Skript zur Vorlesung "Physik auf dem Computer"

JP Dr. A. Arnold
Universität Stuttgart
Institut für Computerphysik

unter Mithilfe von
Dr. O. Lenz

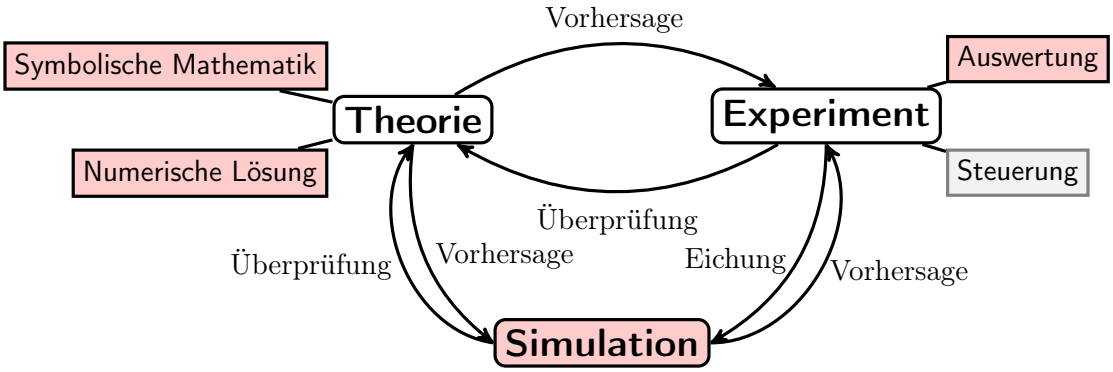
basierend auf Vorarbeiten von:
PD Dr. R. Hilfer, M. Lätzel, R. Mück, T. Ihle,
Prof. H. Herrmann, PD Dr. S. Luding, Dr. S. Schwarzer,
M. Müller, Dr. H.-G. Matuttis, M. Brunner, C. Manwart,
T. Karle, S. Manmana und C. Kunert

Sommersemester 2012

Inhaltsverzeichnis

1 Einleitung

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

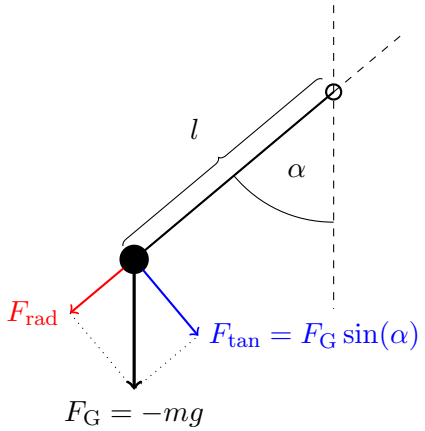


Abbildung 1.1: Schematisches Fadenpendel der Masse m, das an einem masselosen, steifen Faden der Länge l hängt.

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.

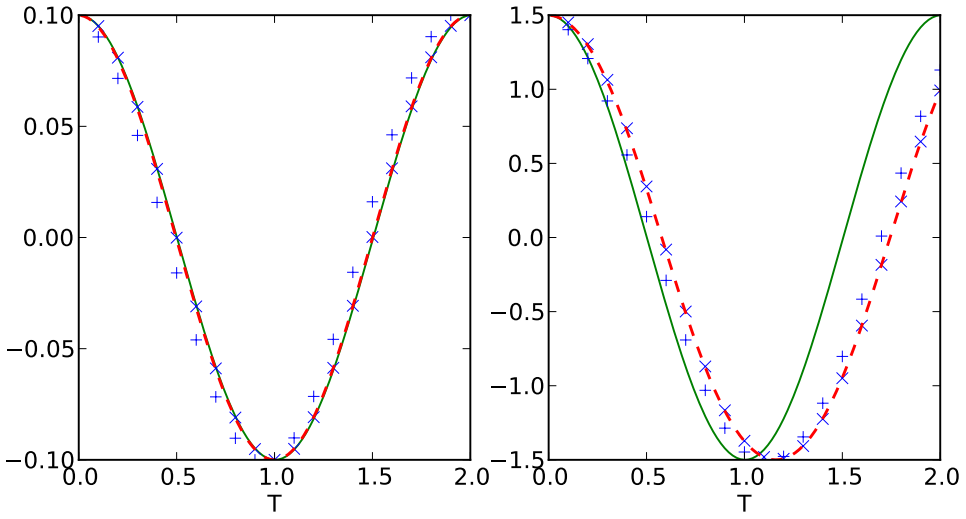


Abbildung 1.2: Lösungen für ein Fadenpendel der Länge $l = 1m$. Im linken Graphen ist die Ausgangslage $\alpha = 1{,}5$, im rechten $\alpha = 0{,}1$; in beiden Fällen ist die Ausgangsgeschwindigkeit 0. Die durchgezogene grüne Linie markiert die analytische Näherungslösung (1.4) für kleine Winkel. $\times$ 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.

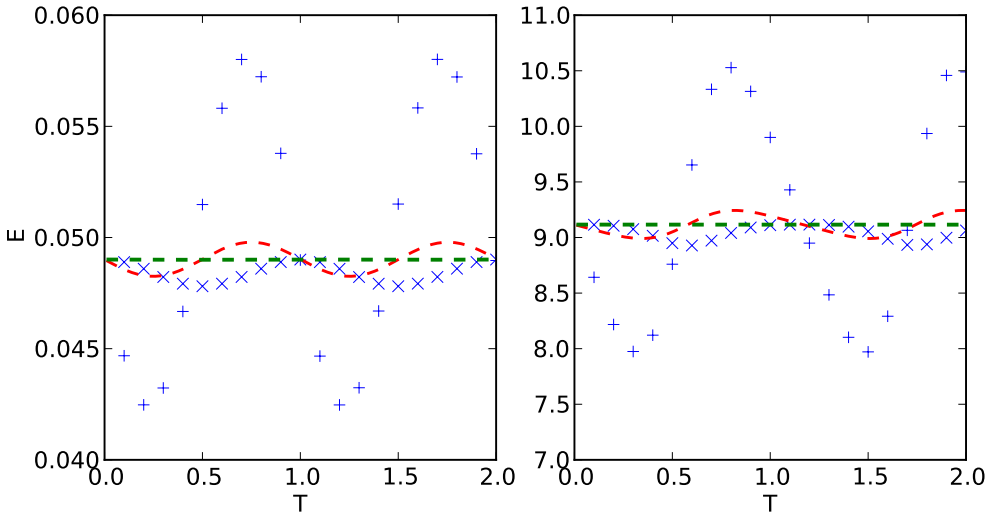


Abbildung 1.3: Energie als Funktion der Zeit, wieder für $l = 1m$, und Ausgangslage $\alpha = 1{,}5$ (links) und $\alpha = 0{,}1$ (rechts) in Ruhe. + markiert die Ergebnisse einer Integration mit dem einfachen Vorwärtsschritt (1.11) mit Zeitschritt $0{,}1s$, die gestrichelte rote Linie mit Zeitschritt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

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

Mit dem verbleibenden Resttableau wird nun genauso weiter verfahren:

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

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.

Index