Skript zur Vorlesung "Physik auf dem Computer"

JP Dr. A. Arnold
Universität Stuttgart
Institut für Computerphysik

unter Mithilfe von
Dr. O. Lenz

basierend auf Vorarbeiten von:
PD Dr. R. Hilfer, M. Lätzel, R. Mück, T. Ihle,
Prof. H. Herrmann, PD Dr. S. Luding, Dr. S. Schwarzer,
M. Müller, Dr. H.-G. Matuttis, M. Brunner, C. Manwart,
T. Karle, S. Manmana und C. Kunert

Sommersemester 2012

Inhaltsverzeichnis

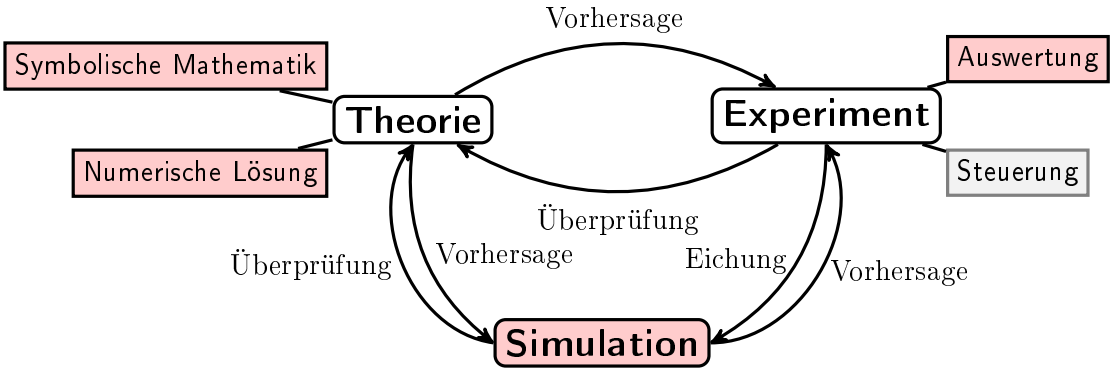
1 Einleitung **5**
- 1.1 Über dieses Skript . . . 6
- 1.2 Beispiel: Fadenpendel . . . 7
  - 1.2.1 Modell . . . 8
  - 1.2.2 Näherung: der harmonische Oszillator . . . 9
  - 1.2.3 Numerische Lösung . . . 10

2 Lineare Algebra I **15**
- 2.1 Dreiecksmatrizen . . . 15
- 2.2 Gaußelimination . . . 16
- 2.3 Matrixinversion . . . 18
- 2.4 LR-Zerlegung . . . 19
- 2.5 Bandmatrizen . . . 20
- 2.6 Cholesky-Zerlegung . . . 20

3 Darstellung von Funktionen **23**
- 3.1 Horner-Schema . . . 23
- 3.2 Taylorreihen . . . 24
- 3.3 Polynom- oder Lagrangeinterpolation . . . 25
  - 3.3.1 Lagrangepolynome . . . 26
  - 3.3.2 Neville-Aitken-Schema . . . 27
  - 3.3.3 Newtonsche Darstellung . . . 28
  - 3.3.4 Chebyshev-Stützstellen . . . 29
- 3.4 Splines . . . 30
- 3.5 Ausgleichsrechnung, Methode der kleinsten Quadrate . . . 32
- 3.6 Fourierreihen . . . 33
  - 3.6.1 Komplexe Fourierreihen . . . 34
  - 3.6.2 Reelle Fourierreihen . . . 36
  - 3.6.3 Diskrete Fouriertransformation . . . 38
  - 3.6.4 Schnelle Fouriertransformation . . . 39
- 3.7 Wavelets . . . 40

1 Einleitung

In dieser Vorlesung geht es darum, wie der Computer in der modernen Physik eingesetzt wird, um neue Erkenntnisse zu gewinnen. Klassisch war die Physik ein Zusammenspiel aus Experiment und Theorie. Die Theorie macht Vorhersagen, die im Experiment überprüft werden. Umgekehrt kann im Experiment ein neuer Effekt beobachtet werden, für den die Theorie eine Erklärung liefert. Durch den Einsatz von Computern ist dieses Bild komplizierter geworden. In der folgenden Graphik sind die Bereiche farblich hinterlegt, in denen heutzutage Computer zum Einsatz kommen, die hellroten Bereiche werden in dieser Vorlesung behandelt:



Zu den klassischen Säulen Theorie und Experiment ist die *Simulation* als Mittelding zwischen Theorie und Experiment gekommen. Computersimulationen stellen Experimente im Computer nach, ausgehend von bekannten theoretischen Grundlagen. Praktisch alles kann simuliert werden, von Galaxien bis hin zu Elektronen und Quarks. Dazu gibt es eine Vielzahl an unterschiedlichen Methoden. Simulationen erfüllen zwei Hauptaufgaben: Simulationen können einerseits Experimente ziemlich genau reproduzieren, andererseits kann man mit Ihrer Hilfe theoretische Modelle in ihrer vollen Komplexität untersuchen.

Simulationen, die an ein Experiment angepasst (geeicht) sind, können zusätzliche Informationen liefern, die experimentell nicht zugänglich sind. Zum Beispiel kann man dort Energiebeiträge getrennt messen oder sehr kurzlebige Zwischenprodukte beobachten. Außerdem erlauben Simulationen, Wechselwirkungen und andere Parameter gezielt zu verändern, und damit Vorhersagen über zukünftige Experimente zu machen.

Simulationen, die auf theoretischen Modellen basieren, sind oft ein gutes Mittel, um notwendige Näherungen auf Plausibilität zu überprüfen oder um einen ersten Eindruck vom Verhalten dieses Modells zu erhalten. Damit können Simulationen auch helfen, zu entscheiden, ob notwendige Näherungen oder das Modell unvollständig ist, wenn Theorie und Experiment nicht zu einander passen.

In der klassischen theoretischen Physik werden Papier und Bleistift zunehmend vom Computer verdrängt, denn *Computeralgebra* ist mittlerweile sehr leistungsfähig und kann zum Beispiel in wenigen Sekunden komplexe Integrale analytisch lösen. Und falls eine Gleichung doch einmal zu kompliziert ist für eine analytische Lösung, so kann der Computer mit *numerischen Verfahren* oft sehr gute Näherungen finden.

In der experimentellen Physik fallen immer größere Datenmengen an. Der LHC erzeugt zum Beispiel pro Jahr etwa 10 Petabyte an Daten, also etwa 200 Millionen DVDs, was über mehrere Rechenzentren verteilt gespeichert und ausgewertet werden muss. Klar ist, dass nur Computer diese gigantischen Datenmengen durchforsten können. Aber auch bei einfacheren Experimenten helfen Computer bei der *Auswertung und Aufbereitung* der Daten, zum Beispiel durch Filtern oder statistische Analysen. Viele Experimente, nicht nur der LHC, sind aber auch so komplex, dass Computer zur *Steuerung* der Experimente benötigt werden, was wir in dieser Vorlesung aber nicht behandeln können. Die Auswertung und Aufbereitung der Daten hingegen wird besprochen, auch weil dies genauso auch für Computersimulationen benutzt wird.

Neben diesen direkten Anwendungen in der Physik ist der Computer mittlerweile natürlich auch ein wichtiges Mittel für den Wissensaustausch unter Physikern. Quasi alle wissenschaftlichen Arbeiten, wie etwa dieses Skript, werden heute nicht mit der Schreibmaschine und Schablonen erzeugt, sondern auf dem Computer. Die großen Verlage verlangen mittlerweile auch, Manuskripte als elektronische Dokumente zur Publikation einzureichen. Umgekehrt stehen wissenschaftliche Arbeit, vor allem Zeitschriftentexte, normalerweise nur noch in elektronischen Bibliotheken zur Verfügung, dafür aber Texte aus der gesamten Welt. Zur Suche in diesen riesigen Datenmengen dienen wiederum Computer. Und schließlich ist der Computer natürlich auch unverzichtbar, um international zusammenzuarbeiten - Brief und Telefon wären schlicht zu langsam und unflexibel. Diese Aspekte wurden aber schon in den Computergrundlagen behandelt, und sind nicht Teil dieser Vorlesung.

Um den Computer für Simulationen, Auswertung von Daten oder auch Lösung komplexer Differenzialgleichungen nutzen zu können, sind neben physikalischen Kenntnissen auch solche in Programmierung, numerischer Mathematik und Informatik gefragt. In diesem Skript geht es vor allem um die grundlegenden Methoden und wie diese angewandt werden, daher dominiert die numerische Mathematik etwas. Anders als in einer richtigen Vorlesung zur Numerik stehen hier aber die Methoden und Anwendungen anstatt der Herleitungen im Vordergrund.

1.1 Über dieses Skript

Im folgenden wird eine in der numerischen Mathematik übliche Notation benutzt. Wie auch in den meisten Programmiersprachen werden skalare und vektorielle Variablen nicht durch ihre Schreibweise unterschieden, allerdings werden üblicherweise die Namen i–l für (ganzzahlige) Schleifenindizes benutzt, n und m für Dimensionen. Da Schleifen sehr häufig auftreten, wird hierfür die Kurznotation Anfang(Inkrement)Ende benutzt. Zum

Beispiel bedeuten

$$1(1)n = 1,2,\ldots,n$$
$$n(-2)1 = n, n-2, \ldots, 3, 1.$$

Alle anderen Variablen sind reellwertige Skalare oder Vektoren, $\mathbb{R}^n$ bezeichnet dabei den n-dimensionalen Vektorraum reller Zahlen. Mit e_i wird dabei der Einheitsvektor der i-ten Spalte bezeichnet, mit e_i^T seine Transponierte, also der Einheitsvektor der i-ten Zeile.

Integrale werden mit dem Volumenelement am Ende geschrieben, dessen Dimensionalität sich aus dem Integrationsbereich erschließt. Sehr häufig werden Abschätzungen mit Hilfe der *Landau*-Symbole verkürzt. Wie üblich heißt für zwei Funktionen f und g

$$f = \mathcal{O}_{x\to a}(g) \quad \Longleftrightarrow \quad \lim_{x\to a} \frac{|f(x)|}{|g(x)|} < \infty.$$

In den meisten Fällen ist $a = 0$ oder $a = \infty$ und aus dem Kontext klar, welcher Grenzwert gemeint ist. Dann wird die Angabe weggelassen. Oft wird auch die Notation $f = g + \mathcal{O}(h)$ benutzt, um $f + g = \mathcal{O}(h)$ auszudrücken. $f(x) \doteq g(x)$ schließlich bedeutet $f(x) - g(x) = \mathcal{O}x \to 0(x)$.

Um einzelne Methoden konkret vorstellen zu können, wird in diesem Skript auf die Sprachen Python und C zurückgegriffen. Im Bereich des Hochleistungsrechnens werden vor allem die Sprachen Fortran und C/C++ eingesetzt, weil diese in Verbindung mit guten Compilern sehr effizienten Code ergeben. Allerdings bieten diese Sprachen keine nativen Datentypen wie zum Bespiel Listen oder Wörterbücher und verlangen die explizite Typisierung von Variablen, was Beispiele unnötig verkompliziert. Daher benutzt dieses Skript die Programmiersprache Python[1] mit den Erweiterungen NumPy und SciPy[2], die eine leistungsfähige numerische Bibliothek und umfangreiche Visualisierungsmöglichkeiten bietet. Für elementare Beispiele hingegen greift dieses Skript auf das hardwarenähere C zurück.

Zum Erlernen der Programmiersprachen Python und C sei auf die Materialien der Veranstaltung „Computergrundlagen“ hingewiesen, die im Fachbereich Physik der Universität Stuttgart jährlich angeboten wird.

Die meisten der in diesem Skript vorgestellten numerischen Methoden werden von SciPy direkt unterstützt. Die Qualität dieser Implementationen ist mit eigenem Code nur schwierig zu überbieten. Wo immer möglich, wird daher auf die entsprechend SciPy-Befehle verwiesen. Zum Beispiel wird auf die Funktion „method“ im SciPy-Modul „library“ in der in der Form `scipy.library.method(arg1, arg2, ...)` verwiesen. Trotzdem sind viele Methoden auch als expliziter Code gezeigt, da man natürlich eine Vorstellung davon haben sollte, was diese Methoden tun.

1.2 Beispiel: Fadenpendel

Wir betrachten ein einfaches Beispielsystem, nämlich ein Fadenpendel. Wird dieses Pendel nun ausgelenkt, vollführt es eine periodische Schwingung um die Ruhelage, d.h., den

[1] www.python.org
[2] http://www.scipy.org

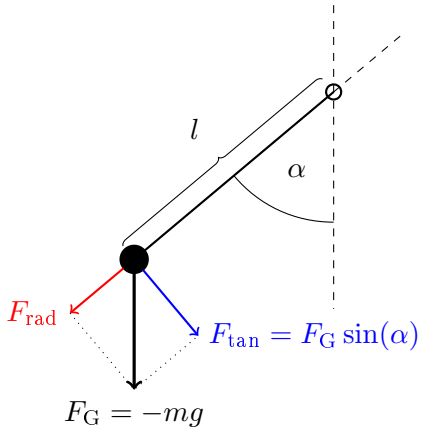


Abbildung 1.1: Schematisches Fadenpendel der Masse m, das an einem masselosen, steifen Faden der Länge l hängt.

tiefsten Punkt. Unser Ziel als Physiker ist nun, die Position der Kugel als Funktion der Zeit vorherzusagen. Das allerdings ist eine unmögliche Aufgabe — man stelle sich zum Beispiel eine stark inhomogene Masse vor (oder ein Fadenpendel als Masse) oder dass der Faden elastisch ist. Daher müssen wir zunächst ein geeignet vereinfachtes *Modell* erstellen, auf das wir dann die bekannten physikalischen Gesetze anwenden können.

1.2.1 Modell

Als Modell wählen wir eine homogene Kugel der Masse m, die an einem masselosen, steifen Faden der Länge l hängt (vergleiche Figur 1.1). Auf diese Kugel wirkt nur eine Gewichtskraft der Größe mg senkrecht nach, alle anderen Kräfte vernachlässigen wir komplett, insbesondere auch die Reibung.

Da der Faden unendlich steif sein soll, kann sich Kugel lediglich auf einem Kreis mit Radius l um die Aufhängung bewegen, d.h. die Position der Kugel ist durch die Auslenkung α aus dem tiefsten Punkt vollständig beschrieben. Weiter wird die Komponente der Kraft parallel zum Faden komplett von diesem kompensiert, daher bleibt bei Auslenkung α von der Gewichtskraft nur ihre Komponente

$$F_{\text{tan}} = F_{\text{G}} \sin(\alpha) = -mg \sin(\alpha) \tag{1.1}$$

senkrecht zum Faden übrig. Das Newtonsche Gesetz besagt nun, dass die Tangentialbeschleunigung, also die Beschleunigung entlang α

$$l\ddot{\alpha} = F_{\text{tan}}/m = -g \sin(\alpha) \tag{1.2}$$

beträgt. Dies ist jetzt eine Differentialgleichung für die Auslenkung $\alpha(t)$ des Pendels als Funktion der Zeit. Diese wiederum liefert uns die gewünschte Position $(\cos(\alpha)l, \sin(\alpha)l)$ der Kugel relativ zur Aufhängung als Funktion der Zeit. Leider hat selbst diese einfache Differentialgleichung keine geschlossene Lösung, und wir müssen weitere Näherungen einführen, um eine analytische Lösung zu erhalten.

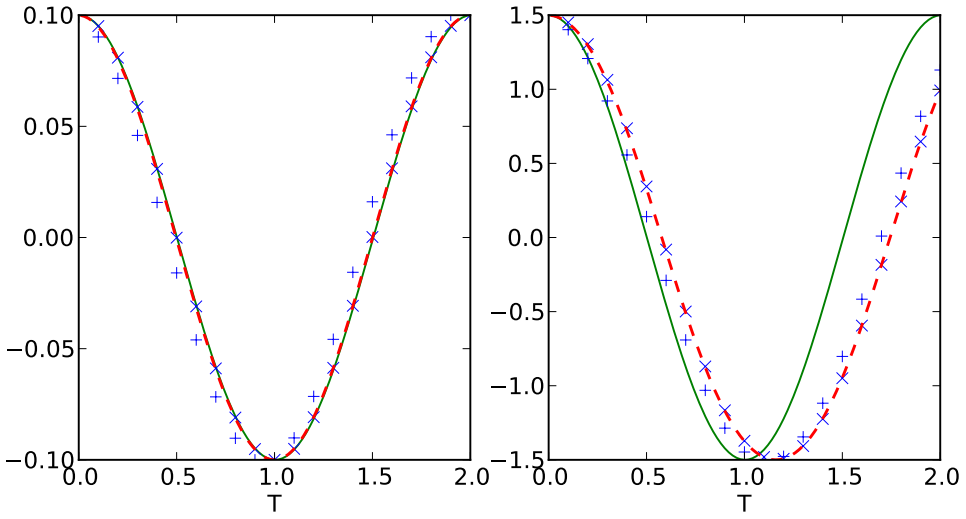


Abbildung 1.2: Lösungen für ein Fadenpendel der Länge $l = 1m$. Im linken Graphen ist die Ausgangslage $\alpha = 1{,}5$, im rechten $\alpha = 0{,}1$; in beiden Fällen ist die Ausgangsgeschwindigkeit 0. Die durchgezogene grüne Linie markiert die analytische Näherungslösung (1.4) für kleine Winkel. $\times$ markiert die Ergebnisse einer Integration mit dem einfachen Vorwärtsschritt (1.11) mit Zeitschritt $0{,}1s$, die gestrichelte rote Linie mit Zeitschritt $0{,}01s$. $+$ markiert die Lösung mit Hilfe des Velocity-Verlet-Algorithmus und Zeitschritt $0{,}1s$.

1.2.2 Näherung: der harmonische Oszillator

Für kleine Winkel gilt $\sin(\alpha) \approx \alpha$, und damit

$$\ddot{\alpha} \approx= -\frac{g}{l}\alpha. \tag{1.3}$$

Diese Differentialgleichung hat die allgemeine Lösung

$$\alpha(t) = A\sin(\omega t + \phi) \tag{1.4}$$

mit $\omega = \sqrt{g/l}$, wie man sich leicht durch Einsetzen überzeugt. Die Größen A und ϕ ergeben sich aus den Anfangsbedingungen, nämlich der Anfangsposition

$$\alpha_0 = A\sin(\phi) \tag{1.5}$$

und -geschwindigkeit

$$v_0 = A\omega\cos(\phi). \tag{1.6}$$

Ist zum Beispiel $v_0 = 0$, so ist $\phi = \pi/2$ und $A = \alpha_0$, im allgemeinen Fall ist

$$\phi = \arctan\left(\frac{\alpha_0\omega}{v_0}\right) \quad \text{und} \quad A = \frac{\alpha_0}{\sin(\phi)}. \tag{1.7}$$

Wir haben nun eine geschlossene Lösung für die Position des Pendels, so lange die Ausgangslage nicht zu sehr ausgelenkt ist. Um diese Lösung zu visualisieren, nutzt man heute üblicherweise den Computer, siehe Graph 1.2.

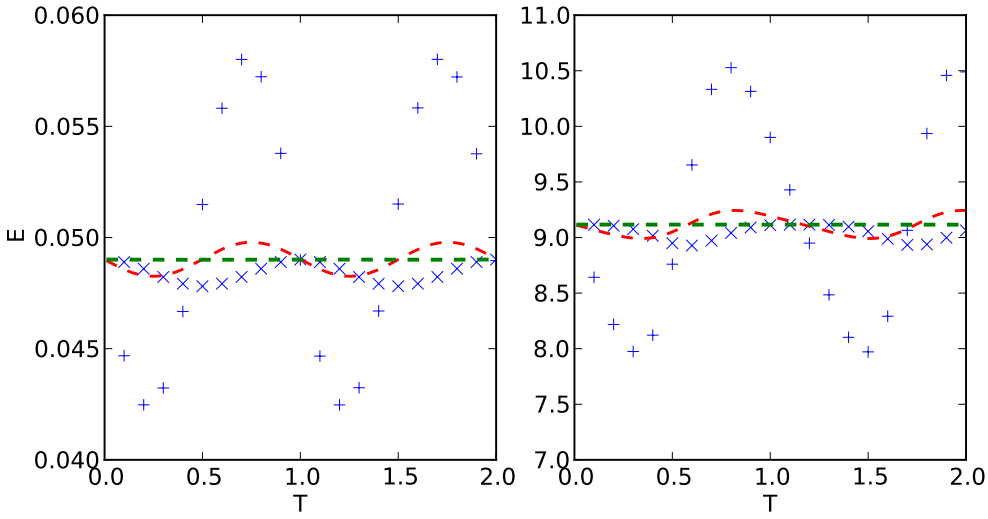


Abbildung 1.3: Energie als Funktion der Zeit, wieder für $l = 1m$, und Ausgangslage $\alpha = 1{,}5$ (links) und $\alpha = 0{,}1$ (rechts) in Ruhe. + markiert die Ergebnisse einer Integration mit dem einfachen Vorwärtsschritt (1.11) mit Zeitschritt $0{,}1s$, die gestrichelte rote Linie mit Zeitschritt $0{,}01s$. $\times$ markiert die Lösung mit Hilfe des Velocity-Verlet-Algorithmus und Zeitschritt $0{,}1s$, und die gestrichelte blaue Linie mit $0{,}01s$.

1.2.3 Numerische Lösung

Was passiert nun, wenn das System stärker ausgelenkt ist? Mit sehr viel mehr Aufwand lässt sich auch für diesen Fall eine analytische Lösung finden, allerdings in Form einer Reihe, die nicht mehr so einfach zu zeichnen ist. Eine Alternative ist, die Differentialgleichung (1.2) mit Hilfe des Computers zu berechnen. Wir sagen, wir „simulieren“ das Pendel. Dazu fixieren wir ein Einheitensystem, zum Beispiel eine Sekunde als Zeiteinheit und einen Meter als Längeneinheit. In diesem System ist also $l = 1$, $g \approx 9{,}81$ und $\omega \approx 3{,}13$, falls des Pendel einen Meter lang ist.

Zunächst müssen wir das Problem aber für den Computer anpassen, der ja nur mit (endlich vielen) gewöhnlichen Zahlen rechnen kann, wir müssen das Problem *diskretisieren*. Wir betrachten nur die Zeitpunkte

$$t_n = n\delta t, n = 1(1)N, \tag{1.8}$$

wobei der Zeitschritt δt frei wählbar ist. Je kleiner δt, desto genauer können wir $\alpha(t)$ bestimmen, allerdings steigt natürlich die Anzahl der Schritte, die nötig sind, um eine feste Gesamtzeit zu erreichen. Unsere Lösung, die Funktion $\alpha(t)$ wird also durch ihre Werte $\alpha(t_n)$ an den diskreten Zeitpunkten dargestellt.

Um Gleichung (1.2) auf den Computer zu bringen, müssen wir uns allerdings noch überlegen, wie wir mit der Ableitung verfahren. Da wir die Ausgangsposition und -ge-

schwindigkeit gegeben haben, liegt es nahe, die Gleichung zu integrieren:

$$v(t+\delta t) = \dot{\alpha}(t+\delta t) = v(t) + \int_t^{t+\delta t} -\omega^2 \sin\alpha(\tau)\, d\tau. \tag{1.9}$$

Da δt aber unser Zeitschritt ist, wir also nichts weiter über $\alpha(\tau)$ wissen, bietet sich die folgende Näherung an:

$$v(t+\delta t) \approx v(t) - \omega^2 \sin\alpha(t)\delta t. \tag{1.10}$$

Analog ergibt sich dann durch nochmalige Integration:

$$\alpha(t+\delta t) \approx \alpha(t) + v(t)\delta t. \tag{1.11}$$

Ausgehend von

$$\alpha(0) = \alpha_0 \quad \text{und} \quad v(0) = v_0 \tag{1.12}$$

lässt sich damit also $\alpha(t)$ numerisch bestimmen. Der Quellcode 1.1 zeigt, wie eine einfach Implementation in Python aussehen könnte.

Wie kann man nun überprüfen, ob diese Lösung tatsächlich korrekt ist? Da das System abgeschlossen ist, muss seine Energie

$$E = \frac{1}{2} l^2 v(t)^2 + gl(1 - \cos(\alpha(t))) \tag{1.13}$$

erhalten sein. Lässt man sich diese allerdings ausgeben, stellt man fest, dass $E(t)$ erheblich schwankt, vergleiche Graph 1.3. Dies lässt sich durch Verringern des Zeitschritts beheben, das kostet aber entsprechend mehr Rechenzeit.

Eine bessere Alternative ist, den Algorithmus zu verbessern, was wiederum etwas analytische Arbeit erfordert. Wir betrachten die Taylorentwicklungen

$$\alpha\left(t+\delta t\right) = \alpha\left(t+\frac{\delta t}{2}\right) + \frac{\delta t}{2} v\left(t+\frac{\delta t}{2}\right) + \frac{\delta t^2}{8} F\left(t+\frac{\delta t}{2}\right) + \mathcal{O}(\delta t^3) \tag{1.14}$$

und

$$\alpha\left(t\right) = \alpha\left(t+\frac{\delta t}{2}\right) - \frac{\delta t}{2} v\left(t+\frac{\delta t}{2}\right) + \frac{\delta t^2}{8} F\left(t+\frac{\delta t}{2}\right) - \mathcal{O}(\delta t^3). \tag{1.15}$$

Durch Subtraktion ergibt sich

$$\alpha(t+\delta t) = \alpha(t) + \delta t\, v\left(t+\frac{\delta t}{2}\right) + \mathcal{O}(\delta t^4). \tag{1.16}$$

Die Geschwindigkeiten an den halben Zeitschritten erhält man einfach durch $v(t+\delta t/2) = v(t) + \delta t F(t)/2$. Zusammengefasst ergibt sich der folgende *Velocity-Verlet-Algorithmus*:

$$v(t+\frac{\delta t}{2}) = v(t) + \frac{\delta t}{2} F(t) \tag{1.17}$$

$$\alpha(t+\delta t) = \alpha(t) + v(t+\frac{\delta t}{2}) \tag{1.18}$$

$$v(t+\delta t) = v(t+\frac{\delta t}{2}) + \frac{\delta t}{2} F(t+\delta t), \tag{1.19}$$

der anders als die direkte Vorgehensweise vorher numerisch stabil ist und quasi keine Energieschwankungen aufzeigt, vergleiche Graph 1.3. Im Quellcode 1.1 ist alternativ auch dieser Integrator implementiert. Obwohl er nur unwesentlich komplizierter ist als der einfache Integrator zuvor, erreicht etwa dieselbe Genauigkeit wie dieser mit einem Zehntel der Zeitschritte.

Als weiterer Test bietet sich an, bei kleinen Auslenkungen mit der analytisch bekannten Lösung zu vergleichen, die gut reproduziert wird, siehe Graph 1.2. Bei größeren Anfangsauslenkungen oder -geschwindigkeiten ist die Abweichung allerdings sehr groß, weil hier die analytische Näherung versagt. Im Rahmen ihrer Genauigkeit erlaubt also die numerische Lösung, das vorgegebene Modell in einem größeren Parameterraum auf sein Verhalten hin zu untersuchen, als analytisch möglich wäre.

```
# Simulation der Bahn eines Fadenpendels
##########################################
import scipy as sp
import matplotlib.pyplot as pyplot

# Laenge des Pendelarms
l=1
# Erdbeschleunigung
g = 9.81
# Zeitschritt
dt = 0.01
# Zeitspanne
T = 2
# Methode, "simple" oder "velocity-verlet"
integrator="velocity-verlet"
# (Start-)Position
a = 0.1
# (Start-)Winkelgeschwindigkeit
da = 0
# Zeit
t = 0

# Tabellen fuer die Ausgabe
tn, an, En = [], [], []

# Kraft, die auf die Kugel wirkt
def F(a):
    return -g/l*sp.sin(a)

while t < T:
    if integrator == "simple":
        da += F(a)*dt
        a += da*dt
    elif integrator == "velocity-verlet":
        da += 0.5*F(a)*dt
        a += da*dt
        da += 0.5*F(a)*dt
    t += dt
    tn.append(t)
    an.append(a)
    En.append(0.5*(l*da)**2 + g*(l - l*sp.cos(a)))

# Ausgabe von Graphen
ausgabe = pyplot.figure(figsize=(8,4))

loesung = ausgabe.add_subplot(121)
loesung.set_xlabel("T")
loesung.set_ylabel("Winkel")
loesung.plot(tn, an)

energie = ausgabe.add_subplot(122)
energie.set_xlabel("Zeit")
energie.set_ylabel("Energie")
energie.plot(tn, En)

pyplot.show()
```

Listing 1.1: Python-Code zum Fadenpendel mit graphisch aufbereiteter Ausgabe mit Hilfe der `matplotlib`.

2 Lineare Algebra I

Lineare Gleichungssysteme sind die einfachsten Gleichungssysteme, für deren Lösung man oft den Computer benutzt. Vor allem werden auch komplexere Probleme, wie zum Beispiel Differentialgleichungen, auf die Lösung eines Satzes von linearen Gleichungssystemen zurückgeführt. Der händischen Lösung der Systeme steht dabei vor allem ihre Größe im Weg — mit modernen Algorithmen lassen sich Systeme mit Tausenden von Variablen zuverlässig behandeln. In diesem Kapitel lernen wir die grundlegende Methode zum Lösen von Gleichungssystemen kennen, nämlich die allgemeine, aber langsame Gaußelimination. Daneben lernen wir noch die LR-Zerlegung und die Choleskyzerlegung kennen, die mit etwas Vorarbeit eine effizientere Lösung erlauben und im folgenden oft zum Einsatz kommen werden.

Wir betrachten also folgendes Problem: Sei $A = (a_{ik}) \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$. Gesucht ist die Lösung $x \in \mathbb{R}^n$ des Gleichungssystems

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array} \tag{2.1}$$

oder kurz $Ax = b$. In dieser allgemeinen Form ist weder garantiert, dass es eine Lösung gibt (z.B. $A = 0$, $b \neq 0$), noch, dass diese eindeutig ist ($A = 0$, $b \neq 0$).

2.1 Dreiecksmatrizen

Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt eine *rechte obere Dreiecksmatrix*, wenn sie quadratisch ist und $a_{ij} = 0$ für $i > j$. Analog kann man auch die linken unteren Dreiecksmatrizen definieren, mit $a_{ij} = 0$ für $i < j$. In jedem Fall bilden rechte obere und linke untere Dreiecksmatrizen jeweils Unteralgebren der Matrixalgebra, d.h., sie sind abgeschlossen unter Addition und Multiplikation. Die Schnittmenge dieser Algebren ist wiederum die Algebra der *Diagonalmatrizen*.

Ist A eine rechte obere Dreiecksmatrix, so hat das Gleichungssystem die Form

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ & & & & \ddots & & \vdots & & \vdots \\ & & & & & & a_{nn}x_n & = & b_n. \end{array} \tag{2.2}$$

Dieses Gleichungssystem hat genau dann eine Lösung, wenn A regulär ist, also $\det A = \prod_{i=1}^{n} a_{ii} \neq 0$. Die Lösung kann dann durch *Rücksubstitution* direkt bestimmt werden:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{k=i+1}^{n} a_{ik} x_k \right) \quad \text{für } i = n(-1)1. \tag{2.3}$$

Für reguläre *linke untere Dreiecksmatrizen* ergibt sich die Lösung entsprechend durch *Vorwärtssubstitution*:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{k=1}^{i-1} a_{ik} x_k \right) \quad \text{für } i = 1(1)n. \tag{2.4}$$

Für Diagonalmatrizen ist die Situation natürlich einfacher, es gilt

$$x_i = \frac{1}{a_{ii}} b_i \quad \text{für } i = 1(1)n. \tag{2.5}$$

SciPy stellt für Dreiecksmatrizen spezielle Löserroutinen zur Verfügung, `scipy.linalg.solve_triangular(A, b, lower=False)`, wobei `lower` angibt, ob `A` eine linke untere statt rechte obere Dreiecksmatrix ist.

2.2 Gaußelimination

Die Gaußelimination ist ein Verfahren, um eine beliebiges Gleichungssystem $Ax = b$, mit $A \in \mathbb{R}^{m \times n}$, auf die äquivalente Form

$$\begin{pmatrix} R & K \\ 0 & 0 \end{pmatrix} x' = b' \tag{2.6}$$

zu bringen, wobei R eine reguläre rechte obere Dreiecksmatrix und $K \in \mathbb{R}^{k \times l}$ beliebig ist, und x' eine Permutation (Umordnung) von x. Dieses Gleichungssystem hat offenbar nur dann eine Lösung, wenn $b'_i = 0$ für $i = m - k + 1(1)m$.

Diese ist im allgemeinen auch nicht eindeutig, vielmehr können die freien Variablen $x_{\mathrm{K}} = (x'_i)_{i=n-k+1}^{n}$ frei gewählt werden. Ist $x_{\mathrm{R}} = (x'_i)_{i=1}^{n-k}$ der Satz der verbleibenden Lösungsvariablen, so gilt also

$$x_{\mathrm{L}} = R^{-1} b' - R^{-1} K x_{\mathrm{K}}.$$

Die Lösungen ergeben sich daraus als

$$x' = \begin{pmatrix} R^{-1} b' \\ 0 \end{pmatrix} + \left\langle \begin{pmatrix} -R^{-1} K_i \\ e_i \end{pmatrix} \right\rangle, \tag{2.7}$$

wobei K_i die i-te Spalte von K und $\langle\rangle$ den aufgespannten Vektorraum bezeichnet. Die Ausdrücke, die R^{-1} enthalten, können durch Rücksubstitution bestimmt werden.

Um das System $Ax = b$, das wir im folgenden als $A|b$ zusammenfassen, auf diese Form zu bringen, stehen folgende Elementaroperationen zur Verfügung, die offensichtlich die Lösung nicht verändern:

1. Vertauschen zweier Gleichungen (Zeilentausch in $A|b$)
2. Vertauschen zweier Spalten in x und A (Variablenaustausch)
3. Addieren eines Vielfachen einer Zeile zu einer anderen
4. Multiplikation einer Zeile mit einer Konstanten ungleich 0

Die Gausselimination nutzt nun diese Operationen, um die Matrix spaltenweise auf die gewünschte Dreiecksform zu bringen. Dazu werden Vielfache der ersten Zeile von von allen anderen abgezogen, so dass die Gleichung die Form

$$\left(\begin{array}{cccc|c} a_{11}^{(0)} & a_{12}^{(0)} \dots & a_{1n}^{(0)} & & b_1^{(0)} \\ 0 & a_{22}^{(1)} \dots & a_{2n}^{(1)} & & b_2^{(1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & a_{m2}^{(1)} \dots & a_{mn}^{(1)} & & b_m^{(1)} \end{array}\right) =: A^{(1)}|b^{(1)} \tag{2.8}$$

annimmt, wobei

$$\begin{array}{rcll} a_{ik}^{(1)} & = & a_{ik}^{(0)} - l_i^{(1)} a_{1k}^{(0)} & \text{für } i = 2(1)n, k = 1(1)m \\ b_i^{(1)} & = & b_i^{(0)} - l_i^{(1)} b_1^{(0)} & \text{für } i = 2(1)n \qquad \text{mit } l_i^{(1)} = \dfrac{a_{i1}^{(0)}}{a_{11}^{(0)}}. \\ a_{1k}^{(1)} & = & a_{1k}^{(0)}, \quad b_1^{(1)} = b_1^{(0)} & \text{sonst} \end{array} \tag{2.9}$$

Mit dem verbleibenden Resttableau wird nun genauso weiter verfahren:

$$\begin{array}{rcll} a_{ik}^{(r)} & = & a_{ik}^{(r-1)} - l_i^{(r)} a_{r-1,k}^{(r-1)} & \text{für } i = r+1(1)n, k = r(1)m \\ b_i^{(r)} & = & b_i^{(r-1)} - l_i^{(r)} b_1^{(r-1)} & \text{für } i = r+1(1)n \qquad \text{mit } l_i^{(r)} = \dfrac{a_{ir}^{(r-1)}}{a_{rr}^{(r-1)}}. \\ a_{ik}^{(r)} & = & a_{ik}^{(r-1)}, \quad b_i^{(r)} = b_i^{(r-1)} & \text{sonst} \end{array} \tag{2.10}$$

Das Verfahren ist beendet, wenn das Resttableau nur noch eine Zeile hat.

Ist während eines Schrittes $a_{rr}^{(r-1)} = 0$ und

1. nicht alle $a_{ir}^{(r-1)} = 0$, $i = r+1(1)m$. Dann tauscht man Zeile r gegen eine Zeile i mit $a_{ir}^{(r-1)} \neq 0$, und fährt fort.
2. alle $a_{ir}^{(r-1)} = 0$, $i = r(1)m$, aber es gibt ein $a_{ik}^{(r-1)} \neq 0$ mit i,$k \geq r$. Dann vertauscht man zunächst Zeile r mit Zeile i, tauscht anschließend Spalte k mit Spalte r, und fährt fort.
3. alle $a_{ik}^{(r-1)} = 0$ für i,$k \geq r$. Dann hat $A^{(r-1)}|b^{(r-1)}$ die gewünschte Form (2.6) erreicht, und das Verfahren terminiert.

Das Element $a_{rr}^{(r-1)}$ heißt auch *Pivotelement*, da es sozusagen der Dreh- und Angelpunkt des iterativen Verfahrens ist. In der Praxis ist es numerisch günstiger, wenn dieses Element möglichst groß ist. Das lässt sich erreichen, in dem wie in den singulären Fällen verfahren wird, also Zeilen oder Spalten getauscht werden, um das betragsmäßig maximale $a_{ik}^{(r-1)}$ nach vorne zu bringen. Folgende Verfahren werden unterschieden

- *kanonische Pivotwahl*: es wird stets $a_{rr}^{(r-1)}$ gewählt und abgebrochen, falls dieses betragsmäßig zu klein wird. Diese Verfahren scheitert schon bei einfachen Matrizen (z.B. $\left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)$), und kann daher nur eingesetzt werden, wenn die Struktur der Matrix sicherstellt, das $a_{rr}^{(r-1)}$ stets hinreichend groß ist.

- *Spaltenpivotwahl*: es wird wie oben im 2. Fall nur in der Spalte maximiert, d.h. wir wählen als Pivotelement
$$i_0 = \text{argmax}_{i>r}|a_{ir}^{(r-1)}| \tag{2.11}$$
und tauschen Zeilen i_0 und r; die Variablenreihenfolge bleibt unverändert. Ist die Matrix A quadratisch, bricht das Verfahren genau dann ab, wenn A singulär ist.

- *Totalpivotwahl*: wie oben im 3. Fall wird stets das maximale Matrixelement im gesamten Resttableau gensucht, also
$$i_0,k_0 = \text{argmax}_{i,k>r}|a_{ik}^{(r-1)}|. \tag{2.12}$$
Dann vertauscht man zunächst Zeile r mit Zeile i_0, und tauscht anschließend Spalte k_0 mit Spalte r, wobei man sich noch die Permutation der Variablen geeignet merken muss, zum Beispiel als Vektor von Indizes.

Unabhängig von der Pivotwahl benötigt die Gaußelimination bei quadratischen Matrizen im wesentlichen $\mathcal{O}(n^3)$ Fließkommaoperationen. Das ist relativ langsam, daher werden wir später bessere approximative Verfahren kennenlernen. Für Matrizen bestimmter Struktur, zum Beispiel Bandmatrizen, ist die Gaußelimination aber gut geeignet. NumPy bzw. SciPy stellen daher auch keine Gaußelimination direkt zur Verfügung. `scipy.linalg.solve(A, b)` ist allerdings ein Löser für Gleichungssysteme $Ax = b$, der auf der LR-Zerlegung durch Gaußelimination basiert. Dieser Löser setzt allerdings voraus, dass die Matrix nicht singulär ist, also eindeutig lösbar.

2.3 Matrixinversion

Ist $A \in \mathbb{R}^{n\times n}$ regulär, so liefert die Rücksubstitution implizit die Inverse von A, da für beliebige b das Gleichungssystem $Ax = b$ gelöst werden kann. Allerdings muss das für jedes b von neuem geschehen. Alternativ kann mit Hilfe der Gaußelimination auch die Inverse von A bestimmt werden. Dazu wird das Tableau $A|I$ in das Tableau $I|A^{-1}$ transformiert, wobei I die $n\times n$-Einheitsmatrix bezeichnet. Die Vorgehensweise entspricht zunächst der Gaußelimination mit Spaltenpivotwahl. Allerdings werden nicht nur die Elemente unterhalb der Diagonalen, sondern auch oberhalb eliminiert. Zusätzlich wird die Pivotzeile noch mit $1/a_{ii}^{(i-1)}$ multipliziert, so dass das A schrittweise die Form
$$\begin{pmatrix} 1 & 0 & a_{12}^{(2)} & \dots & a_{1n}^{(2)} \\ 0 & 1 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & 0 & a_{32}^{(2)} & \dots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix} \tag{2.13}$$

annimmt. Das Verfahren ist allerdings numerisch nicht sehr stabil, und generell sollte die explizite Berechnung der Inversen wann inner möglich vermieden werden. SciPy stellt die Matrixinversion als Funktion `scipy.linalg.inv(A)` zur Verfügung.

Eine Ausnahme bilden Matrizen der Form $I + A$ mit $\|A\| = \max\|Ax\|/\|x\| < 1$. Dann ist

$$(I + A)^{-1} = I - A + A^2 - A^3 + \cdots \tag{2.14}$$

eine gut konvergierende Näherung der Inversen.

2.4 LR-Zerlegung

Eine weitere Anwendung der Gaußelimination ist die LR-Zerlegung von bestimmten quadratischen Matrizen. Dabei wird eine Matrix $A \in \mathbb{R}^{n\times n}$ so in eine ein linke untere Dreiecksmatrix L und eine rechte obere Dreiecksmatrix R zerlegt, dass $A = L \cdot R$. Um die LR-Zerlegung eindeutig zu machen, vereinbart man üblicherweise, dass $l_{ii} = 1$ für $i = 1(1)n$.

Ist eine solche Zerlegung einmal gefunden, lässt sich das Gleichungssystem $Ax = b$ für beliebige b effizient durch Vorwärts- und Rücksubstitution lösen:

$$Ly = b, Rx = y \quad \Longrightarrow \quad Ax = L\,Rx = Ly = b. \tag{2.15}$$

Zunächst wird also y durch Vorwärtssubstitution berechnet, anschließend x durch Rückwärtssubstitution. Die Inverse lässt sich so auch bestimmen:

$$Ly_i = e_i, Rx_i = y_i \quad \text{für } i = 1(1)n \quad \Longrightarrow \quad A^{-1} = (x_1, \ldots, x_n)\,. \tag{2.16}$$

Die Determinante von $A = L \cdot R$ ist ebenfalls einfach zu bestimmen:

$$\det A = \det L \, \det R = \prod_{i=1}^{n} r_{ii} \tag{2.17}$$

Um die LR-Zerlegung zu berechnen, nutzen wir wieder die Gaußelimination. Kann bei $A \in \mathbb{R}^{n\times n}$ die Gaußelimination in kanonischer Pivotwahl durchgeführt werden, so sei $R = A^{(n-1)}$, also das finale Tableau, und

$$L = \begin{pmatrix} 1 & & & & 0 \\ l_1^{(0)} & 1 & & & \\ l_2^{(0)} & l_2^{(1)} & 1 & & \\ \vdots & & \ddots & \ddots & \\ l_n^{(0)} & \ldots & \ldots & l_n^{(n-1)} & 1 \end{pmatrix} \tag{2.18}$$

die Matrix der Updatekoeffizienten aus (2.10). Dann sind R und L genau die die LR-Zerlegung von A.

Wie bereits gesagt, ist die Voraussetzung, dass die Gaußelimination mit kanonischer Pivotwahl durchgeführt werden kann, stark, und schließt selbst einfache Matrizen wie $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ aus. Wie man sich leicht überlegt, besitzt diese Matrix allerdings keine LR-Zerlegung.

Für manche Anwendungen ist es günstiger, wenn L und R normiert sind. Dann benutzt man die LDU-Zerlegung $A = LDU$, mit L linker unterer Dreiecksmatrix, D Diagonalmatrix und R rechte obere Dreiecksmatrix, wobei jetzt $l_{ii} = 1$ und $r_{ii} = 1$ sein soll. Die LDU-Zerlegung ergibt sich aus der LR-Zerlegung durch $d_{ii} = r_{ii}$ und $u_{ik} = r_{ik}/r_{ii}$.

2.5 Bandmatrizen

Im folgenden werden wir oft mit k-Bandmatrizen zu tun haben, also Matrizen, bei denen nur die Diagonale und einige Nebendiagonalen besetzt sind. Diagonalmatrizen sind also 1-Bandmatrizen, eine *Dreibandmatrix* hat die Form

$$\begin{pmatrix} d_1 & t_1 & & & 0 \\ b_1 & d_2 & t_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & d_{n-1} & t_{n-1} \\ 0 & & & b_{n-1} & d_n \end{pmatrix}. \tag{2.19}$$

Für Matrizen dieser Form ist die Gaußelimination mit kanonischer Pivotwahl sehr effizient, da pro Iteration jeweils nur die erste Zeile des Resttableaus verändert werden muss. Dadurch reduziert sich der Rechenaufwand auf $\mathcal{O}(n^2)$. Die resultierenden L und R der LR-Zerlegung sind ebenfalls (Drei-)Bandmatrizen, wobei L nur auf der Haupt und der unteren Nebendiagonalen von Null verschiedene Einträge hat, R nur auf der Diagonalen und der Nebendiagonalen oberhalb.

SciPy stellt für Bandmatrizen ebenfalls spezielle Löserroutinen zur Verfügung, `scipy.linalg.solve_banded((l,u), A, b)`, wobei `l` und `u` die Anzahl der Nebendiagonalen oberhalb und unterhalb angeben, und `A` die Matrix in Bandform angibt.

2.6 Cholesky-Zerlegung

Wir betrachten im folgenden nur symmetrische, positiv definite Matrizen, wie sie gerade in der Physik oft vorkommen. Auch in der Optimierung spielen diese eine wichtige Rolle. Sei $A = LDU$ eine LDU-Zerlegung einer symmetrischen Matrix, dann gilt

$$LDU = A = A^T = (LDU)^T = U^T D L^T. \tag{2.20}$$

Da die LDU-Zerlegung aber eindeutig ist und U^T eine normierte, linke untere Dreiecksmatrix und L^T eine normierte, rechte obere Dreiecksmatrix, so gilt $U = U^T$, und damit

$$A = U^T D U = \widehat{U}^T \widehat{U} \quad \text{mit } \widehat{U} = \operatorname{diag}(\sqrt{d_{ii}})U. \tag{2.21}$$

Dies ist die Cholesky-Zerlegung. Anstatt die Gaußelimination durchzuführen, lässt sich die Zerlegung aber auch direkt mit Hilfe des *Cholesky-Verfahrens* bestimmen: Sei $A = \widehat{R}^T \widehat{R}$ eine Cholesky-Zerlegung. Da $\widehat{R}$ unterhalb der Diagonalen nur 0 enthält, gilt

$$a_{ik} = \sum_{l=1}^{i} \hat{r}_{li} \hat{r}_{lk} \quad \text{für } i = 1(1)n, k = 1(1)n. \tag{2.22}$$

Daraus lässt sich die erste Zeile von $\widehat{R}$ direkt ablesen:

$$\hat{r}_{11} = \sqrt{a_{11}} \quad \text{und } \hat{r}_{1k} = \frac{a_{1k}}{\hat{r}_{11}} \quad \text{für } k = 2(1)n. \tag{2.23}$$

Die nächsten Zeilen lassen sich analog bestimmen, da für jedes i

$$a_{ii} = \sum_{l=1}^{i} \hat{r}_{li}^2 \quad \Longrightarrow \quad \hat{r}_{ii}^2 = \sqrt{a_{ii} - \sum_{l=1}^{i-1} \hat{r}_{li}^2}. \tag{2.24}$$

Für die restlichen Elemente der Zeile gilt

$$\hat{r}_{ik} = \frac{1}{\hat{r}_{ii}} \left(a_{ik} - \sum_{l=1}^{i-1} \hat{r}_{li} \hat{r}_{lk} \right) \quad \text{für } k = i+1(1)n \tag{2.25}$$

Das Cholesky-Verfahren ist wie die Gaußelimination von der Ordnung $\mathcal{O}(n^3)$, braucht aber nur halb so viele Operationen. In SciPy ist die Cholesky-Zerlegung als **`scipy.linalg.cholesky(A)`** implementiert.

3 Darstellung von Funktionen

Die Lösung $\alpha(t)$ der Pendelbahn im einleitenden Beispiel hatten wir diskretisiert gespeichert, also als Vektor von Funktionswerten $\alpha(t_n)$ an bestimmten Zeitpunkten t_n. Für die graphische Darstellung reicht das, aber um zum Beispiel die Sinusfunktion mit wenigstens sechs Stellen Genauigkeit im Computer bereitzustellen, wären Millionen von Stützstellen nötig. Daher müssen bessere Darstellungen für Funktionen genutzt werden. Um die Funktion mit Computer auswerten zu können, muss sie aber letztlich auf (stückweise definierte) Polynome zurückgeführt werden, da auch moderne Prozessoren nur die Grundrechenarten beherrschen. Alle komplizierteren Funktionen, auch die Sinusfunktion, werden durch die Grundrechenarten dargestellt, auch wenn dies bei modernen Prozessoren in Hardware geschieht.

3.1 Horner-Schema

Die naive Auswertung eines Polynoms mit n Termen benötigt $n-1$ Additionen und $2(n-1)$ Multiplikationen sowie einen Zwischenspeicher für die Potenzen x^i des Arguments x. Besser ist die Auswertung nach dem Horner-Schema:

$$\sum_{i=0}^{n} c_i x^i = c_0 + x(c_1 + x(c_2 + x(\ldots(c_{n-1} + xc_n)))). \tag{3.1}$$

Wird dieser Ausdruck von rechts nach links ausgewertet, so muss das Ergebnis in jedem Schritt nur mit x multipliziert und der nächste Koeffizient addiert werden, was nur $n-1$ Multiplikationen und Additionen benötigt. Auch muss kein Zwischenwert gespeichert werden, was Prozessorregister spart. Als C-Code sieht die Auswertung des Hornerschemas so aus:

```
double horner(double *series, int n, double x)
{
    double r = c[n-1];
    for(int i = n-2; i >= 0; --i)
        r = r*x + c[i];
    return r;
}
```

Die Polynomauswertung stellt NumPy als **`numpy.polynomial.polyval(x, c)`** zur Verfügung. `c` bezeichnet die Koeffizienten des Polynoms und `x` das Argument, für das das Polynom ausgewertet werden soll.

Eine weitere Anwendung des Hornerschemas ist die Polynomdivision durch lineare Polynome der Form $x - x_0$, die zum beispiel wichtig für die iterative Bestimmung von

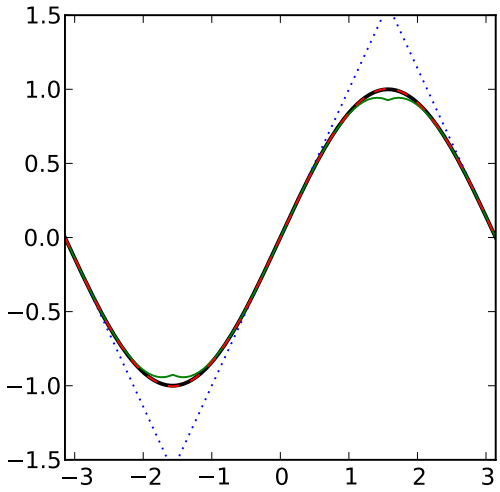


Abbildung 3.1: Näherung der Sinusfunktion durch die abgeschnittene Taylorreihe. Als schwarze durchgezogene Linie ist die tatsächliche Sinusfunktion dargestellt, blau gepunktet ist die Näherung erster Ordnung um Null, x, grün durchgezogen ist die kubische Näherung $x - x^3/6$, und rot gestrichelt $x - x^3/6 + x^5/120$. Die Kurven nutzen die Symmetrie der Sinuskurve, sind also an $\pm\pi/2$ gespiegelt.

Nullstellen ist. Wie man sich leicht durch Induktion überlegt, gilt für die Zwischenterme $d_i = c_i + x_0(c_{i+1} + x_0(\ldots(c_{n-1} + x_o c_n)))$, die sich bei Auswertung an der Stelle x_0 ergeben:

$$\sum_{i=0}^{n} c_i x^i = \sum_{i=0}^{n-1} d_{i+1} x^i (x - x_0) + d_0, \tag{3.2}$$

d.h., die ersten n Terme d_i liefern das Ergebnispolynom, und d_0 den Rest.

3.2 Taylorreihen

Nachdem wir nun wissen, wie Polynome effizient ausgewertet werden können, stellt sich die Frage, wie man ein gutes Näherungspolynom für eine Funktion bekommt. Dazu gibt es viele verschiedene Ansätze, deren Vor- und Nachteile im Folgenden kurz besprochen werden. Der älteste Ansatz, der auch in der Analytik weiten Einsatz findet, ist die Taylorentwicklung. Ist eine Funktion f um einen Punkt x_o hinreichend gut differenzierbar, lässt sie sich als bekannterweise lokal als Taylorreihe darstellen:

$$f(x) = \sum_{i=0}^{\infty} \frac{f^{(i)}(x_0)}{i!} (x - x_0)^i, \tag{3.3}$$

wobei $f^{(i)}(x)$ die i-te Ableitung von f an der Stelle x bezeichnet. Falls die Ableitungen existieren und $x - x_0$ klein genug ist, so konvergiert diese Darstellung schnell, und einige

Terme genügen, um zufriedenstellende Genauigkeit zu erreichen. Lokal um den Entwicklungspunkt x_0 is eine abgeschnittene Taylorreihe also eine gute polynomielle Näherung, allerdings gibt es für die meisten Funktionen einen Konvergenzradius, außerhalb dessen die Reihe nicht einmal konvergiert. Daher eignen sich Taylorreihen vor allem gut für kleine Umgebungen. Auch ist eine abgeschnittene Taylorreihe ist nur Entwicklungspunkt x_0 exakt; dort stimmen allerdings gleich die ersten i Ableitungen.

Um zum Beispiel die oben angeführte Sinusfunktion mit 7 Stellen Genauigkeit im Interval $[0 : \pi/2]$ auszuwerten, genügen die ersten 7 Terme der Taylorreihe. Mit Hilfe der Symmetrien der Funktion lässt sie sich damit bereits für alle Argumente auswerten. Da

$$\sin'(x) = \cos(x) \quad \text{und} \quad \cos'(x) = -\sin(x),$$

ergibt sich die bekannte Reihe

$$\sin(x) = \sum_{i=0}^{\infty} \frac{\sin^{(i)}(0)}{i!} x^i = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1}. \tag{3.4}$$

Wie gut diese Darstellung mit entsprechender Rückfaltung funktioniert, zeigt Abbildung 3.1. Für viele andere komplexe Funktionen ist es ebenfalls möglich, Taylorreihen analytisch oder numerisch zu bestimmen, die dann zur Auswertung auf dem Computer genutzt werden können.

3.3 Polynom- oder Lagrangeinterpolation

Wie besprochen ist eine abgeschnittene Taylorreihe exakt nur im Entwicklungspunkt, dafür allerdings gleich mit den ersten paar Ableitungen. Oft sind die Ableitungen der Funktion aber gar nicht relevant, dafür würde man lieber an mehr Punkten eine exakte Darstellung haben. Wie sich zeigt, gibt es dann genau ein Polynom, dass die Funktion an diesen Punkten exakt interpoliert. Genauer: seien Punkte (x_i, y_i), $i = 0(1)n-1$ gegeben mit x_i paarweise verschieden. Dann gibt es genau ein Polynom $P(x) = \sum_{k=0}^{n-1} a_k x^k$ vom Grad $n-1$, so dass $P(x_i) = y_i$, da die Gleichung

$$\begin{aligned} y_1 &= P(x_0) = a_0 + a_1 x_0 + \cdots + a_{n-1} x_0^{n-1} \\ &\vdots \\ y_n &= P(x_{n-1}) = a_0 + a_1 x_{n-1} + \cdots + a_{n-1} x_{n-1}^{n-1} \end{aligned} \tag{3.5}$$

genau eine Lösung hat. Die Punkte x_i werden auch *Stützstellen* genannt, da an diesen Punkten das Polynom genau die vorgegebenen Werte y_i annimmt. Allerdings ist nicht gewährleistet, dass mit steigender Anzahl von Punkten die Funktion auch zwischen den Stützstellen immer besser angenähert wird. Tatsächlich hat Runge ein einfaches Beispiel angegeben, nämlich die Rungefunktion $1/(1+x^2)$, für die die Näherung mit steigender Anzahl an äquidistanten Punkten immer schlechter wird, siehe Abbilding 3.2. In SciPy liefert die Funktion **`scipy.interpolate.lagrange(x, y)`** das interpolierende Polynom durch die Stützstellen (`x[i]`, `y[i]`).

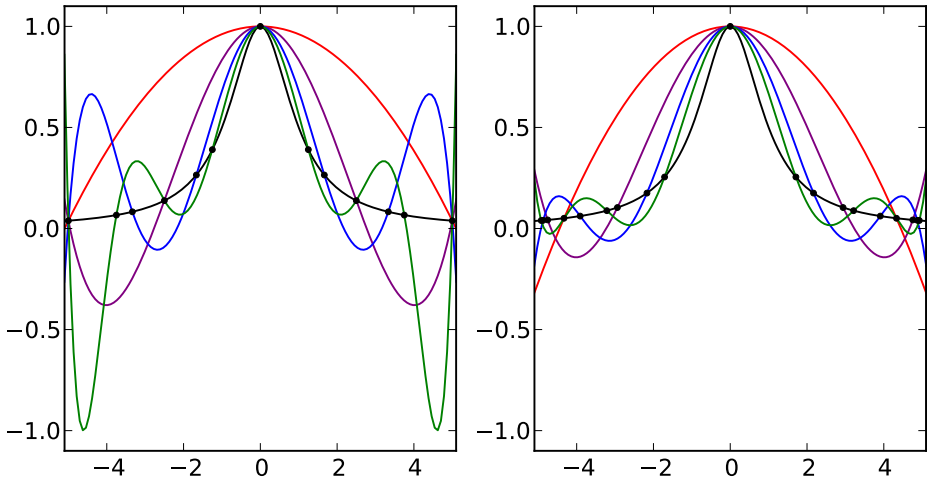


Abbildung 3.2: Lagrange-Interpolation der Rungefunktion $1/(1+x^2)$ (schwarze Linie). Im linken Graph sind die Stützstellen äquidistant gewählt (markierte Punkte), die farbigen Linien sind die interpolierenden Polynome der Ordnungen 2 (rot), 4 (lila), 6 (blau) und 8 (rot). Mit steigender Ordnung wird die Interpolation am Rand immer schlechter. Im rechten Graph sind für die gleichen Ordnungen Chebyshev-Stützstellen gewählt worden, um den Fehler zu minimieren.

Bei der etwas allgemeineren Hermite-Interpolation können an den Stützstellen neben den Funktionswerten auch Ableitungen vorgegeben werden. Das eindeutige interpolierende Polynom hat dann einen Grad, der der Gesamtanzahl an vorgegebenen Funktionswerten und Ableitungen entspricht. Ist zum Beispiel nur eine Stützstelle x_0 gegeben und neben dem Funktionswert n Ableitungen, so entspricht das Hermite-Polynom genau den ersten $n+1$ Termen der Taylorreihe.

Das interpolierende Polynom kann nicht nur zur Interpolation verwendet werden, also der Bestimmung an Punkten zwischen den Stützstellen, sondern — mit Vorsicht — auch zur Extrapolation, also um Werte außerhalb des Bereichs zu bestimmen. Da bei der Hermite-Interpolation auch die Ableitungen insbesondere am Rand kontrolliert werden können, ist diese hier tendenziell vorteilhafter. Extrapolation spielt eine wichtige Rolle, wenn eine direkte Auswertung der Zielfunktion numerisch zu teuer oder unmöglich wird. Bei Computersimulation tritt dies insbesondere in der Nähe von kritischen Punkten auf.

3.3.1 Lagrangepolynome

Die Koeffiziente a_i können im Prinzip als Lösung von Gleichung (3.5) mit geeigneten Lösern für lineare Gleichungssysteme gefunden werden, was im allgemeinen allerdings recht langsam ist. Daher benutzt man besser eine direkte Darstellung mit Hilfe der *La-*

grangepolynome, die wie folgt definiert sind:

$$L_i(x) = \prod_{k \neq i} \frac{x - x_k}{x_i - x_k}. \tag{3.6}$$

Die Polynominterpolation wird daher auch Lagrange-Interpolation genannt. Wie man leicht sieht, gilt $L_i(x_k) = \delta_{ik}$, so dass das Polynom

$$P(x) = \sum_{i=1}^{n} y_i \, L_i(x) \tag{3.7}$$

das eindeutige interpolierende Polynom durch (x_i, y_i) ist. Diese Darstellung ist allerdings für praktische Zwecke nur sinnvoll, wenn sich die Stützstellen x_i nicht ändern, da die Bestimmung der Lagrangepolynome $L_i(x)$ zeitaufwändig ist. Geeigneter ist die *baryzentrische Darstellung*

$$P(x) = \sum_{i=0}^{n-1} y_i \, \mu_i \Big/ \sum_{i=0}^{n-1} \mu_i \quad \text{mit} \quad \mu_i := \frac{1}{x - x_i} \prod_{k \neq i} \frac{1}{x_i - x_k}, \tag{3.8}$$

bei der lediglich der Quotient zweier rationaler Funktionen gebildet werden muss.

3.3.2 Neville-Aitken-Schema

Das rekursive Neville-Schema ist eine effiziente Möglichkeit, das interpolierende Polynom auszuwerten ohne es tatsächlich zu berechnen. Das ist nützlich, wenn nur wenige Auswertungen nötig sind, wie zum Beispiel beim Romberg-Integrationsverfahren, bei dem zur Schrittweite 0 extrapoliert wird.

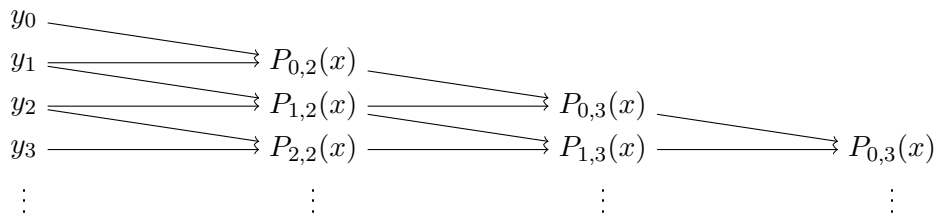
Wir definieren $P_{i,k}$ als das interpolierende Polynom der Ordnung $k-1$ durch die Stützstellen $x_j, j = i(1)i+k-1$. Gesucht ist der Wert $P(x) = P_{0,n}(x)$ des interpolierenden Polynoms an der Stelle x. Dann ist

$$P_{i,1}(x) = y_i \quad \text{für } i = 0(1)n-1 \tag{3.9}$$

und

$$P_{i,k}(x) = \frac{P_{i,k-1}(x)(x_{i+k-1} - x) + P_{i+1,k-1}(x)(x - x_i)}{x_{i+k-1} - x_i} \quad \text{für } k = 2(1)n, i = 0(1)n-k, \tag{3.10}$$

da ja an den inneren Stützstellen x_l, $l = i+1(1)i+k-2$, $P_{i,k-1}(x_l) = P_{i+1,k-1}(x_l) = y_l$ gilt, und per Konstruktion $P_{i,k}(x_i) = y_i$ und $P_{i,k}(x_{i+k-1}) = y_{i+k-1}$. Durch sukzessives Berechnen zunächst der $P_{i,2}(x)$, dann der $P_{i,3}(x)$, usw. lässt sich das interpolierende Polynom bequem an einer fixen Stelle auswerten. Als (Neville-)Schema sieht das so aus:



wobei die Pfeilpaare dividierte Differenzen gemäß (3.10) bedeuten.

3.3.3 Newtonsche Darstellung

Wir betrachten nun die Polynome $P_{0,k}$ des Nevilleschemas. Es gilt offenbar

$$P_{0,k}(x) - P_{0,k-1}(x) = \gamma_k(x - x_0)\cdots(x - x_{k-2}), \tag{3.11}$$

da die beiden Polynome in den Stützstellen $x_0, \ldots, x_{k-2}$ übereinstimmen und die Differenz ein Polynom vom Grad $k-1$ ist, also höchstens $k-1$ Nullstellen hat. Weiter ist γ_k der führende Koeffizient des Polynoms $P_{0,k}(t)$, da $P_{0,k-1}(t)$ ja einen niedrigeren Grad hat. Daraus ergibt sich die folgende *Newtonsche Darstellung* des interpolierenden Polynoms:

$$\begin{aligned} P_{0,n}(x) &= y_0 + \sum_{k=2}^{n} P_{0,k}(x) - P_{0,k-1}(x) \\ &= y_0 + \gamma_2(x - x_0) + \gamma_3(x - x_0)(x - x_1) + \cdots + \gamma_n(x - x_0)\cdots(x - x_{n-2}) \\ &= y_0 + (x - x_0)\Big(\gamma_2 + (x - x_1)\Big(\gamma_3 + \cdots\big(\gamma_{n-1} + (x - x_{n-2})\gamma_n\big)\cdots\Big)\Big). \end{aligned} \tag{3.12}$$

Die letztere Umformung zeigt, dass sich die Newtonsche Darstellung effizient mit einem leicht abgewandelten Hornerschema auswerten lässt:

```
def horner(x0, x, gamma):
    r = 0
    for k in range(len(x)-1, -1, -1):
        r = r*(x0-x[k]) + gamma[k];
    return r
```

Die Koeffizienten γ_i, $i = 2(1)n$ lassen sich dabei bequem mit dem Nevilleschema bestimmen. γ_k ist ja der höchste Koeffizient von $P_{0,k}$ ist, der sich leicht aus (3.10) berechnen lässt. Wenn $\gamma_{i,k}$ den führenden Koeffizienten des Polynoms $P_{i,k}$ bezeichnet, so erhalten wir das Nevilleschema

$$\gamma_{i,1} = y_i \quad \text{für } i = 0(1)n-1 \quad \text{und} \tag{3.13}$$

$$\gamma_{i,k} = \frac{\gamma_{i+1,k-1} - \gamma_{i,k-1}}{x_{i+k-1} - x_i} \quad \text{für } k = 2(1)n, i = 0(1)n-k. \tag{3.14}$$

Da letztlich nur die $\gamma_{0,k}$ interessant sind, also die obere Diagonale des Nevilleschemas, benötigt man für die Berechnung nur einen Vektor

$$\gamma' = (\gamma_{0,1},\gamma_{0,2},\ldots,\gamma_{0,k-1},\gamma_{0,k},\gamma_{1,k},\ldots,\gamma_{n-k,k})\,, \tag{3.15}$$

der wie folgt berechnet wird:

```
def neville(x, y):
    n = len(x)
    gamma = y.copy()
    for k in range(1, n):
        for i in range(n-k-1, -1, -1):
            gamma[i+k] = (gamma[i+k] - gamma[i+k-1])/(x[i+k] - x[i])
    return gamma
```

Man beachte, dass die Schleife über i herunterläuft, um benötigte Werte nicht zu überschreiben.

3.3.4 Chebyshev-Stützstellen

Bis jetzt haben wir wenig zur Wahl der Stützstellen gesagt. Oft liegt es auch nahe, äquidistante Stützstellen zu verwenden wie im Fadenpendel-Beispiel. Man kann allerdings zeigen, dass die Chebychev-Stützstellen den Fehler der Polynominterpolation minimieren. Diese sind definiert als die Nullstellen der Polynome (!)

$$T_n(\cos\phi) = \cos(n\phi), \tag{3.16}$$

die offensichtlich zwischen -1 und 1 liegen und daher geeignet skaliert werden müssen für die Interpolation in einem allgemeinen Interval. Die Chebychev-Polynome T_n, $n \geq 0$, bilden eine orthogonale Basis der Funktionen über $[-1,1]$ bezüglich des mit $1/\sqrt{1-x^2}$ gewichteten Skalarprodukts. Daher kann jede genügend glatte Funktion auf $[-1:1]$ als eine Reihe

$$f(x) = \sum_{n=0}^{\infty} a_n T_n(x) \tag{3.17}$$

dargestellt werden, die sogenannte Chebyshev-Reihe (siehe auch z.B. Abramowitz und Stegun [AS70]).

Explizit sind diese Nullstellen gegeben durch

$$x_{k,n} = \cos\left(\frac{2k+1}{2n}\pi\right), \quad k = 0(1)n-1. \tag{3.18}$$

Wird die Rungefunktion mit Chebyshevstützstellen interpoliert, so konvergiert das interpolierende Polynom, im Gegensatz zu äquidistanten Stützstellen.

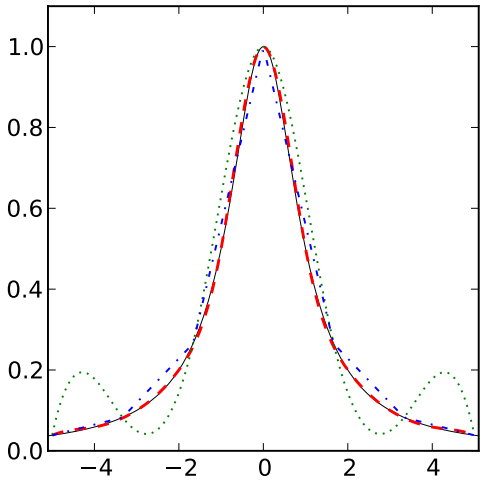


Abbildung 3.3: Spline-Interpolation der Rungefunktion (durchgezogene schwarze Linie). Die gestrichelte blaue Linie ist die lineare Spline-Interpolierende mit 7 Stützstellen, die anderen Kurven sind kubische Splines mit 7 (grün gestrichelt) und 11 Stützstellen (blaue Striche und Punkte).

3.4 Splines

Wie wir gesehen haben, kann unter ungünstigen Umständen die Güte der Polynominterpolation mit steigender Anzahl an Stützstellen sinken, vor allem, wenn diese äquidistant verteilt sind. Oft ist das aber nicht zu vermeiden, zum Beispiel, wenn die Daten in einem Experiment regelmäßig gemessen werden. Das Problem ist, das die Koeffizienten des Polynoms global gelten, sich glatte Funktionen aber nur lokal wie ein Polynom verhalten (Taylorentwicklung!). Daher ist es günstiger, statt der gesamten Funktion nur kleine Abschnitte durch Polynome zu nähern.

Der einfachste Fall einer solchen Näherung ist die *lineare Interpolation*, bei der die Stützstellen durch Geraden, also Polynome vom Grad 1, verbunden werden. Sind die Stützstellen (x_i, y_i), $i = 1(1)n$ gegeben, so ist der lineare interpolierende Spline

$$P_1(x) = \frac{(x_{i+1} - x)y_i + (x - x_i)y_{i+1}}{x_{i+1} - x_i} \quad \text{für } x_i \leq x < x_{i+1}. \tag{3.19}$$

Diese Funktionen sind aber an den Stützstellen im allgemeinen nicht differenzierbar. Soll die Interpolierende differenzierbar sein, müssen Polynome höherer Ordnung genutzt werden. Solche stückweise definierten Polynome heißen *Splines* — das englische Wort Splines bezeichnete dünne Latten, die vor dem Computerzeitalter benutzt wurden, um glatte, gebogene Oberflächen vor allem für Schiffsrümpfe zu entwickeln. Der wichtigste Spline ist der *kubische* oder *natürliche* Spline, der aus Polynomen dritten Grades zusam-

mengesetzt und zweifach stetig differenzierbar ist. Seine allgemeine Form ist

$$P_3(x) = y_i + m_i(x - x_i) + \frac{1}{2}M_i(x - x_i)^2 + \frac{1}{6}\alpha_i(t - t_i)^3 \quad \text{für } x_i \le x < x_{i+1}. \tag{3.20}$$

Da die zwei rechten und linken zweiten Ableitungen an den Stützstellen übereinstimmen müssen, gilt

$$\alpha_i = \frac{M_{i+1} - M_i}{x_{i+1} - x_i}. \tag{3.21}$$

Aus der Gleichheit der Funktionswerte an den Stützstellen ergibt sich

$$m_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{1}{6}(x_{i+1} - x_i)^2(2M_i + M_{i+1}). \tag{3.22}$$

Aus der Gleichheit der ersten Ableitungen ergibt sich schliesslich ein Gleichungssystem für die M_i. Hier kommen in den Gleichungen gleichzeitig M_{i-1}, M_i und M_{i+1} vor, daher müssen für die Randwerte weitere Vorgaben gemacht werden. Sollen die Splines am Rand festgelegte 2. Ableitungen M_0 und M_n haben, so hat das Gleichungssystem die Form

$$\begin{pmatrix} 2 & \lambda_1 & & & & 0 \\ \mu_2 & 2 & \lambda_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \mu_{n-2} & 2 & \lambda_{n-2} & \\ 0 & & & \mu_{n-1} & 2 & \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} 6S_1 - \mu_1 M_0 \\ 6S_2 \\ \vdots \\ 6S_{n-2} \\ 6S_{n-1} - \lambda_{n-1}M_n \end{pmatrix}, \tag{3.23}$$

mit

$$\lambda_i = \frac{x_i - x_{i-1}}{x_{i+1} - x_{i-1}}, \quad \mu_i = \frac{x_{i+1} - x_i}{x_{i+1} + x_{i-1}} \quad \text{und} \quad S_i = \frac{\frac{y_{i+1}-y_i}{x_{i+1}-x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}}}{x_{i+1} - x_{i-1}}. \tag{3.24}$$

Auch periodische Funktionen können kubisch interpoliert werden, wobei dann die zusätzliche Bedingungen durch die Kontinuität über die periodische Grenze hinweg gegeben sind. Die Gleichungen für α_i und m_i sind dabei unverändert, nur das Gleichungssystem wird

$$\begin{pmatrix} 2 & \lambda_1 & & & & \mu_1 \\ \mu_2 & 2 & \lambda_2 & & 0 & \\ & \ddots & \ddots & \ddots & & \\ & 0 & & \mu_{n-2} & 2 & \lambda_{n-2} \\ \lambda_{n-1} & & & & \mu_{n-1} & 2 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} 6S_1 \\ 6S_2 \\ \vdots \\ 6S_{n-2} \\ 6S_{n-1} \end{pmatrix}, \tag{3.25}$$

wobei die Funktion als $x_n - x_1$-periodisch mit $y_1 = y_n$ vorausgesetzt wird. Abbildung 3.3 zeigt die Spline-Interpolation der Rungefunktion, die für diesen Interpolationstyp keine Probleme zeigt.

Die Gleichungssysteme (3.23) und (3.25) sind sehr gut konditioniert und mit einem einfachen Gleichungslöser zu behandeln. Zum Beispiel ist die Gauß-Elimination ist für die hier auftretenden einfachen Bandstrukturen sehr effizient. In SciPy gibt es selbstverständlich bereits eine fertige Routine für die Spline-Interpolation, nämlich **`scipy.interpolate.interp1d(x, y, kind)`**. (`x`,`y`) sind dabei die Stützstellen, und `kind` eine Zeichenkette, die den Typ des Splines bestimmt. Mögliche Werte sind zum Beispiel „linear“ und „cubic“ für lineare bzw. kubische interpolierende Splines.

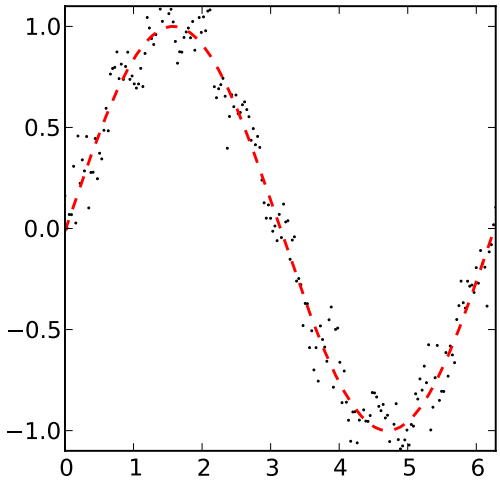


Abbildung 3.4: Methode der kleinsten Quadrate zum Fitten der Sinusfunktion $y = \sin(x)$. 200 Datenpunkte zwischen 0 und 2π wurden als $\sin(x) + 0{,}1\, sin(10x) + \xi$ erzeugt, wobei ξ eine Gauß-verteilte Pseudozufallsvariable mit Varianz 0,01 war. Die resultierende Sinusfunktion (rot gestrichelt) hat die Form $a \sin(bx + c)$, wobei die Koeffizienten auf gut 2% Genauigkeit $a = b = 1$ und $c = 0$ entsprechen. Die kleine höherfrequente Schwingung kann durch einen Fit allerdings nicht zuverlässig erkannt werden.

3.5 Ausgleichsrechnung, Methode der kleinsten Quadrate

Interpolierende Polynome, Taylorreihen und Splines haben gemeinsam, das diese exakt durch die gegebenen Stützstellen verlaufen. Oftmals ist das aber gar nicht gewünscht, da die Daten selber nicht exakt sind, zum Beispiel wenn diese aus einem Experiment oder einer Simulation stammen. In diesem Fall hat man üblicherweise eine Vorstellung, welche funktionelle Form die Daten annehmen, und möchte nun wissen, mit welchen Parametern diese Funktion am besten mit den Daten verträglich ist. Dazu muss man den Parametersatz bestimmen, so dass der Abstand der Daten von der Funktion minimiert wird.

Seien also wieder Daten (x_i, y_i), $i = 1(1)n$ und eine Funktion $f_v(x)$ gegeben. Gesucht ist dann derjenige Parametervektor v, der die Abweichung

$$\Delta(v) = \sum_i (f_v(x_i) - y_i)^2 \tag{3.26}$$

minimiert. Dieses Verfahren wird auch Methode der kleinste Quadrate genannt, da ja die quadrierten Abweichungen minimiert werden sollen. Ist $f_{a,b}(x) = ax + b$ eine Gerade, spricht man auch von linearer Regression. In diesem Fall lässt sich das Optimum einfach

bestimmen, da

$$0 = \frac{d}{da}\Delta(a,b) = \sum_i 2(ax_i + b - y_i)x_i = 2N\left(a\left\langle x_i^2\right\rangle + b\left\langle x_i\right\rangle - \left\langle y_i x_i\right\rangle\right) \tag{3.27}$$

und

$$0 = \frac{d}{db}\Delta(a,b) = \sum_i 2(ax_i + b - y_i) = 2N\left(a\left\langle x_i\right\rangle + b - \left\langle y_i\right\rangle\right), \tag{3.28}$$

wobei $\langle\cdot\rangle$ den Mittelwert über alle Datenpunkte bedeutet. Daraus ergibt sich

$$a = \frac{\left\langle y_i x_i\right\rangle - \left\langle y_i\right\rangle\left\langle x_i\right\rangle}{\left\langle x_i^2\right\rangle - \left\langle x_i\right\rangle^2} \quad \text{und } b = \left\langle y_i\right\rangle - a\left\langle x_i\right\rangle, \tag{3.29}$$

was sich einfach auf dem Computer berechnen lässt.

Auch für quadratische und andere einfache Funktionen lassen sich die Koeffizienten geschlossen darstellen, aber bei allgemeinen Funktionen ist dies nicht immer der Fall. Dann muss die nichtlineare Optimierungsaufgabe (3.26) numerisch gelöst werden, was wir später behandeln werden. Für den Moment genügt uns, dass SciPy die Funktion `scipy.optimize.leastsq(delta, v0, (x, y))` dafür bereitstellt. `(x, y)` sind dabei die Ausgangsdaten, die hier zu einem Tupel zusammengefasst sind. `v0` ist der Startwert für die Berechnung, der nicht zu weit vom (unbekannten) Optimum entfernt liegen darf. `delta` ist eine Python-Funktion, die als Argumente v, x_i und y_i nimmt und $f_v(x_i) - y_i$ zurückliefert. Da $f_v(x)$ eine beliebig komplizierte Form annehmen kann, ist diese Aufgaben im allgemeinen nicht lösbar, allerdings funktioniert ein solcher *Fit* für einfache Funktionen meistens recht gut. Abbildung 3.4 zeigt einen solchen Funktionsfit an eine verrauschte Sinusfunktion, die mit 200 Datenpunkten auf etwa 2% genau gefittet werden kann. Man beachte, das der Ausgangswert für den Fit mit Hilfe der SciPy-Funktion `leastsq` $a = 0$, $b = 1$, $c = 0$ war; beim Startwert $a = 0$, $b = 0$, $c = 0$ bricht das Verfahren ab. Das zeigt, dass man tatsächlich nicht zuweit vom Optimum starten kann, was ein gewisses Verständnis der Zielfunktion voraussetzt.

Ist die Funktionsform, die den Daten zugrundeliegt, unbekannt, ist es normalerweise keine gute Idee, die Form zu raten. Generell sollte auch die Anzahl der Parameter sehr klein sein, da sich sonst fast alles „gut“ fitten lässt („With four parameters I can fit an elephant and with five I can make him wiggle his trunk.“ — J. von Neumann).

Soll aber zum Beispiel für Visualisierungszwecke eine ansprechende Kurve entlang der Daten gelegt werden, deren tatsächliche Abhängigkeit unbekannt ist, dann sind *Padé-Funktionen* oft eine gute Wahl. Diese haben die Gestalt $P(x)/Q(x)$, wobei P und Q zwei Polynome mit paarweise verschiedenen Nullstellen sind. Üblicherweise lassen sich schon niedrigen Polynomgraden ansprechende Fits finden, sofern die Grade der beiden Polynome in etwa gleich gewählt werden.

3.6 Fourierreihen

Bis jetzt waren unsere Näherungsfunktionen auf Polynomen basierend, da diese einerseits vom Computer verarbeitet werden können und andererseits aufgrund der Taylorentwicklung glatte Funktionen meist gut approximieren. Für periodische Funktionen sind

Polynome aber an sich erst einmal wenig geeignet, da sie selber nicht periodisch sind. Splines können zwar auch periodisch gemacht werden, aber trotzdem sind trigonometrische Funktionen besser geeignet, um periodische Funktionen darzustellen. Fourierreihen und -transformationen stellen Funktionen als trigonometrische Reihen dar, die meist gut konvergieren und darüberhinaus einige nützliche Eigenschaften haben.

Es gibt zwei Hauptanwendungen der Fourierdarstellung: die Analyse und Aufbereitung periodischer Signale und die Lösung von Differentialgleichungen. Die Analyse des Spektrums einer Funktion gibt nützliche Informationen über die charakteristischen Zeitskalen, zum Beispiel die Tonhöhe und die Obertonreihe eines Instruments. In dieser Frequenzdarstellung lassen sich auch gezielt einzelne Frequenzen dämpfen, was Rauschen unterdrücken kann und im ursprünglichen Funktionsraum teure Faltungen erfordert. Bei Differentialgleichungen nutzt man aus, dass die Ableitung im Frequenzraum eine algebraische Operation ist, und die Differentialgleichung somit in eine gewöhnliche algebraische (und oft sogar lineare) übergeht.

3.6.1 Komplexe Fourierreihen

Wir betrachten eine periodische Funktion $f(t)$ mit $f(t+T) = f(t)$ für alle $t \in \mathbb{R}$, d.h. f hat Periode T. Dann ist die Fourierdarstellung von f gegeben durch

$$f(t) = \sum_{n \in \mathbb{Z}} \hat{f}_n e^{in\omega t} \tag{3.30}$$

mit der Grundfrequenz $\omega = 2\pi/T$. Die Koeffizienten $\hat{f}_n$ lassen sich berechnen als

$$\hat{f}_n = \frac{1}{T} \int_0^T f(t) e^{-in\omega t}\, dt$$

und sind im allgemeinen komplex, auch wenn f reellwertig ist. (3.30) lässt sich auch so lesen, dass

$$e^{-in\omega t} = \cos(n\omega t) + i \sin(n\omega t) \tag{3.31}$$

eine orthonormale Basis bezüglich des Skalarprodukts

$$(f, g) = \frac{1}{T} \int_0^T f(t)\overline{g(t)}\, dt \tag{3.32}$$

bilden. Ähnlich wie ein Vektor im $\mathbb{R}^n$ wird die Funktion f also durch die Fouriertransformation in ihre Schwingungskomponenten zerlegt. Insbesondere sind die Fourierkoeffizienten linear in der Funktion, d.h.

$$\widehat{f + \lambda g}_n = \hat{f}_n + \lambda \hat{g}_n. \tag{3.33}$$

Die Voraussetzungen für die Konvergenz der Fourierreihe sind sehr schwach - solange die Funktion wenigstens quadratintegrabel ist, konvergiert die Fourierreihe fast überall, d.h.

$$\left\| f(t) - \sum_{n=-N}^{N} \hat{f}_n e^{in\omega t} \right\| \xrightarrow{N \to \infty} 0. \tag{3.34}$$

Daneben ist die Transformation $f \to \hat{f}$ eine *Isometrie*, genauer gilt das *Parsevaltheorem*

$$\sum_{n\in\mathbb{Z}} |\hat{f}_n|^2 = \frac{1}{\omega}\int_0^t |f(t)|^2\,dt. \tag{3.35}$$

Das Parsevaltheorem besagt auch, dass die Restbeiträge von großen n immer kleiner werden, so dass also eine abgeschnittene Fourierreihe eine Approximation an die gesuchte Funktion darstellt. Anders als abgeschnittene Taylorreihen, die nur in einer schmalen Umgebung um den Aufpunkt exakt sind, konvergiert die Fourierreihe gleichmäßig. Allerdings muss die abgeschnittene Fourierreihe im allgemeinen keinen einzigen Punkt mit der Zielfunktion gemeinsam haben, anders als Taylorreihen oder Splines.

Weiter gilt:

- Die Fourierreihe über einem Interval $[0,T)$ kann aus der Fourierreihe für das Interval $[0,2\pi)$ durch Streckung mit der Grundfrequenz ω berechnet werden:

$$\widehat{f(t)}_n = \frac{1}{T}\int_0^T f(t)e^{-in\omega t}\,dt = \frac{1}{2\pi}\int_0^{2\pi} f(t'/\omega)e^{-int'}\,dt', \tag{3.36}$$

- Die Beiträge zu $\pm n$ unterscheiden sich nur in ihrer Phase, haben aber dieselbe Frequenz. Durch geeignete Wahl der Koeffizienten $\hat{f}_n$ kann die Phase nach Belieben verschoben werden.

- Für die komplexe Konjugation gilt stets $\hat{\overline{f}}_n = \overline{\hat{f}_n}$, da die Fouriertransformation ja linear ist.

- Ist Funktion f symmetrisch, also $f(t) = f(-t) = f(T-t)$, so ist $\hat{f}_{-n} = \hat{f}_n$.

- Ist Funktion f ungerade, also $f(t) = -f(-t) = -f(T-t)$, so ist $\hat{f}_{-n} = -\hat{f}_n$.

- Ist Funktion f reellwertig, also $f(t) = \overline{f(t)}$, so ist $\hat{f}_{-n} = \overline{\hat{f}_n}$. Allerdings sind die Fourierkoeffizienten im allgemeinen komplex!

- Ist die komplexwertige Funktion $f(t) = g(t) + ih(t)$ mit g, h reellwertig, gilt also

$$\hat{f}_n + \hat{\overline{f}}_n = \widehat{2g}_n \quad \text{und } \hat{f}_n - \hat{\overline{f}}_n = \widehat{2ih}_n. \tag{3.37}$$

  Dies bedeutet, dass sich die Fourierreihen zweier reellwertiger Funktionen zusammen berechnen und anschließend wieder trennen lassen. Da die Berechnung der Fourierkoeffizienten sowieso komplex erfolgen muss, erspart dies bei numerischer Auswertung eine Transformation.

- Die Ableitung der Fourierreihe ist sehr einfach:

$$\frac{d}{dt}f(t) = \sum_{n\in\mathbb{Z}} \hat{f}_n in\omega e^{in\omega t} = \sum_{n\in\mathbb{Z}} \widehat{\left(\frac{df}{dt}\right)}_n e^{in\omega t} \quad \Longrightarrow \quad \widehat{\left(\frac{df}{dt}\right)}_n = in\omega\hat{f}_n. \tag{3.38}$$

3.6.2 Reelle Fourierreihen

Da die Fourieranalyse besonders zur Analyse und Bearbeitung von Messdaten genutzt wird, sind die Fourierreihen reellwertiger Funktionen besonders wichtig. Ist die Funktion f rellwertig, so ist

$$\begin{aligned}\hat{f}_n e^{in\omega t} + \hat{f}_{-n} e^{-in\omega t} = \hat{f}_n e^{in\omega t} + \overline{\hat{f}_{-n} e^{in\omega t}} &= 2\mathrm{Re}(\hat{f}_n e^{in\omega t}) \\ &= 2\mathrm{Re}(\hat{f}_n)\cos(n\omega t) - 2\mathrm{Im}(\hat{f}_n)\sin(n\omega t). \quad (3.39)\end{aligned}$$

Daraus folgt, dass sich die Fourierreihe auch komplett reellwertig schreiben lässt:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(n\omega t) + b_n \sin(n\omega t) \quad (3.40)$$

mit

$$a_n = 2\mathrm{Re}(\hat{f}_n) = \frac{2}{T}\int_0^T f(t)\cos(n\omega t)\,dt \quad (3.41)$$

und

$$b_n = -2\mathrm{Im}(\hat{f}_n) = \frac{2}{T}\int_0^T f(t)\sin(n\omega t)\,dt. \quad (3.42)$$

Für symmetrische Funktionen ist offenbar $b_n = 0$, für ungerade Funktionen $a_n = 0$.

Einige reelle Fourierreihen sind zum Beispiel:

- Konstante $f(t) = f_0$:
$$a_0 = 2f_0, \quad a_n, b_n = 0 \quad \text{sonst} \quad (3.43)$$

- Rechteckfunktion
$$f(t) = \begin{cases} 1 & \text{für } 0 \le t < \frac{T}{2} \\ -1 & \text{für } \frac{T}{2} \le t < T \end{cases} = \frac{4}{\pi}\sum_{n=1}^{\infty} \frac{1}{2n-1}\sin\left((2n-1)\omega t\right) \quad (3.44)$$

- kurzer Rechteckpuls. Wir betrachten nun die auf konstanten Flächeninhalt normierte Funktion
$$f_S(t) = \begin{cases} 1/S & \text{für } 0 \le t < S \\ 0 & \text{für } S \le t < T \end{cases}, \quad (3.45)$$
deren Fourierreihe
$$f_s(t) = \frac{1}{2\pi} + \frac{2}{T}\sum_{n=1}^{\infty} \frac{\sin(n\omega S)}{n\omega S}\cos(n\omega t) + \frac{1-\cos(n\omega S)}{n\omega S}\sin(n\omega t) \quad (3.46)$$

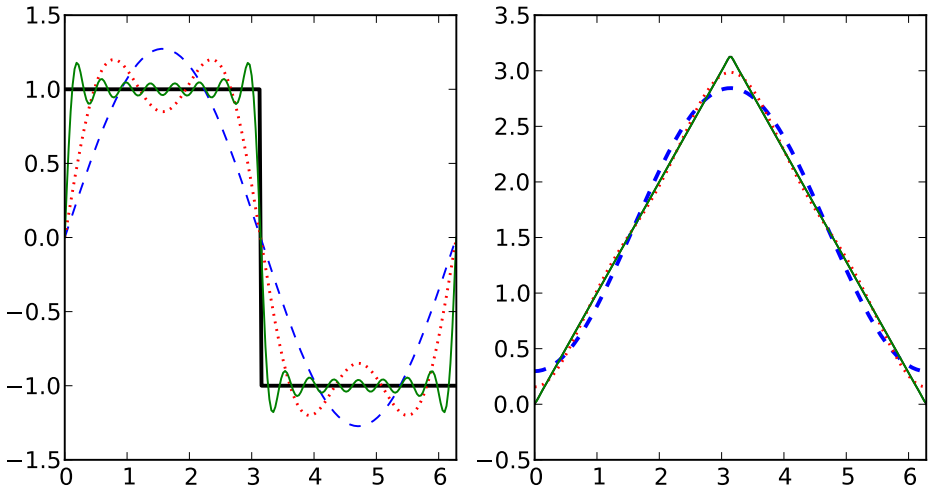


Abbildung 3.5: Abgeschnittene Fourierreihen der Rechteckfunktion (links) und eines Dreieckpulses (rechts). Die Funktionen sind jeweils als schwarze durchgezogene Linien eingezeichnet, die Näherungen mit einem Term blau gestrichelt, mit zwei Termen rot gepunktet, und mit 20 Termen grün durchgezogen. Für den Rechteckpuls ist letztere Näherung nicht mehr von der Funktion zu unterscheiden.

ist. Je kleiner S wird und damit der Träger von f_S, desto langsamer konvergiert ihre Fourierreihe, da die Funktion $\sin(x)/x$ immer dichter an der Null ausgewertet wird. Für jede feste Frequenz n gilt schließlich

$$\widehat{(f_S)}_n \xrightarrow{S\to 0} 1 = \hat{\delta}_n \quad \text{für alle } n \in \mathbb{Z} \tag{3.47}$$

bzw. $a_n \to 1$ und $b_n \to 0$. Die δ-Funktion, die ja der formale Grenzwert der f_S ist, , die ja den kleinstmöglichen Träger hat, hat also in gewisser Weise die am schlechtesten (tatsächlich gar nicht!) konvergierende Fourierreihe.

- Dreiecksfunktion

$$f(t) = \begin{cases} t & \text{für } 0 \le t < \frac{T}{2} \\ T - t & \text{für } \frac{T}{2} \le t < T \end{cases} \quad = \quad \frac{\pi}{2} - \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{1}{(2n-1)^2} \cos\left((2n-1)\omega t\right) \tag{3.48}$$

Genau wie die komplexe Fourierreihe lässt sich natürlich auch die reelle Fourierreihe abschneiden, um Näherungen für Funktionen zu bekommen, vergleiche Abbildung 3.5. Es fällt auf, das die Fourierreihe besonders schlecht dort konvergieren, wo die Funktion nicht differenzierbar ist bzw. einen Sprung aufweist.

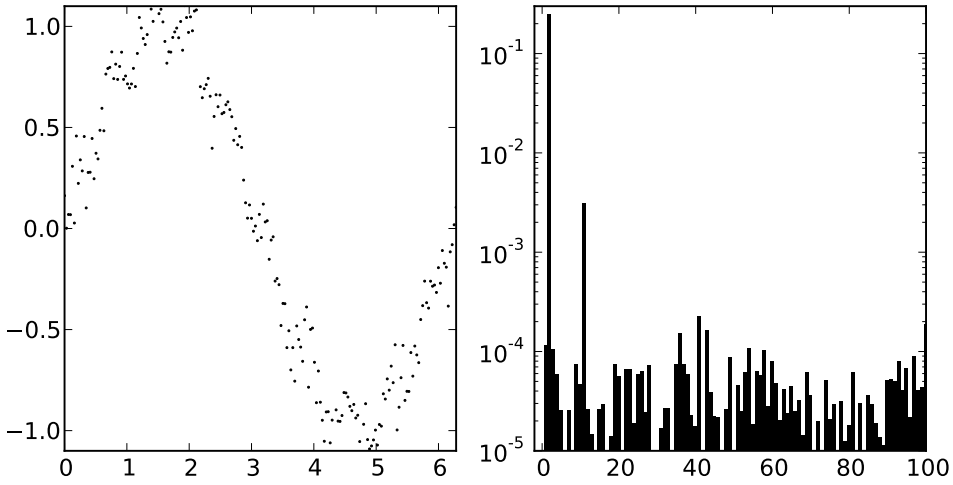


Abbildung 3.6: Diskrete Fouriertransformation von 200 diskreten Datenpunkten, die analog zu Abbildung 3.4 zwischen 0 und 2π als $\sin(x) + 0{,}1\,sin(10x) + \xi$ erzeugt wurden. Anders als der Funktionsfit erlaubt die DFT, auch die kleine zusätzliche Schwingung gut vom Rauschen zu unterscheiden. Im Graphen ist das Leistungsspektrum $|DFT(k)|^2$ gezeigt. Man erkennt die Amplitudenquadrate 0,25 bei 1 und 0,01 bei 10, auch wenn letztere Frequenz durch das Rauschen etwas an Intensität verloren hat.

3.6.3 Diskrete Fouriertransformation

Bei praktischen Anwendungen sind die Integrale zur Bestimmung der Koeffiziente oft nicht analytisch lösbar, oder die Funktion ist von vorneherein nur an diskreten Punkten gegeben, etwa weil es sich um Messdaten handelt. In diesem Fall müssen die Integrale numerisch approximiert werden. Wir betrachten nun also nicht mehr eine kontinuierliche Funktion f, sondern Daten $f_k = f(t_k)$ mit $t_k = k\Delta$, $k = 0(1)N-1$ und $\Delta = \frac{T}{N}$. Dann ist

$$\hat{f}_n = \frac{1}{T}\int_0^T f(t)e^{-in\omega t}\,dt \approx \frac{\Delta}{T}\sum_{k=0}^{N-1} f(k\Delta)e^{-in\omega k\Delta} = \frac{1}{N}\sum_{k=0}^{N-1} f_k e^{-i\frac{2\pi}{N}nk} =: \frac{g_n}{N}. \tag{3.49}$$

Die Koeffizienten

$$\mathrm{DFT}(f_k)_n = g_n = \sum_{k=0}^{N-1} f_k e^{-i\frac{2\pi}{N}nk} \tag{3.50}$$

werden als die *diskrete Fouriertransformierte* bezeichnet, die sehr effizient berechnet werden kann, wie wir im folgenden sehen werden. Analog wird die *inverse diskrete Fouriertransformation*

$$\mathrm{iDFT}(g_n)_k = f(t_k) = \sum_{k=0}^{N-1} \frac{g_n}{N} e^{i\frac{2\pi}{N}nk} \tag{3.51}$$

definiert, die aus den Koeffizienten wieder die Funktion f an den diskreten Eingangspunkten t_k berechnet.

Die Koeffizienten sind offenbar periodisch, da

$$g_{n+N} = \sum_{k=0}^{N-1} f_k e^{-i\frac{2\pi}{N}(n+N)k} = \sum_{k=0}^{N-1} f_k e^{-i\frac{2\pi}{N}nk} \underbrace{e^{-2\pi ik}}_{=1} = g_n. \tag{3.52}$$

Insbesondere ist $g_{-k} = g_{N-k}$, und es gibt nur N echt verschiedene Koeffizienten. DFT-Bibliotheken speichern die Koeffizienten daher meist als Vektor $(g_0, \ldots, g_{N-1})$ bzw. $(g_0, \ldots, g_{N/2-1}, g_{-N/2}, \ldots, g_{-1})$. Ist f reell, so gilt noch dazu $g_{-k} = \overline{g_k}$, sodass lediglich $\lceil N/2 \rceil$ Koeffizienten wirklich verschieden sind. Allerdings sind diese im allgemeinen komplex, so dass die N reellen Freiheitsgrade der Eingangsfunktion erhalten bleiben. Abbildung 3.6 zeigt zum die DFT der Summe zweier verrauschter Sinusfunktionen, aus der die beiden Basisfrequenzen und deren Amplituden klar gegenüber dem Rauschen zu erkennen sind.

3.6.4 Schnelle Fouriertransformation

Die Berechnung der Fouriertransformierten nach (3.50) ist zwar möglich, aber ziemlich langsam — jeder der N Koeffizienten benötigt offenbar $\mathcal{O}(N)$ Operationen, so dass die DFT insgesamt $\mathcal{O}(N^2)$ Operationen braucht. Das limitiert für praktische Anwendungen N auf einige tausend, was für viele Anwendungen zu wenig ist. Die DFT konnte daher nur durch die *schnelle Fouriertransformation* (FFT) von *Cooley und Tukey* zu breiter Anwendung finden. Diese basiert auf der Beobachtung, dass für $N = 2M$

$$\mathrm{DFT}(f_k)_n = \sum_{k=0}^{M-1} f_{2k} e^{-i\frac{2\pi}{2M}n\,2k} + \sum_{k=0}^{M-1} f_{2k+1} e^{-i\frac{2\pi}{2M}n\,(2k+1)} \tag{3.53}$$

$$= \mathrm{DFT}(f_{2k})_n + e^{-i\frac{2\pi}{2M}n}\mathrm{DFT}(f_{2k+1})_n, \tag{3.54}$$

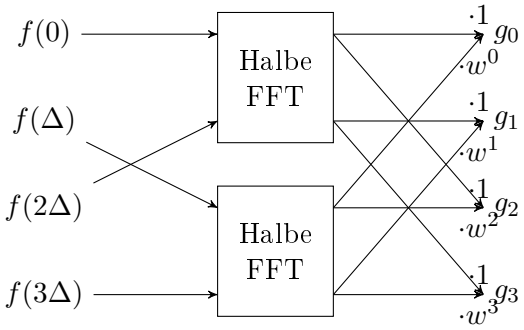
wobei $\mathrm{DFT}(f_{2k})_n$ den n-ten Koeffizienten einer DFT auf den Datenpunkten f_{2k}, $k = 0(1)M-1$, bezeichnet. Gemäß (3.52) ist dabei $\mathrm{DFT}(f_{2k})_n = \mathrm{DFT}(f_{2k})_{n-M}$ für $n > M$.

Die Fouriertransformatierte der N Datenpunkte ergibt sich also als einfache Summe von zwei Fouriertransformierten mit lediglich der halben Menge M an Datenpunkten, wobei die ungerade Hälfte mit der *Einheitswurzel*

$$w^n := e^{-i\frac{2\pi}{2M}n} \tag{3.55}$$

multipliziert wird. Ist nun M wieder durch zwei teilbar, so lassen sich diese Fouriertransformierten ebenfalls als Summe zweier nochmals halb so langer Fouriertransformationen darstellen. Wenn nun N eine Zweierpotenz ist, kann man so fortfahren, bis $M = 1$ erreicht ist, also $\mathrm{DFT}(f_0)_0 = f_0$. Dabei gibt es offenbar $\log_2(N)$ viele Unterteilungsschritte, die jeder $\mathcal{O}(N)$ Operationen kosten. Insgesamt benötigt die FFT also lediglich $\mathcal{O}(N \log N)$ Operationen.

Schematisch funktioniert ein FFT-Schritt wie folgt:



Aufgrund ihres Aussehens wird dieses Datenpfadschema auch als Butterfly-Schema genannt. Damit die beiden Unter-FFTs auf einem zusammenhängenden Satz von Daten operieren können, müssen also auch die Eingabedaten f_k umsortiert werden, ebenso wie auch für die Unter-FFTs. Man kann sich leicht überlegen, dass dabei f_k auf $f_{k'}$ sortiert wird, wobei die Bits k' in Binärdarstellung dieselben wie von k sind, nur in umgedrehter Reihenfolge.

Die FFT erlaubt also die effiziente Zerlegung einer Funktion in ihre Schwingungskomponenten, was viele wichtige Anwendungen nicht nur in der Physik hat. Daher gibt es eine Reihe sehr guter Implementierungen der FFT, allen voran die „Fastest Fourier Transform in the West“ (FFTW, `http://www.fftw.org`). Selbstverständlich bietet auch NumPy eine FFT, **`numpy.fft.fft(f_k)`**, mit der inversen FFT **`numpy.fft.ifft(g_n)`**. Die Routinen sind so implementiert, dass bis auf Maschinengenauigkeit $\mathrm{iFFT}(\mathrm{FFT}(f_k)) = f_k$.

Wichtige Anwendungsbeispiele der diskreten Fouriertransformation sind zum Beispiel die Datenformate JPEG, MPEG und MP3, die alle drei auf einer Abwandlung der DFT beruhen, der *diskreten Cosinustransformation* (DCT) für reelle Daten. Bei dieser wird der Datensatz so in der Zeitdomäne verdoppelt, dass er in jedem Fall eine gerade Funktion repräsentiert, wodurch die Fourierreihe in eine reine Cosinusreihe übergeht mit nur reellen Koeffizienten. Die DCT ist also eine Umwandlung reeller in reelle Zahlen. Wegen Ihrer Wichtigkeit gibt es nicht nur extrem effiziente Implementierungen für die meisten Prozessortypen, sondern auch spezielle Hardware.

3.7 Wavelets

Die Fouriertransformation wird vor allem deshalb für die Kompression von Audio- oder Bilddaten genutzt, weil sie hochfrequente von niederfrequenten Signalen trennt und die menschlichen Sinne die hochfrequenten Anteil meist nicht gut wahrnehmen können. Das ist allerdings nicht ganz korrekt, tatsächlich können wir nur stark lokale Änderungen nicht gut wahrnehmen. Dafür sind Fourierreihen an sich gar nicht so gut geeignet, da ja auch die hochfrequenten Schwingungen alles andere als lokal sind. Als Alternative hat sich die *Multiskalenanalyse* (MSA) oder *diskrete Wavelettransformation* etabliert, die auch im transformierten Raum lokal ist.

Anders als bei der Fouriertransformation, die eine Zerlegung in trigonometrische Funk-

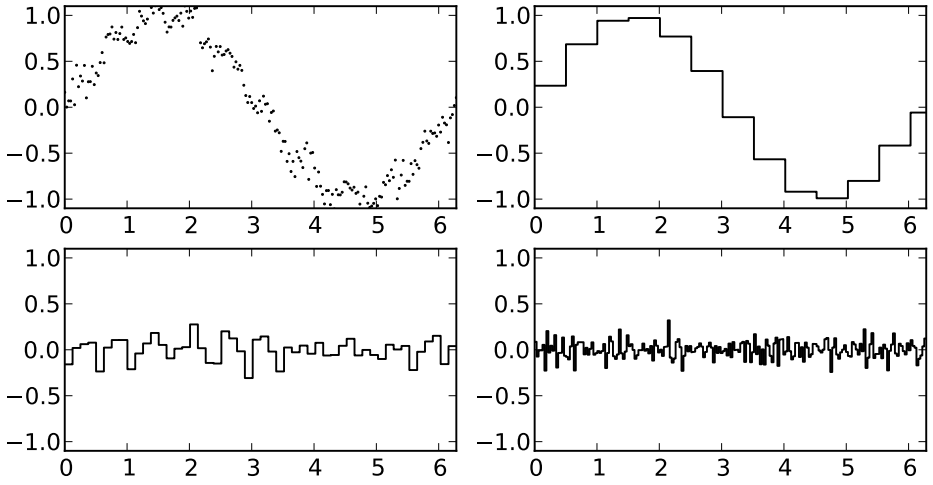


Abbildung 3.7: Diskrete Wavelettransformation von 200 diskreten Datenpunkten, die analog zu Abbildung 3.6 zwischen 0 und 2π als $\sin(x)+0{,}1\,sin(10x)+\xi$ erzeugt wurden. Für die Transformation wurden die Wavelets von (0,1] auf den Bereich (0,2π] gestreckt. Der linke obere Graph zeigt nochmals das Ausgangssignal, von rechts oben nach rechts unten folgen die Anteile der Stufen 0–3, also $(f,\phi_0)\phi_0 + \sum_{j=0}^{3}\sum_{k=0}^{2^j-1}(f,\psi_{jk})\psi_{jk}$, dann Stufen 4 und 5 ($\sum_{j=4}^{5}\sum_{k=0}^{2^j-1}(f,\psi_{jk})\psi_{jk}$) und schließlich Stufen 6–8, womit die Auflösung der Ausgangsdaten erreicht ist.

tionen darstellt, gibt es für die MSA verschiedene Sätze von Basisfunktionen mit verschiedenen Eigenschaften wie Differenzierbarkeit und Lokalität. Im folgenden soll die MSA mit Hilfe des Haar-Wavelets dargestellt werden, dass das einfachste und älteste bekannte Wavelet ist. Zunächste betrachten wir die *Skalierungsfunktion*

$$\phi(x) = \chi_{(0,1]} = \begin{cases} 1 & \text{für } 0 < x \leq 1 \\ 0 & \text{sonst} \end{cases} \tag{3.56}$$

sowie das Haar-*Wavelet*

$$\psi(x) = \begin{cases} -1 & \text{für } 0 < x \leq \frac{1}{2} \\ 1 & \text{für } \frac{1}{2} < x \leq 1 \\ 0 & \text{sonst,} \end{cases} \tag{3.57}$$

aus denen wir die Basisfunktionen $\phi_k(x) := \phi(x-k)$ der nullten Stufe und $\psi_{jk}(x) := 2^{j/2}\psi(2^j x - k)$ der j-ten Stufe konstruieren. Durch die Skalierung mit 2^j werden die $\psi_{j,k}$ also immer schmaler, sind aber wegen des Vorfaktors alle normiert, d.h. $\|\psi_{jk}\| = 1$. Ebenso sind auch die ϕ_k normiert. Zusätzlich sind sämtliche Basisfunktionen zu einander orthogonal, wie man sich leicht überlegt. Daher lässt sich jede quadratintegrable Funktion

f wie folgt zerlegen:

$$f(x) = \sum_{k\in\mathbb{Z}} (f,\phi_k)\phi_k + \sum_{j\in\mathbb{N}_0}\sum_{k\in\mathbb{Z}} (f,\psi_{jk})\psi_{jk} \tag{3.58}$$

Dies ist die Multiskalenanalyse von f. Die Koeffizienten der Stufe j werden auch Details der Stufe j genannt. In der Praxis ist das Signal durch endlich viele äquidistante Datenpunkte gegeben, analog zur diskreten Fouriertransformation. In diesem Fall sind die Summen endlich, da einerseits der Träger endlich ist und damit nur endlich viele $(f,\phi_k) \neq 0$, und es andererseits keine Details unterhalb der Auflösung des Signals gibt. Man skaliert dann die Wavelets und Skalierungsfunktion so, dass der Abstand der Datenpunkte gerade der halben Breite des Wavelets auf der feinsten Auflösung entspricht, und $\phi = \phi_0$ bereits das gesamte Interval überdeckt. Für eine nur auf [0,1] nichtverschwindende Funktion, deren Werte an 2^N Punkten äquidistanten Punkten bekannt ist, reduziert sich die Multiskalenanalyse zur *diskreten Wavelettransformation*

$$f(x) = (f,\phi)\phi + \sum_{j=0}^{N-1}\sum_{k=0}^{2^j-1} (f,\psi_{jk})\psi_{jk}. \tag{3.59}$$

Die Anzahl der Koeffizienten ist dann $1+1+2+\cdots+2^{N-1} = 2^N$, also genau die Anzahl der Eingabedaten. Genau wie die diskrete Fouriertransformation bildet die Wavelettransformation 2^N Werte $f(k/2^N)$ auf 2^N Werte (f,ϕ) und (f,ψ_{jk}) ab und besitzt eine exakte Rücktransformation, (3.59).

Analog zur schnellen Fouriertransformation gibt es auch eine schnelle Wavelettransformation, die sogar linearen Aufwand hat, also $\mathcal{O}(N)$ Schritte bei N Datenpunkten benötigt. Eine einfache Implementation der FWT und der inversen FWT für das Haar-Wavelet zeigt Codebeispiel 3.1. Der Kern dieser Transformation liegt darin, die Transformierte von der höchsten Detailauflösung herab aufzubauen, und dadurch die die Integrale approximierenden Summen schrittweise aufzubauen (*Downsampling*). Für genauere Informationen siehe zum Bespiel Daubechies [Dau92].

Abbildung 3.7 zeigt einige Detailstufen der Wavelet-Zerlegung der verrauschten Sinusfunktionen analog Abbildung 3.6. Auch hier lässt sich das Rauschen auf den höheren Detailstufen gut vom Nutzsignal trennen, allerdings kann die Oberschwingung nicht detektiert werden. Das hängt allerdings vor allem daran, dass das Haar-Wavelet nicht sehr geeignet ist, da es nicht glatt ist, im Gegensatz zum Nutzsignal. Daher sind in den meisten Fällen glatte Wavelets besser geeignet. Das bekannteste Beispiel von glatten Wavelets sind die Daubechies-Wavelets, die daneben auch einen kompakten Träger haben, also stark lokalisiert sind. Mit solchen Wavelets lassen sich sogar reale Musikdaten in Akkorde zurücktransformieren. Auch der JPEG-Nachfolger JPEG2000 basiert auf einer Wavelettransformation statt einer Cosinustransformation, allerdings mit Cohen-Daubechies-Feauveau-Wavelets.

```
# Haar-Wavelet-Transformation
############################
from scipy import *

def haar_trafo(data):
    "Diskrete Wavelettransformation mit Hilfe des Haar-Wavelets."
    # Daten mit kleinstem Integrationsschritt multiplizieren
    c = data.copy() / len(data)
    # Temporaerer Puffer, um benoetigte Werte nicht zu ueberschreiben
    ctmp = zeros(c.shape)
    width = len(c)/2
    while width >= 1:
        for n in range(width):
            tmp1 = c[2*n]
            tmp2 = c[2*n+1]
            # Detail
            ctmp[width + n] = tmp1 - tmp2
            # Downsampling
            ctmp[n]         = tmp1 + tmp2
        # Puffer zurueckschreiben
        c[:2*width] = ctmp[:2*width]
        width = width / 2
    return c

def inverse_haar_trafo(c):
    "Inverse Diskrete Wavelettransformation mit Hilfe des Haar-Wavelets"
    # Rueckgabewerte
    data = zeros(len(c))
    # phi mitnehmen auf der niedrigsten Stufe
    data[0] = c[0]
    width = 1
    cstart = 1
    while width <= len(c)/2:
        for n in range(width-1, -1, -1):
            tmp = data[n]
            data[2*n]     = tmp + width*c[cstart + n]
            data[2*n + 1] = tmp - width*c[cstart + n]
        cstart += width
        width = width * 2
    return data

# Anwendungsbeispiel

x = linspace(0,1,256)
y = cos(x)
coeff = haar_trafo(y)
yrueck = inverse_haar_trafo(coeff)
print max(abs(yrueck - y))
```

Listing 3.1: Diskrete Wavelettransformation und ihre Inverse als Python-Code. Die Länge der Eingabedaten muss eine Zweierpotenz 2^N sein. Die Details sind in einem Vektor c gespeichert, in der Form $c = ((f,\phi_0), (f,\psi_{00}), (f,\psi_{10}), (f,\psi_{11}), (f,\psi_{20}), (f,\psi_{21}), (f,\psi_{22}), \ldots, (f,\psi_{N-1,2^{N-1}-1}))$.

Literatur

[AS70] M. Abramowitz und I. Stegun. *Handbook of mathematical functions*. New York: Dover Publications Inc., 1970.

[Dau92] Ingrid Daubechies. *Ten lectures on wavelets*. Bd. 61. Society for Industrial Mathematics, 1992.

Index

A

Ausgleichsrechnung 32

B

Bandmatrizen 20

C

Chebyshev-Stützstellen 29
Cholesky
    -Verfahren 20
    -Zerlegung 20

D

DFT 38
Diagonalmatrizen 15
Dreibandmatrizen 20
Dreiecksmatrizen 15

F

Fadenpendel 7
FFT 39
Fitting 32
Fourierreihen
    komplexe 34
    reelle 36
Fouriertransformation
    diskrete 38
    komplexe 34
    reelle 36
    schnelle 39

G

Gaußelimination 16
Gleichungssysteme 15

H

Horner-Schema 23

I

Interpolation 25
    Lagrange- 25
    lineare 30
    Polynom- 25
    Spline- 30
interpolierendes Polynom
    baryzentrische Darstellung 27
    Lagrangedarstellung 27
    Newtonsche Darstellung 28

L

Lagrangepolynome 26
LDU-Zerlegung 20
lineare Regression 32
LR-Zerlegung 19

M

Matrixinversion 18
Methode der kleinsten Quadrate ... 32
Multiskalenanalyse 40

N

Neville-Aitken-Schema 27

P

Parsevaltheorem 35
Pivotwahl 18

S

Spline 30
    kubisch 30
    natürlich 30

T

Taylorreihe 24

W

Wavelets 40
Wavelettransformation 42