```

```
// vektoren.cc
#include <iostream>     // notwendig zur Ausgabe
#include "hdnum.hh"     // hdnum header

template<class T>
void product (hdnum::Vector<T> &x)
{
  for (int i=1; i<x.size(); i=i+1)
    x[i] = x[i]*x[i-1];
}

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

int main ()
{
  // Konstruktion
  hdnum::Vector<float> x(10);         // Vektor mit 10 Elementen
  hdnum::Vector<double> y(10,3.14);   // 10 Elemente initialisiert
  hdnum::Vector<float> a;             // ein leerer Vektor
  x.resize(117);               // vergrößern, Daten gelöscht!
  x.resize(23,2.71);           // verkleinern geht auch

  // Zugriff auf Vektorelemente
  for (int i=0; i<x.size(); i=i+1)
  x[i] = i;                   // Zugriff auf Elemente

  // Kopie und Zuweisung
```

```
  hdnum::Vector<float> z(x);          // Kopie erstellen
  z[2] = 1.24;                  // Wert verändern

  a = z;                        // hat Werte von z
  a[2] = -0.33;
  a = 5.4;                      // Zuweisung an alle Elemente

  hdnum::Vector<float> w(x);

  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
  x[0] = w*z;                   // skalare Multiplikation
  std::cout << x.two_norm() << std::endl; // euklidische Norm

  // Ausgabe
  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

  // 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]

  // Funktionsaufruf
  product(x);
  std::cout << "x=" << x << std::endl;
  std::cout << sum(x) << std::endl;
}
```

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
```

HDNUM Matrix

- In C++ gibt es keine Standardtypen für Matrizen.
- Deshalb führt HDNUM auch einen Datentyp `DenseMatrix<typ>` ein.
- Er steht ebenfalls nach Einbinden des Headers `hdnum.hh` zur Verfügung.

```
// matrizen.cc
#include <iostream>     // notwendig zur Ausgabe
#include "hdnum.hh"     // hdnum header

// Beispiel wie man A und b für ein
// Gleichungssystem initialisieren könnte
template<class T>
void initialize (hdnum::DenseMatrix<T> &A, hdnum::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;
  }
}

int main ()
{
  // Konstruktion
  hdnum::DenseMatrix<float> A;             // leere Matrix mit Größe 0x0
  hdnum::DenseMatrix<float> B(10,10);      // 10x10 Matrix uninitialisiert
  hdnum::DenseMatrix<float> C(10,10,0.0); // 10x10 Matrix initialisiert

  // Zugriff auf Vektorelemente
  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

  // Kopie und Zuweisung
  hdnum::DenseMatrix<float> D(B);     // D ist eine Kopie von B
  A = D;                           // A ist nun identisch mit D!
  A[0][0] = 3.14;
  B[0][0] = 3.14;

  // Rechnen mit Matrizen und Vektoren
  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

  hdnum::Vector<float> x(10,1.0);     // make two vectors
  hdnum::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
  A.iwidth(2);                     // Stellen in Indexausgabe
  A.width(11);                     // Anzahl Stellen gesamt
  A.precision(4);                  // Anzahl Nachkommastellen
  std::cout << A << std::endl;// schöne Ausgabe

  // Hilfsfunktionen
  identity(A);                     // setze A auf Einheitsmatrix
  std::cout << A << std::endl;
```

```
  spd(A);                       // eine s.p.d. Matrix
  std::cout << A << std::endl;
  fill(x,(float)1,(float)1);
  vandermonde(A,x);             // Vandermondematrix
  std::cout << A << std::endl;
}
```

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
```

HDNUM Timing

- Für Effizienzvergleiche ist es notwendig die Laufzeit numerischer Algorithmen zu messen.
- Dazu gibt es in HDNUM den Typ `hdnum::Timer` .
- Auch dafür ist das Einbinden des Headers `hdnum.hh` notwendig.

```
// pendelmittimer.cc
#include <iostream> // header für Ein-/Ausgabe
#include <cmath>    // mathematische Funktionen
#include "hdnum.hh" // Zeitmessung
int main ()
{
  double l(1.34);   // Pendellänge in Meter
  double phi0(0.2); // Amplitude im Bogenmaß
  double dt(0.05);  // Zeitschritt in Sekunden
  double T(30.0);   // Ende in Sekunden
  hdnum::Timer zeit,zeitIter;
  for (double t=0.0; t<=T; t=t+dt)
  {
    zeitIter.reset();
    std::cout << t << "␣"
              << phi0*cos(sqrt(9.81/l)*t)
              << std::endl;
    std::cout << "Durchgang␣" << int(t/dt) << "␣brauchte␣"
              << std::scientific << zeitIter.elapsed()
              << "␣Sekunden" << std::endl;
  }
  std::cout << "Die␣Ausgabe␣aller␣Werte␣brauchte␣" << zeit.elapsed()
            << "␣Sekunden" << std::endl;
}
```

Übungsaufgaben

1. Schreiben Sie ein Programm zum Berechnen der Fibonacci-Folge mit Hilfe einer Funktion, die sich selbst aufruft (rekursive Programmierung), entsprechend der Vorschrift:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n) &= Fib(n-1) + Fib(n-2) \end{aligned}$$

2. Schreiben Sie ein Programm, das fünf `double`-Werte aus einer Datei in einen `hdnum::Vector` liest, die Werte auf dem Bildschirm ausgibt, einen neuen Vektor mit vertauschter Reihenfolge der Elemente anlegt, die beiden Vektoren addiert und das Ergebnis sowie die euklidische Norm des Vektors auf dem Bildschirm ausgibt. Lassen Sie den Vektor auf 7 Elemente vergrößern und prüfen Sie den Inhalt. Verkleinern Sie ihn auf 4 Elemente und geben Sie ihn wieder aus.

3. Die sogenannte Hilbert-Matrix der Ordnung $n \geq 1$ ist eine quadratische, symmetrisch positiv definite Matrix mit den Komponenten

$$h_{ij} = \frac{1}{i+j-1} \tag{11}$$

(Näheres unter `http://de.wikipedia.org/wiki/Hilbert-Matrix`).

a) Schreiben Sie eine Funktion, die eine per Referenz übergebene Matrix mit den Koeffizienten der Hilbert-Matrix füllt.

b) Testen Sie das Ergebnis durch Ausgabe der Matrix auf dem Bildschirm.

3. c) ***Bonusaufgabe für Fortgeschrittene***: Die Inverse der Hilbert-Matrix hat die Komponenten:

$$(H_n^{-1})_{ij} = \frac{(-1)^{i+j}}{i+j-1} \frac{(n+i-1)!(n+j-1)!}{((i-1)!(j-1)!)^2(n-i)!(n-j)!} \tag{12}$$

Schreiben Sie eine Funktion, die die Inverse der Hilbert-Matrix erstellt (dafür brauchen Sie auch eine Funktion, die die Fakultät berechnet). Testen Sie Ihr Programm durch Berechnung von $H_n \cdot H_n^{-1}$.

Lehrbücher Numerik

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

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

[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.

[Str10] Stroustrup, B.: *Einführung in die Programmierung mit C++* . Pearson Studium, 2010.

Weiterführende Literatur

[Gol91] Goldberg, D.: *What Every Computer Scientist Should Know About Floating Point Arithmetic.* Computing Surveys, 1991. `http://citeseer.ist.psu.edu/viewdoc/download;jsessionid=86013D0FEFFA6CD1A626176C5D4EF9E2?doi=10.1.1.102.244&rep=rep1&type=pdf`.

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

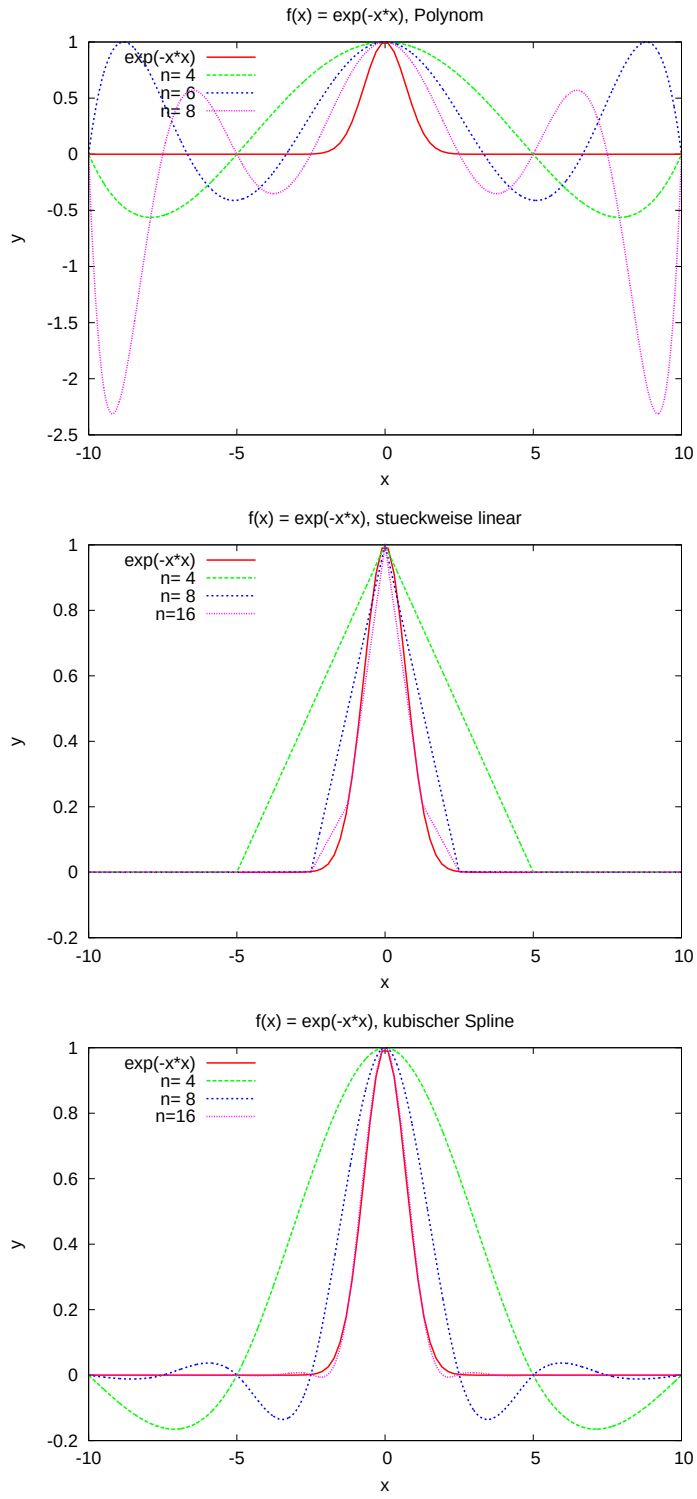


Abbildung 17: Interpolation der Funktion $f_1(x)$ mit Lagrange-Polynomen, stückweise linearen Funktionen und kubischen Splines.

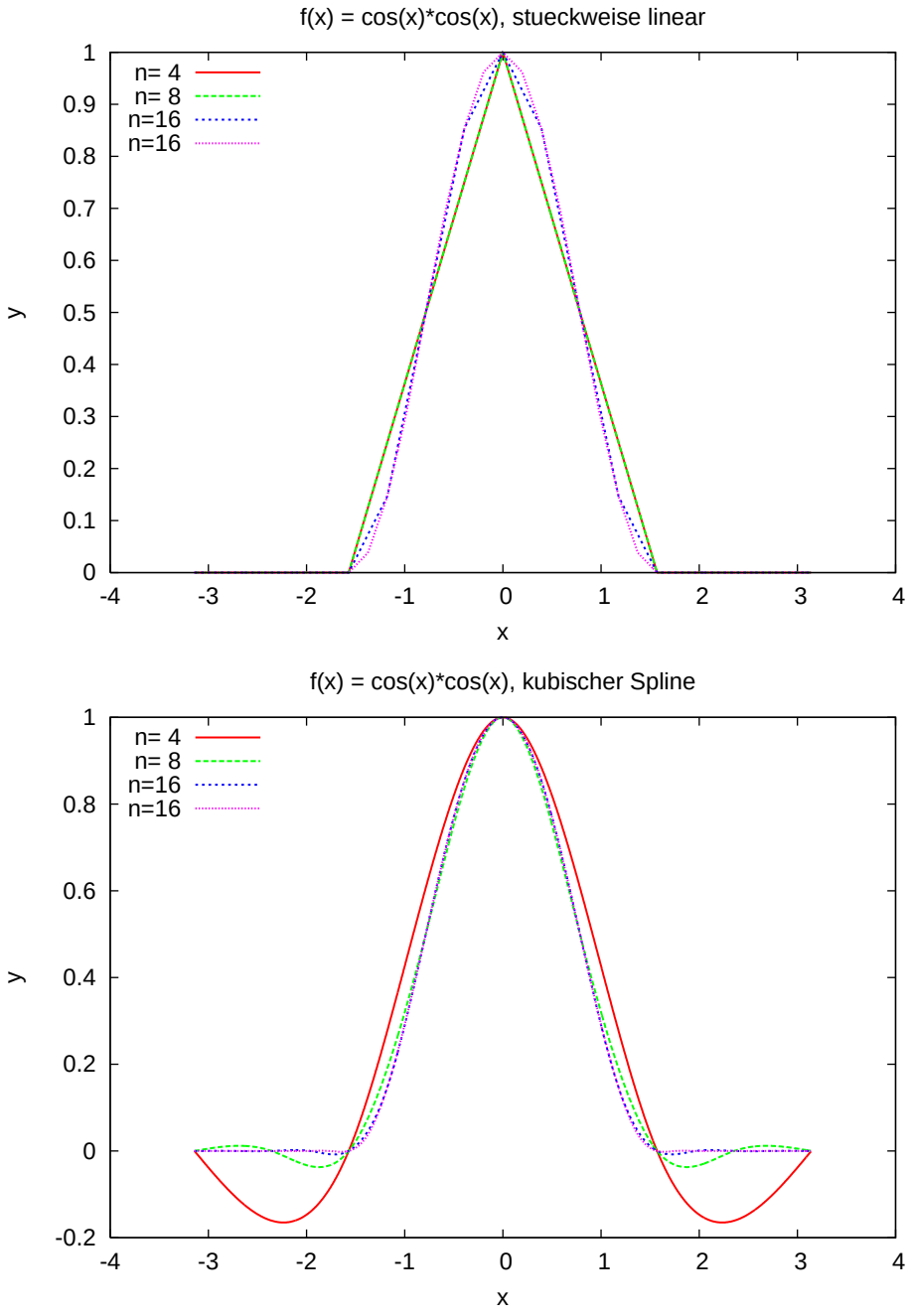


Abbildung 18: Interpolation der Funktion $f_2(x)$ mit Lagrange-Polynomen, stückweise linearen Funktionen und kubischen Splines.

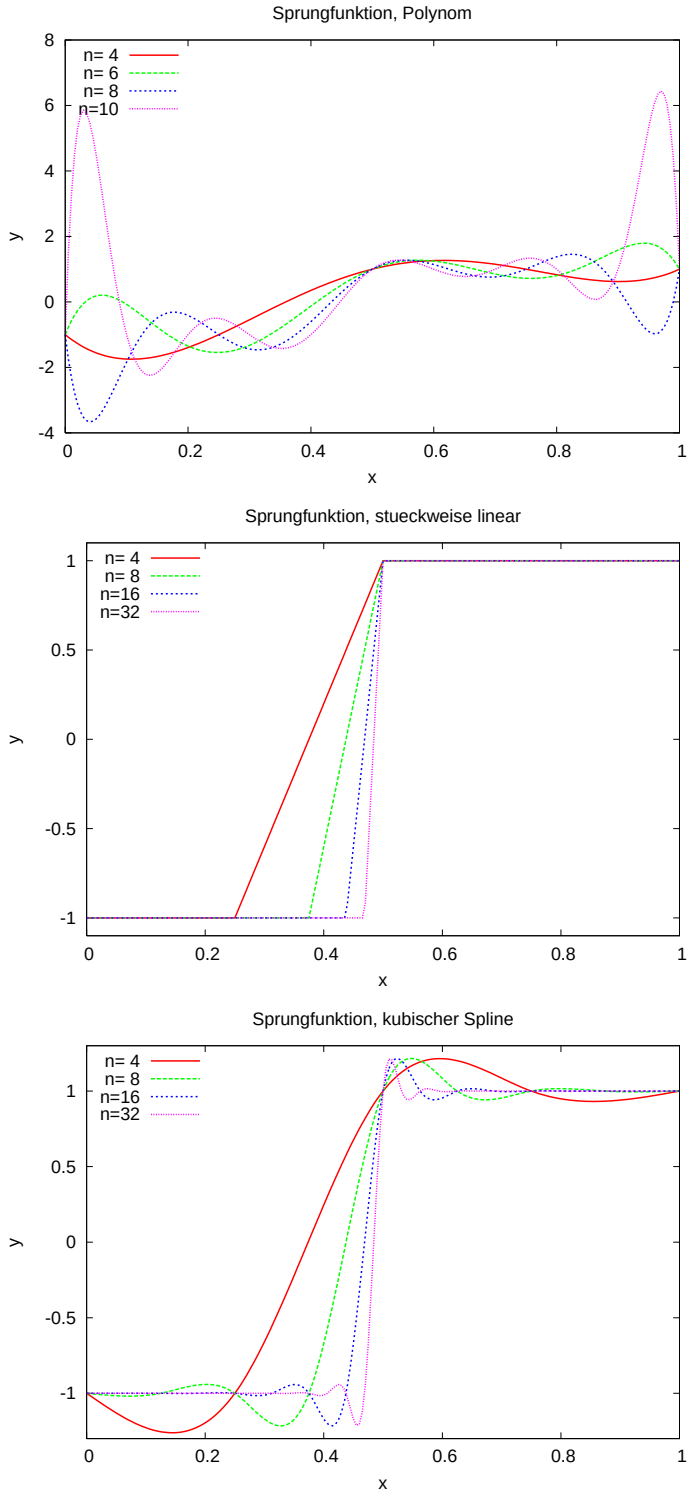


Abbildung 19: Interpolation der Funktion $f_3(x)$ mit Lagrange-Polynomen, stückweise linearen Funktionen und kubischen Splines.

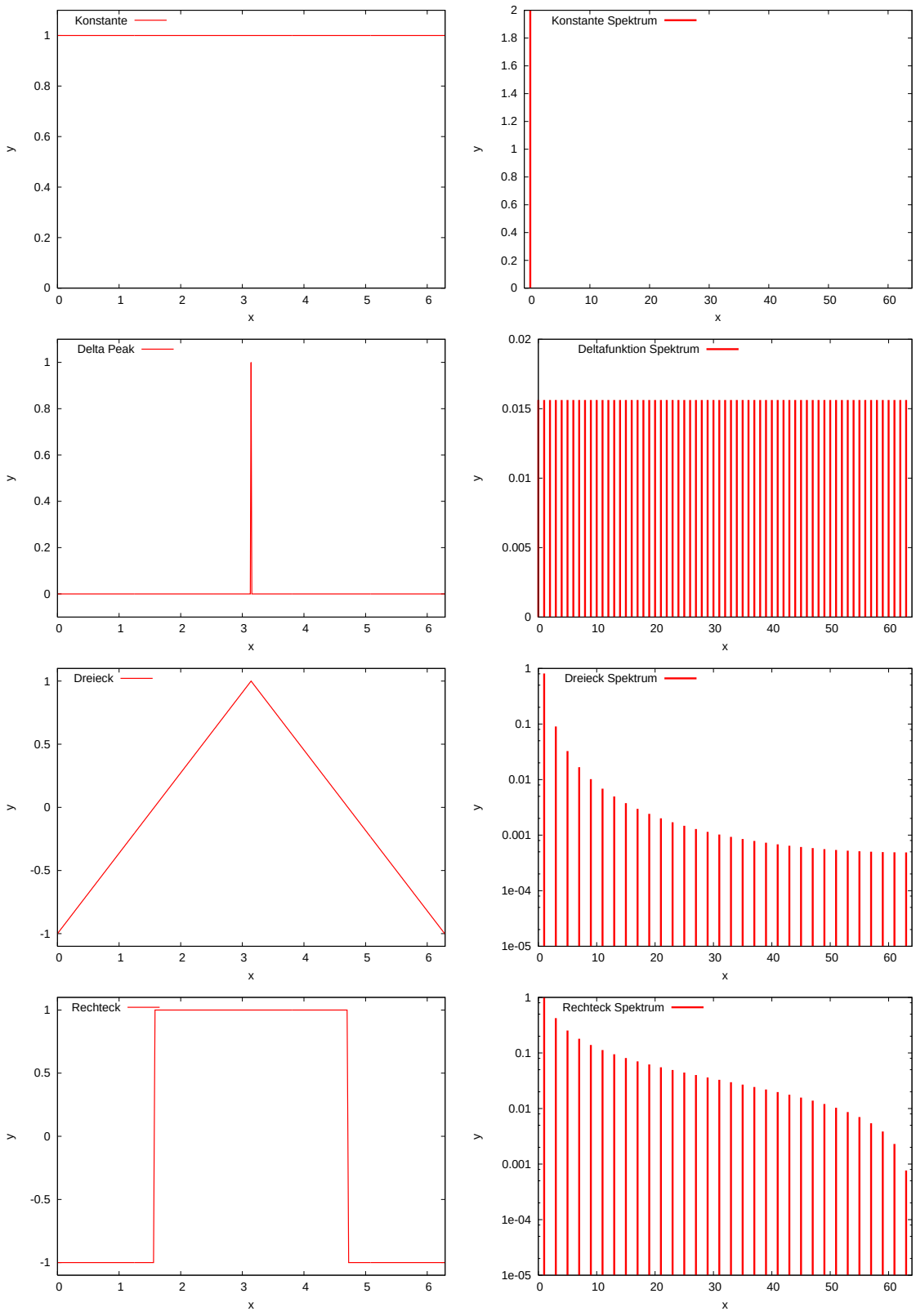


Abbildung 20: Spektren zu verschiedenen Funktionen.

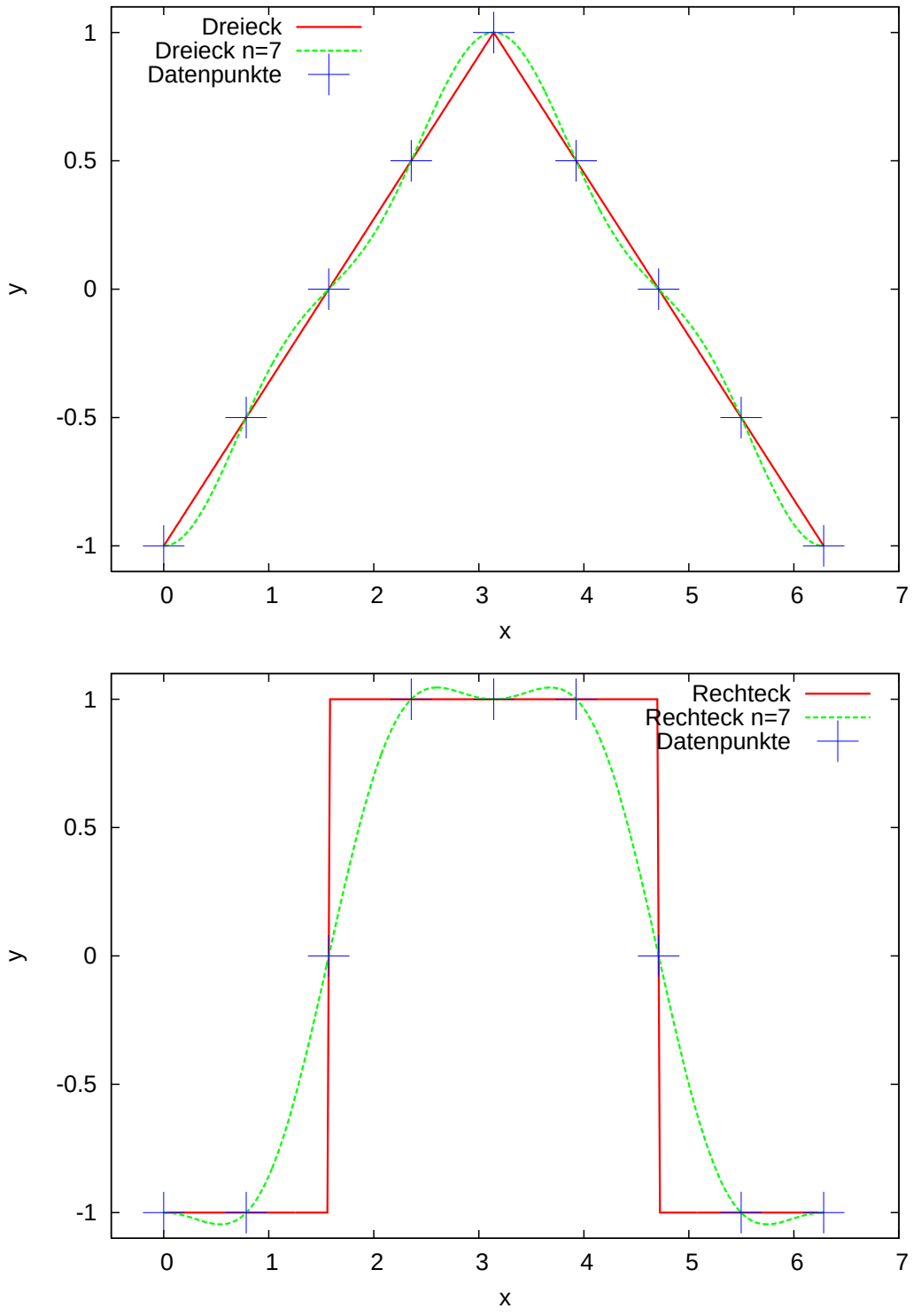


Abbildung 21: Interpolation verschiedener Funktionen.

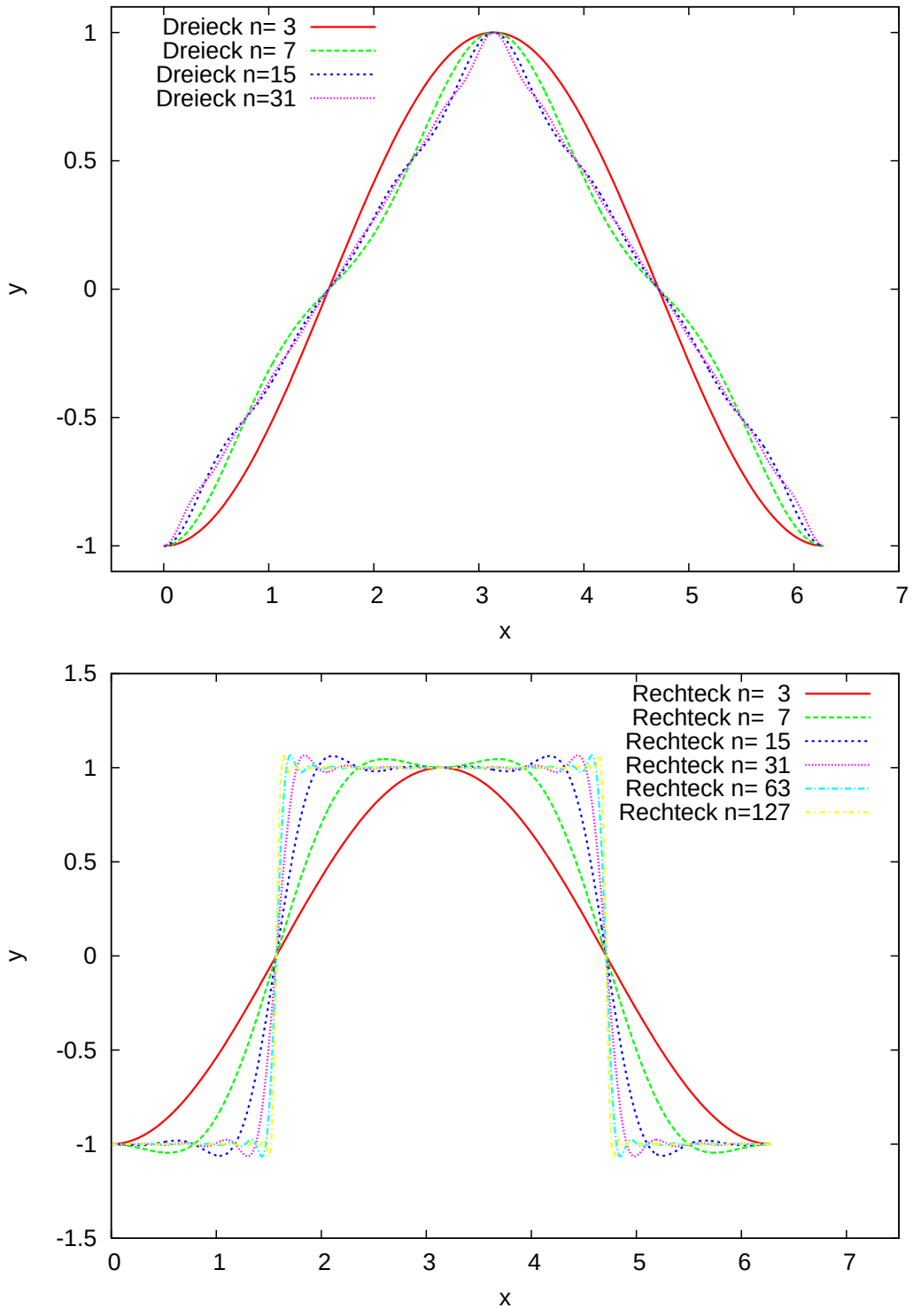


Abbildung 22: Approximation verschiedener Funktionen bei steigendem n.

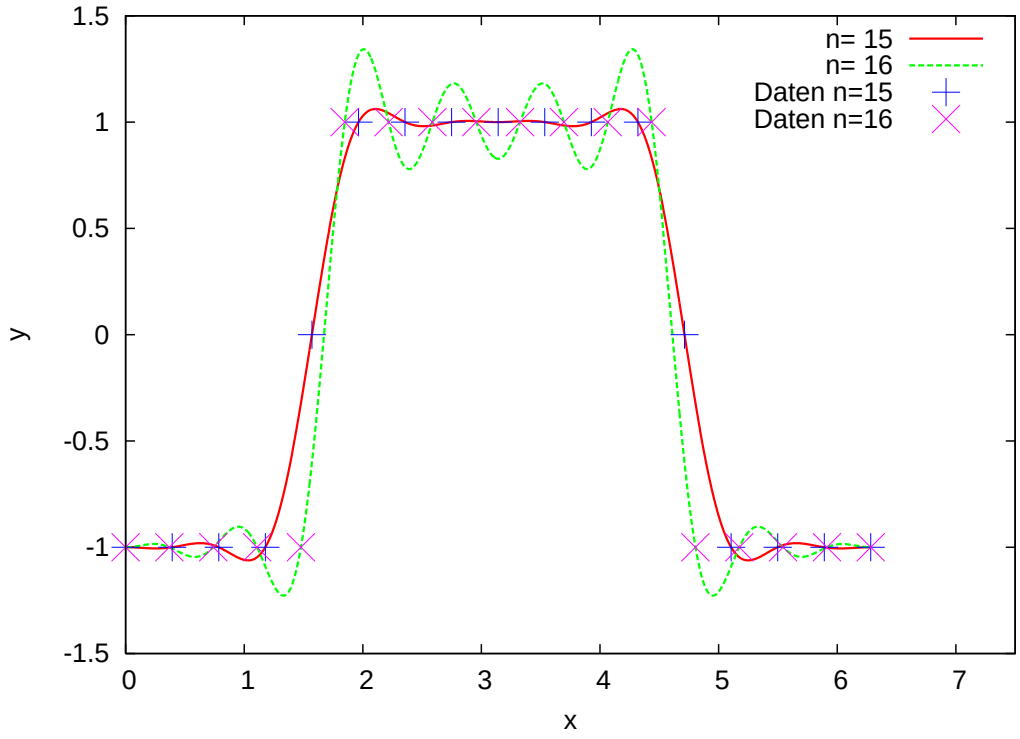


Abbildung 23: Interpolation einer unstetigen Funtion.

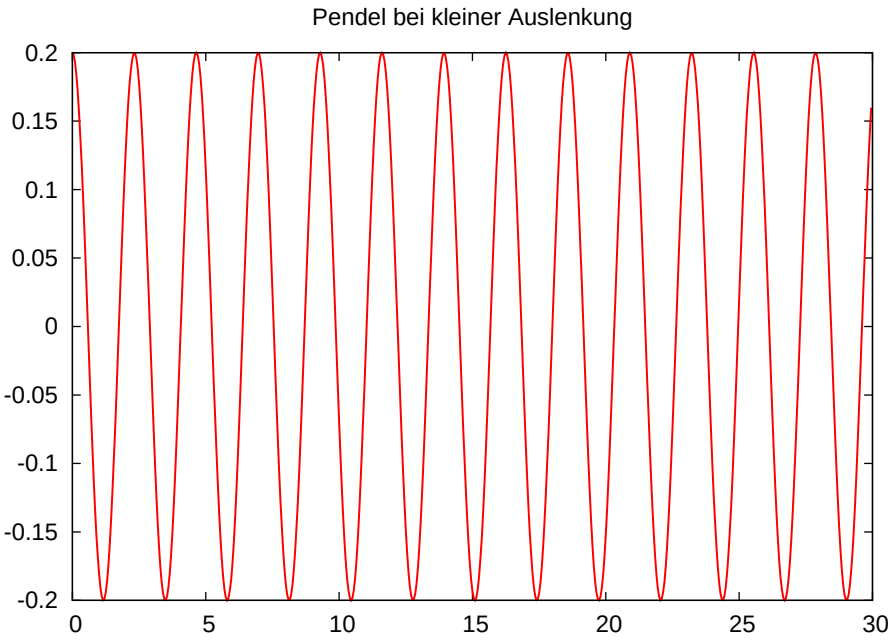


Abbildung 24: Das Pendel in Aktion. Gnuplot-Ausgabe des Programms `pendel.cc`.