

Computing im Physikunterricht

Aegidius Plüss, Bern, www.aplu.ch

1. Der Stellenwert des Computers im Physikunterricht

Im Alltag des Physikers, sei es als Experimentator oder Theoretiker, ist der Computer allgegenwärtig. Sein Einsatz kann aber ganz unterschiedliche Gründe haben: Beispielsweise steuert im Labor der Computer Experimente, sammelt Messdaten und führt Datenfilterungen und Datenanalysen durch. Dann vergleicht er Messresultate mit physikalischen Gesetzmässigkeiten. Für den Theoretiker führt der Computer manchmal schwierige algebraische Rechnungen und Umformungen durch oder simuliert die Theorie in einem virtuellen Experiment.

Ganz anders im gymnasialen Physikunterricht: Hier wird der Computer immer noch nur zögerlich eingesetzt, obschon es an Geräten und Software nicht fehlt. Ein Grund mag sein, dass Physiklehrpersonen manchmal ein gespaltenes Verhältnis zur Programmierung haben. Während es vor 30 Jahren selbstverständlich war, dass jeder Physiker wenigsten mit einer höheren Programmiersprache wie Basic, Pascal, C oder sogar mit Maschinensprache vertraut war, gilt dies heute für die weit verbreiteten Programmiersprachen C++, C#, Pascal/Delphi und Java nicht mehr. Eine Umfrage zeigt, dass das Programmieren mit diesen Programmiersprachen vor allem auf Grund des grafischen Benutzerinterfaces und der Objektorientierung als zu schwierig und für den Unterricht ungeeignet betrachtet wird. Zudem können sich die Physiklehrperson leider selten auf Programmierkenntnisse der Schülerinnen und Schüler abstützen, da es keinen verbindlichen Informatikunterricht gibt oder dort nicht oder in einer nicht für die Physik anwendbaren Programmiersprache unterrichtet wird. Die Verwendung von Tabellenkalkulation (Excel) oder des programmierbaren Taschenrechners ist eher ein Rettungsanker als eine ernstzunehmende Alternative. Die Motivation und der Erfolg beim Einsatz von Excel und Taschenrechnern ist bei Schülerinnen und Schülern darum erfahrungsgemäss bescheiden.

In diesem Dilemma zeigt sich, dass die Programmiersprache Python einen echten Ausweg bieten kann. Als Skriptsprache gilt Python gerade bei Nichtinformatikern als ebenso einfach wie früher Basic, da Python wie Basic und Pascal einen globalen Namensraum besitzt und damit ohne Klassenkonstruktionen auskommt. Andererseits besitzt aber Python ganz anders als Basic die Mächtigkeit einer modernen Programmiersprache und ist darum auch für den weiterführenden Informatikunterricht geeignet. Diese Anwendbarkeit auch auf komplexe Problemstellungen in allen Fächern macht gerade den Unterschied zur Tabellenkalkulation, zu anwendungsspezifischen Sprachen wie LabView, PHP und HTML5 oder zu didaktischen auferlegten Korsetten wie Kara, Logo, Greenfoot, Alice oder Scratch.

Trotz der hier angestrebten Begeisterung für die Verwendung des Computers als Programmierwerkzeug im Fachunterricht müssen aber immer Ziele und Inhalte des Physikunterrichts und nicht die der Informatik im Vordergrund stehen. Gerade deswegen ist es wichtig, ein Programmierwerkzeug zu verwenden, das von Lehrpersonen und Schülerinnen und Schülern mit minimalem Lernaufwand einsetzbar ist und auf

den eigenen Rechnern aller Plattformen, sowie in geschützten Computerpools leicht installiert werden kann. Gerade darin kann die Programmiersprache Python mit der plattformübergreifenden TigerJython IDE gemäss dem Slogan **Copy, Click & Go** punkten, denn es genügt, eine einzige Datei zu kopieren und mit einem Mausklick die Programmierumgebung zu starten. Da die Distribution bereits viele wichtige Zusatzmodule für die Turtlegrafik, Koordinatengrafik, Robotik und Spielprogrammierung enthält, sind auch keine Nachinstallationen notwendig. ([Hier](#) findet man weitere Informationen zur Verwendung und zum Download von TigerJython.)

2. Theorie, Experiment und Simulation

Fragt man an einer Physikprüfung danach, welches das grundlegenden Problemlösungsverfahren in den Naturwissenschaften ist, so erwartet man als Antwort, dass ein Problem grundsätzlich entweder theoretisch, d.h. unter Anwendung von physikalischen Gesetzen oder experimentell, d.h. mit Beobachtungen und Messungen zu lösen ist. Dabei übersieht man aber, dass in der Praxis des Naturwissenschaftlers heutzutage oft eine dritte Methode eingesetzt wird, nämlich die Computersimulation. In der Simulation wird die Realwelt ähnlich wie in einer Theorie modellartig in einer virtuellen Welt abgebildet und statt mit algebraischen mit numerischen Methoden behandelt. Man spricht darum auch oft von einem *Simulationsexperiment*. Wegen der notwendigen Vereinfachungen und der begrenzten Rechengenauigkeit sind die Resultate mit Fehlern behaftet und müssen darum sorgfältig diskutiert werden. Das Simulationsexperiment besitzt in der physikalischen Forschung einen ähnlichen hohen Stellenwert wie Theorie und Experiment und sollte darum auch einen viel grösseren Raum im Physikunterricht einnehmen. An einfachen exemplarisch ausgewählten Situationen und mit einfachen Programmierwerkzeugen lässt sich die Gleichwertigkeit von Theorie, Experiment und Simulation zuerst in der Mechanik, später aus anderen Gebieten besonders hübsch zeigen.

Das hier vorgeschlagene methodische Vorgehen ist das Folgende: Zuerst wird an einem Beispiel, das sich leicht theoretisch behandeln lässt, gezeigt, dass die numerische Lösung wegen der endlichen Rechengenauigkeit immer fehlerhaft ist, der Fehler allerdings bei entsprechendem Rechenaufwand meist so klein gehalten werden kann, dass die Abweichungen der Simulation innerhalb der Grenzen der Beobachtung liegen. Ziel dieser ersten Phase ist es, Vertrauen in die Simulationsmethode zu gewinnen. Nachher wird die gleiche Methode auf Probleme angewendet, für welche die theoretische Behandlung im Mittelschulunterricht zu aufwendig oder zu schwierig ist. Dadurch erfahren Schülerinnen und Schüler, dass die Physik auf reale, praxisrelevante Situationen anwendbar ist, ohne dass diese extrem vereinfachenden Modellannahmen (reibunglos, eindimensional, usw.) unterworfen werden.

2.1 Newtonsche Mechanik

Eine der Grundaufgaben der Physik besteht darin, das zeitliche Verhalten eines exakt präparierten Systems vorauszusagen, oder anders gesagt: auf Grund von bekannten Anfangsbedingungen zur Zeit $t = 0$ und bekannten äusseren Einflüssen den Zustand z als Funktion der Zeit $z = z(t)$ zu ermitteln. Für den einfachen Fall einer

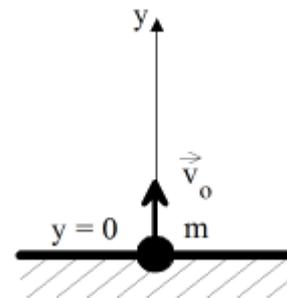
einzelnen Masse m , die sich in einem Kraftfeld befindet, heisst dies, dass aus Anfangsort und Anfangsgeschwindigkeit Ort und Geschwindigkeit zu jeder späteren Zeit $t > 0$ anzugeben ist.

2.1.1 Vertikaler Wurf

Als erstes Beispiel betrachten wir die eindimensionale Bewegung mit konstanter Beschleunigung, also die vertikale reibungslose Wurfbewegung. Wählen wir die die Koordinatenachse y vertikal nach oben mit dem Ursprung bei der Abwurfstelle, so lautet die Ortsfunktion bekanntlich

$$y = v_0 t - \frac{g}{2} t^2$$

wobei v_0 die Anfangsgeschwindigkeit und g die Erdbeschleunigung sind. In der Simulation wählt man einen diskreten Zeitschritt dt und nimmt als Näherung für die Änderung des Ortes eine gleichförmige und für die Änderung der Geschwindigkeit eine gleichmässig beschleunigte Bewegung an. Dieses Vorgehen ist für viele Simulationen sehr typisch. Es ist anzunehmen, dass der Fehler dieser Näherung umso kleiner ist, je kleiner dt im Vergleich zur gesamten Beobachtungszeit T ist. Garantiert ist dies aber nicht, da bei kleiner werdendem dt ja auch die Schrittzahl $n = T/dt$ ansteigt und damit der kumulierte Fehler mit zunehmendem n nicht unbedingt sinkt. Glücklicherweise ist dies aber nur in exotischen Fällen ein Problem.



In *mechanik1.py* zeigt sich, wie einfach die Umsetzung in ein Python-Programm ist. Dabei werden die Werte für y und v im Ausgabefenster formatiert ausgeschrieben.

```
# mechanik1.py
# Vertikaler Wurf ohne Luftwiderstand

g = 9.81 # Erdbeschleunigung (m/s^2)
dt = 0.1 # Zeitschritt (s)

# Anfangsbedingungen:
t = 0
y = 0 # Ort (m)
v = 20 # Geschwindigkeit (m/s)

while t < 10:
    print "%4.1f%8.2f%8.2f" %(t, y, v)
    v = v - g * dt
    y = y + v * dt
    t = t + dt
```

[Ausführen](#) mit Webstart

Die Simulation zeigt, dass die Wurfhöhe ungefähr 20 m beträgt. Da v die zeitliche Ableitung von y ist, handelt es sich bei der hier verwendeten Methode zur Berechnung von y um das **Eulerverfahren** (bzw. **Runge-Kutta 1. Ordnung**):

$$y(t_{i+1}) \approx y(t_i) + y'(t_i)h$$

Manchmal spricht man auch von einer **Linearisierung**.

2.1.2 Vertikaler Wurf mit Grafik

Fast alle Simulationen werden mit grafischen Ausgaben veranschaulicht. In *TigerJython* steht dazu ein Grafikfenster *GPanel* zur Verfügung, das ein frei wählbares dezimales Koordinatensystem hat und darum für physikalische Probleme gut geeignet ist. Damit entfallen mühsame Umrechnungen auf Pixelkoordinaten.

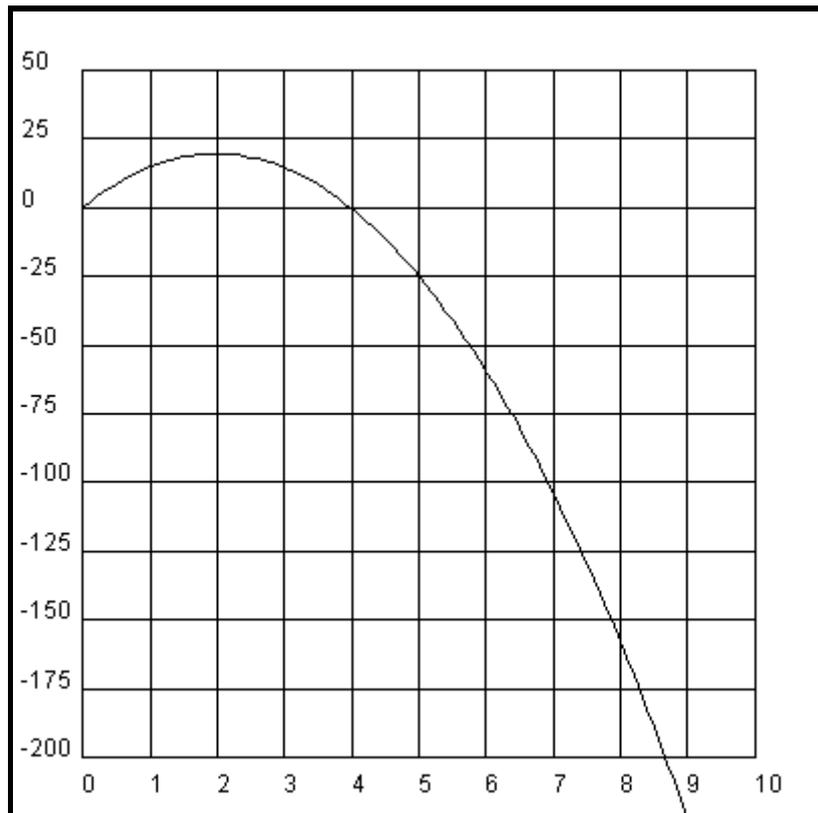
```
# mechanik2.py
# Vertikaler Wurf ohne Luftwiderstand mit Grafik

from gpanel import *

g = 9.81 # Erdbeschleunigung (m/s^2)
dt = 0.1 # Zeitschritt (s)

# Anfangsbedingungen:
t = 0
y = 0 # Ort (m)
v = 20 # Geschwindigkeit (m/s)

makeGPanel(-1, 11, -220, 70) #xmin, xmax, ymin, ymax
drawGrid(0, 10, -200, 50)
move(t, y)
while t < 10:
    draw(t, y)
    v = v - g * dt
    y = y + v * dt
    t = t + dt
```



[Ausführen](#) mit Webstart

Das Koordinatengitter und die Beschriftung sind zwar rudimentär, lassen sich aber automatisch mit der einzigen Anweisung `drawGrid()` einbauen.

2.1.3 Vertikaler Wurf. Vergleich mit Theorie

Um das Vertrauen die Simulation zu stärken, wird der simulierte Verlauf mit der theoretischen Voraussage verglichen. Dabei zeigt sich selbst bei einem relativ grossen Zeitschritt von $dt = 0.1$ s eine gute Übereinstimmung. Es ist nun aber im Gegensatz zum Menschen für den Computer kein Problem, die Simulation von einem zehner oder hundert Mal kleineren Schritt auszuführen. Es ist dann praktisch kein Unterschied zwischen Theorie und Simulation mehr zu erkennen.

```
# mechanik3.py
# Vertikaler Wurf ohne Luftwiderstand
# Vergleich mit Theorie

from gpanel import *

g = 9.81 # Erdbeschleunigung (m/s^2)
dt = 0.1 # Zeitschritt (s), kann verkleinert werden

# Anfangsbedingungen:
t = 0
y = 0 # Ort (m)
v = 20 # Geschwindigkeit (m/s)
```

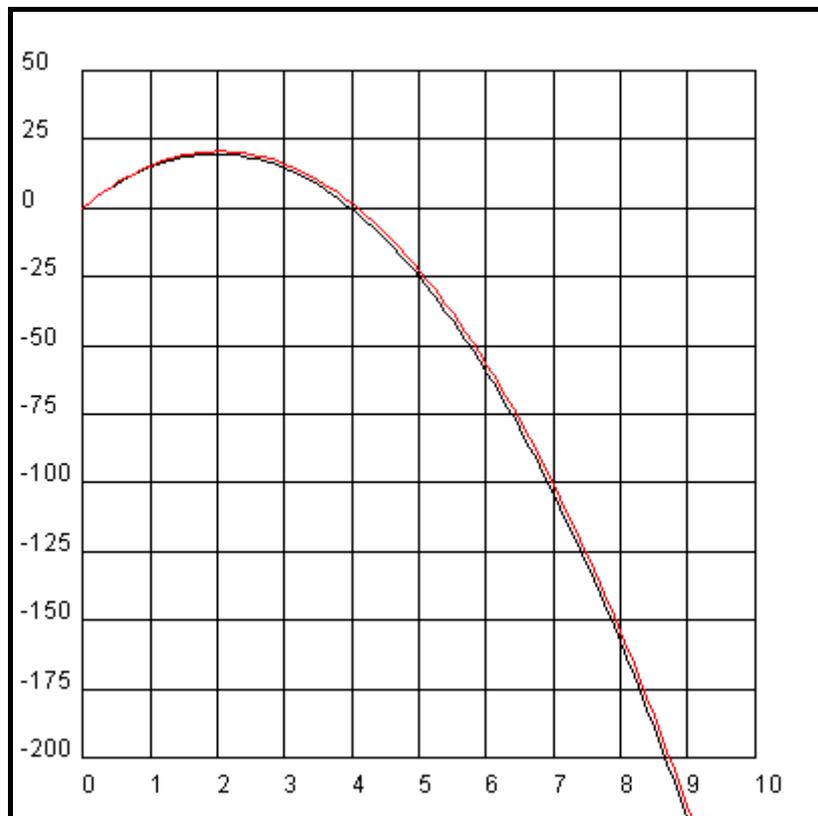
```

makeGPanel(-1, 11, -220, 70) #xmin, xmax, ymin, ymax
drawGrid(0, 10, -200, 50)
move(t, y)
while t < 10:
    draw(t, y)
    v = v - g * dt
    y = y + v * dt
    t = t + dt

# Theorie
setColor("red")
t = 0; y = 0; v0 = 20

move(t, y)
while t < 10:
    y = v0*t - g/2*t*t
    draw(t, y)
    t = t + dt

```

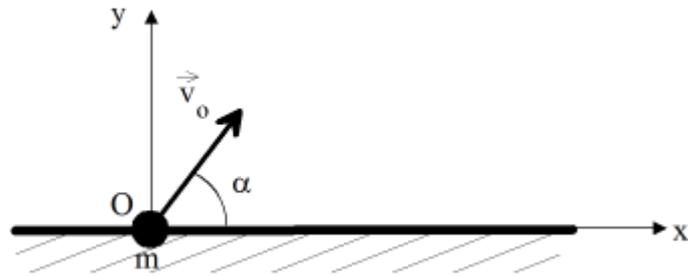


[Ausführen](#) mit Webstart

2.1.5 Schiefer Wurf mit Luftwiderstand (Ballistische Kurve)

Mit gestärktem Vertrauen in die Simulation wenden wir sie auf einen Fall an, der sich mit elementaren Mitteln bereits nicht mehr behandeln lässt. Dabei werfen wir die

Masse m in homogenen Gravitationsfeld, nehmen aber an, dass diese eine zur Geschwindigkeit im Quadrat proportionale Luftreibung gemäss der Beziehung $R = \rho v^2$ oder in vektorieller Schreibweise



$$\vec{R} = -\rho v^2 \frac{\vec{v}}{v} = -\rho v \vec{v} \text{ erf\u00e4hrt.}$$

```
# mechanik4.py
# Schiefer Wurf mit Luftwiderstand
# Luftwiderstand prop v^2

from gpanel import *
from math import *

g = 9.81 # Erdbeschleunigung (m/s^2)
dt = 0.1 # Zeitschritt (s)

m = 8.8e-3 # Kugelmasse (kg)
rho = 1.06e-3 # Reibungskoeffizient

# Anfangsbedingungen:
t = 0
v0 = 20 # Anfangsgeschwindigkeit (m/s)
alpha = 50 # Abschusswinkel (Grad)
alpha = alpha / 180 * pi # Bogenmass
x = 0; y = 0
vx = v0 * cos(alpha)
vy = v0 * sin(alpha)

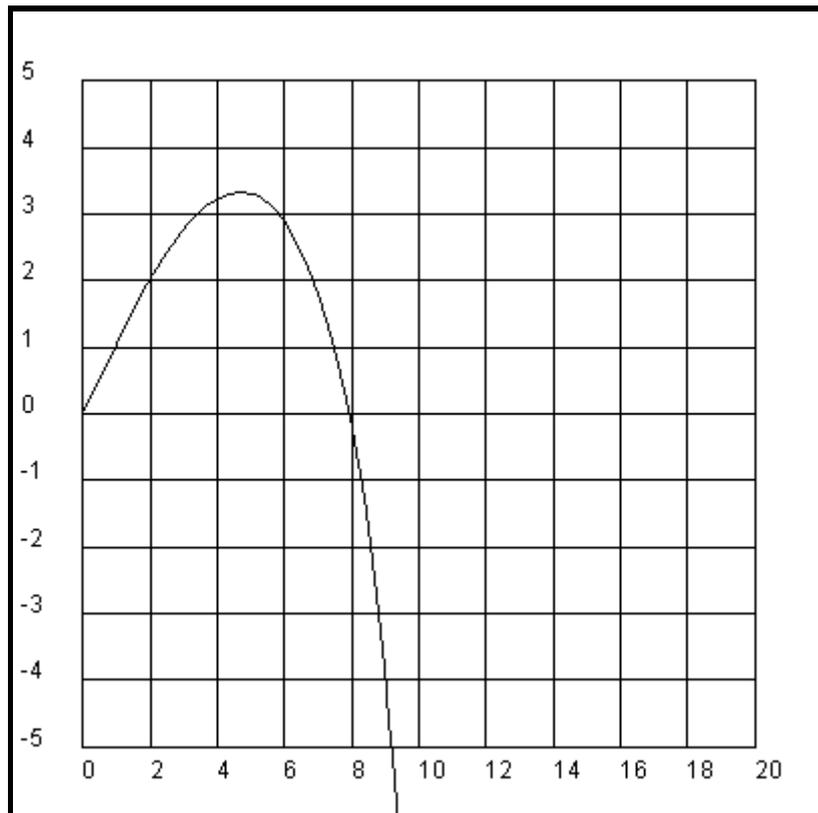
makeGPanel(-2, 22, -6, 6)
drawGrid(0, 20, -5, 5)
move(x, y)
while t < 10:
    draw(x, y)
    fx = -rho*sqrt(vx*vx + vy*vy)*vx # Kraftkoordinaten
    fy = -m*g - rho*sqrt(vx*vx + vy*vy)*vy

    ax = fx/m # Beschleunigungskordinaten
    ay = fy/m

    vx = vx + ax*dt # Geschwindigkeitskordinaten
    vy = vy + ay*dt

    x = x + vx*dt; # Ortskordinaten
    y = y + vy*dt;

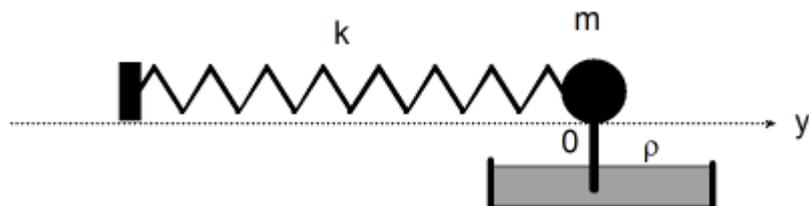
    t = t + dt
```



[Ausführen](#) mit Webstart

2.1.6 Gedämpfte Schwingung

Die harmonischen Schwingung wird wohl in jedem Physikunterricht behandelt, steht sie doch im Zentrum des Themengebiets "Schwingungen und Wellen", das wegen seines Aktualitäts- und Umweltbezugs, seinen Verbindungen zu anderen Themen weit über die Mechanik hinaus und seiner Beliebtheit bei Schülerinnen und Schülern zu den schönsten gymnasialen Themen überhaupt gehört. Bereits bei der Federschwingung werden wir aber bei der theoretischen Lösung mit einer Differentialgleichung und damit mit der Differentialrechnung konfrontiert. Ob dieser Weg gangbar ist, hängt von Schultyp und Schulstufe ab. Mit den oben erarbeiteten elementaren Programmierkenntnissen ist die Simulation geradezu trivial und benötigt keine weitergehenden mathematischen Vorkenntnisse. Wir betrachten gleich den Fall mit einer geschwindigkeitsproportionalen Dämpfung.



```
# harmon1.py
# Gedämpfte harmonische Schwingung

from gpanel import *

dt = 0.01      # Zeitschritt (s)
d = 200       # Gesamtdauer (s)
```

```

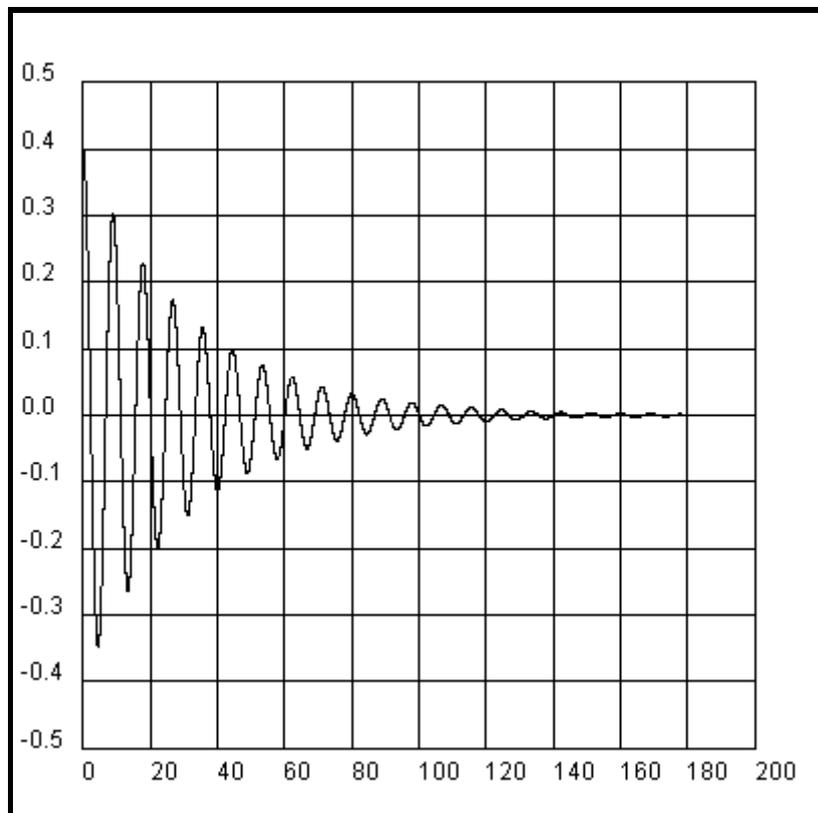
m = 0.8           # Masse (kg)
k = 0.4           # Federkonstante (N/kg)
r = 0.05          # Reibungskoeffizient in N/m/s

t = 0; y = 0.4; v = 0 # Anfangsbedingungen

makeGPanel(-20, 220, -0.6, 0.6)
drawGrid(0, 200, -0.5, 0.5)

move(t, y) # Anfangspunkt
while t < d:
    draw(t, y) # Zeichne
    a = -k*y / m - r / m * v # Beschleunigung
    v = v + a*dt # Neue Geschwindigkeit
    y = y + v*dt # Neue Koordinate
    t = t + dt # Neue Zeit

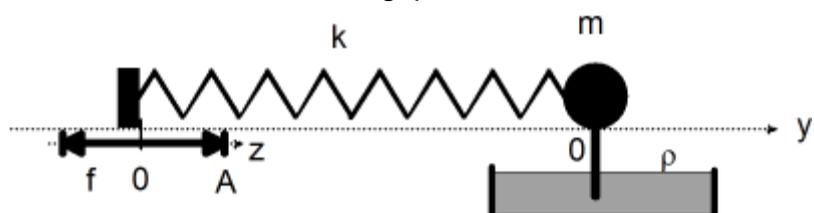
```



[Ausführen](#) mit Webstart

2.1.7 Erzwungene Schwingung

Noch spannender ist es, mit einer Computersimulation das fundamentale Phänomen der **Resonanz** zu untersuchen. Dabei ist der Anfangspunkt der Feder mit einem inusförmigen Erreger verbunden, der mit der Erregerfrequenz f schwingt. Das Simulationsprogramm ist kaum komplizierter.



```

# harmon2.py
# Erzwungene harmonische Schwingung

from gpanel import *
from math import *

dt = 0.01      # Zeitschritt (s)
d = 200       # Gesamtdauer (s)

m = 0.8       # Masse (kg)
k = 0.4       # Federkonstante (N/kg)
r = 0.05      # Reibungskoeffizient in N/m/s
y0 = 0.4      # Anfangsort (m)
v0 = 0        # Anfangsgeschwindigkeit

A = 0.05      # Erregeramplitude (m)
f = 0.1       # Erregerfrequenz (Hz)

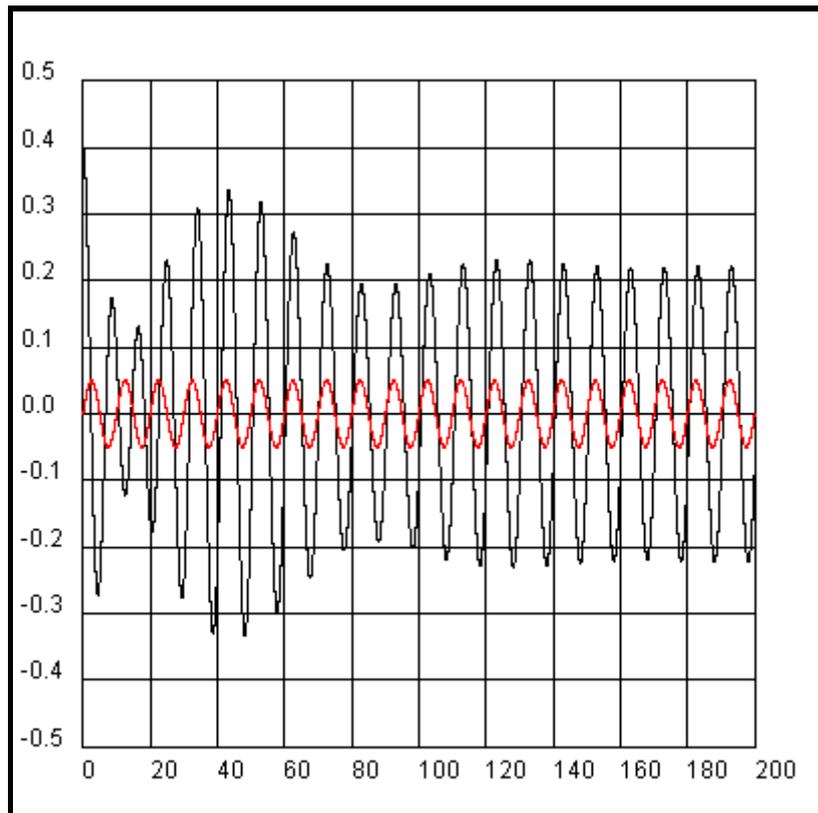
t = 0; y = y0; v = v0 # Anfangsbedingungen
print "Eigenfrequenz =", sqrt(k / m) / 2 / pi, "Hz"

makeGPanel(-20, 220, -0.6, 0.6)
drawGrid(0, 200, -0.5, 0.5)

move(t, y0)   # Anfangspunkt
while t < d:
    draw(t, y) # Zeichne
    z = A * sin(2*pi*f*t) # Erregerort
    a = -k*(y - z) / m - r / m * v # Beschleunigung
    v = v + a*dt           # Neue Geschwindigkeit
    y = y + v*dt          # Neue Koordinate
    t = t + dt           # Neue Zeit

setColor("red")
t = 0
move(0, 0)
while t < d:
    y = A*sin(2*pi*f*t)
    draw(t, y)
    t = t + dt

```



[Ausführen](#) mit Webstart

Durch Variation der Dämpfungskonstanten und der Erregerfrequenz können leicht Resonanzkurven aufgenommen werden. Es ist eine besonders motivierende Herausforderung, dies mit einem Programm automatisch zu machen.

```
# harmon3.py
# Resonanzkurve

from gpanel import *
from math import *

dt = 0.01          # Zeitschritt (s)
d = 200           # Gesamtdauer (s)

m = 0.8           # Masse (kg)
k = 0.4           # Federkonstante (N/kg)
r = 0.1           # Reibungskoeffizient in N/m/s
y0 = 0.4          # Anfangsort (m)
v0 = 0            # Anfangsgeschwindigkeit

fEigen = sqrt(k / m) / 2 / pi

A = 0.05          # Erregeramplitude (m)
makeGPanel(-20, 220, -0.6, 0.6)
f = 0.02 # Erregerfrequenz
points = []

while f <= 0.2:
    clear()
    drawGrid(0, 200, -0.5, 0.5)
```

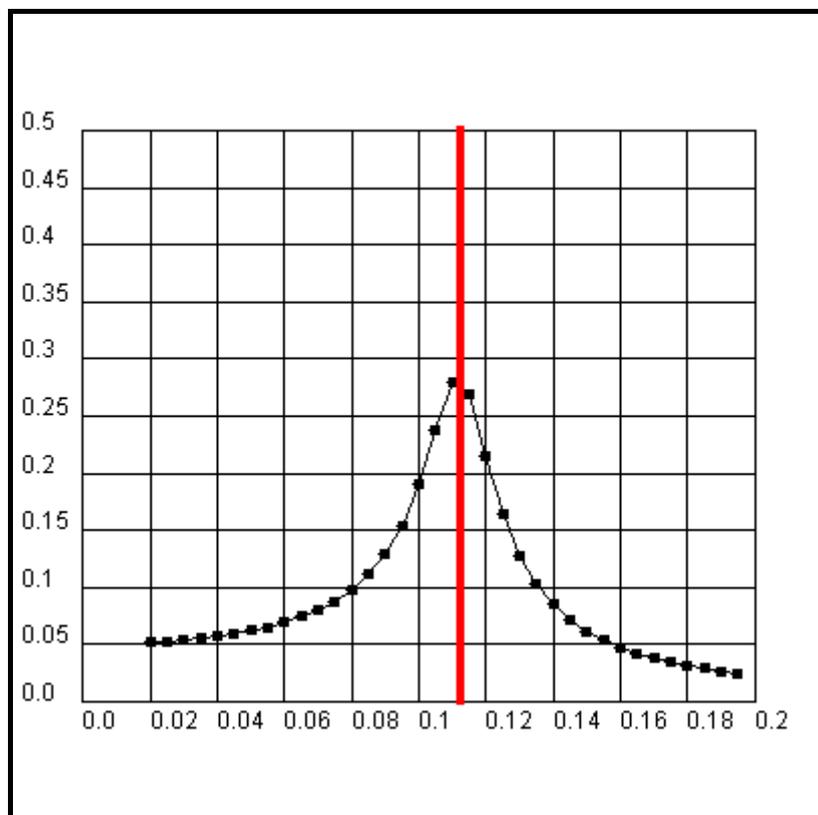
```

t = 0; y = y0; v = v0 # Anfangsbedingungen
title("Erregerfrequenz = " + str(f) + " Hertz")
move(t, y0) # Anfangspunkt
ampl = 0
while t < d:
    draw(t, y) # Zeichne
    z = A * sin(2*pi*f*t) # Erregerort
    a = -k*(y - z) / m - r / m * v # Beschleunigung
    v = v + a*dt # Neue Geschwindigkeit
    y = y + v*dt # Neue Koordinate
    t = t + dt # Neue Zeit
    if t > 150:
        ampl = max(ampl, y)
title("Erregerfrequenz = " + str(f) + " Hertz. Amplitude = " +
str(ampl))
points.append([f, ampl])
f += 0.005

clear()
makeGPanel(-0.02, 0.22, -0.1, 0.6)
drawGrid(0, 0.2, 0, 0.5)
title("Resonanzkurve")
for i in range(len(points)):
    move(points[i])
    fillCircle(0.002)
    if i > 0:
        draw(points[i-1])

setColor("red")
lineWidth(4)
line(fEigen, 0, fEigen, 0.5)

```



[Ausführen](#) mit Webstart

2.1.8 Keplersche Gesetze (Erdsatellit)

Eine weitere didaktische Knacknuss im Physikunterricht ist die Behandlung der Keplerschen Gesetze. Wegen ihrer grossen Bedeutung im Alltag, werden sie wohl in keinem Unterricht fehlen. Ihre Herleitung durch eine theoretische Behandlung des Zweikörperproblems ist aber jenseits der gymnasialen Mathematik. Auch hier kann die Simulation in überzeugender Art zeigen, dass Isaak Newton allein durch Anwendung der beiden Beziehungen

$$\vec{F} = m\vec{a} \quad \text{und} \quad \vec{F} = -\frac{Gm_1m_2}{r^3}\vec{r}$$

die Keplerschen Gesetze beweisen konnte.

```
# kepler.py

from gpanel import *
from math import *
import time

dt = 10 # Zeitschritt (s)
RE = 6370 # Erdradius in km
G = 6.67e-11 # Gravitationskonstante (SI)
M = 5.97e24 # Erdmasse (kg)

# Anfangsbedingungen:
t = 0
h = 320 # Hoehe über der Erdoberfläche (km)
v0 = 9.5 # km/s

x = 1000 * (RE+h)
y = 0
vx = 0
vy = 1000 * v0

# coordinate span
xmin = -31000
xmax = 9000
ymin = -15000
ymax = 15000

def drawEarth():
    move(0, 0)
    setColor("lightblue")
    fillCircle(RE)
    setColor("black")
    circle(RE)

def step():
    global t, x, y, vx, vy
    # Zeichne Markierungskreis
    if int(t) % 500 == 0:
        fillCircle(150)
        title("time = " + str(t) + " s")

    draw(x/1000, y/1000)
    r = sqrt(x*x+y*y)
    ax = -G*M*x / (r * r * r)
```

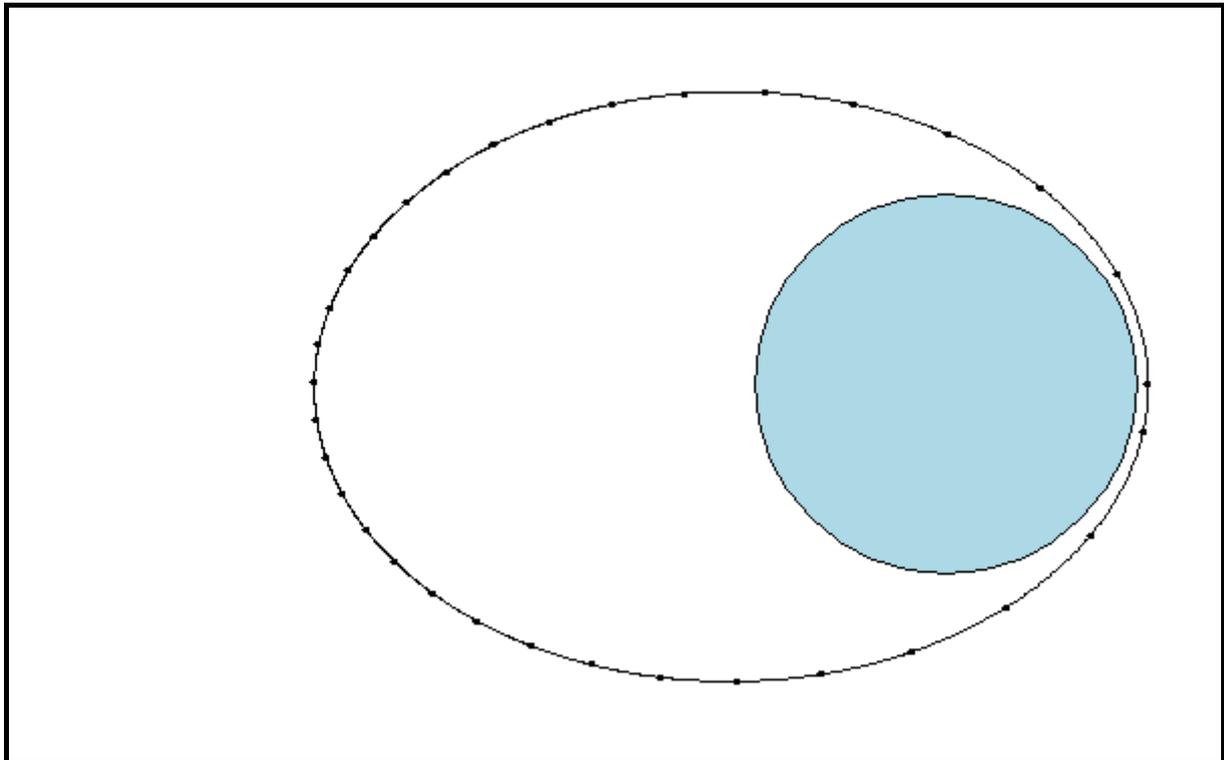
```

ay = -G*M*y / (r * r * r)
vx += ax*dt
vy += ay*dt
x += vx*dt
y += vy*dt
t += dt

makeGPanel(Size(1000, 750))
windowPosition(10, 10)
window(xmin, xmax, ymin, ymax)
drawEarth()
move(x/1000, y/1000)

while True:
    now = time.clock()
    step()
    while (time.clock() - now) < 0.01:
        delay(1)
    p = atan2(y, x)
    if p > -0.01 and p < 0:
        break

```



[Ausführen](#) mit Webstart

Um die Keplerschen Gesetze nachzuweisen, ist noch etwas Handarbeit mit den ausgedruckten Grafiken nötig.

2.2 Kosmologie

2.2.1 Dreikörperproblem

Die Keplerschen Gesetze beziehen sich auf ein extrem einfaches System, bei dem sich eine Masse im Gravitationsfeld eines fixen Zentralkörpers befindet.

Verallgemeinert man diese Situation auf mehrere Körper, die unter gegenseitiger gravitativer Wechselwirkung stehen, so nimmt der Schwierigkeitsgrad für eine analytische Lösung schnell zu. Bereits das allgemeine Dreikörperproblem ist bekanntlich nicht mehr geschlossen lösbar. Die Simulation verwendet aber dasselbe Muster und wird damit mit steigender Zahl der Himmelskörper nur unwesentlich schwieriger. Allerdings nehmen die Rechenzeiten zu und es gilt, sich kumulierende numerische Fehler in den Griff zu bekommen. Ein Mittel besteht darin, den Zeitschritt der Beschleunigung anzupassen.

Das Dreikörperproblem ist besonders instruktiv, da sich dabei typische beobachtbare Situationen diskutieren lassen. Je nach den Massen und der Anfangsbedingungen resultiert ein ganz anderes Verhalten. In den folgenden Beispiele werden Positionsdaten aus einer Datei verwendet, die ein Simulationsprogramm über eine längere Rechenzeit (mehrere Stunden) errechnet hat. Die Bewegung der drei Körper lässt sich so als Animation anschaulich verfolgen. Es handelt sich ausschliesslich um ebene Bewegungen.

Alle Programme verwenden eine Textdatei, in der die x, y Koordinaten der Körper 0, 1 und 2 zeilenweise in folgendem Format gespeichert sind:

```
x-coord0 spaces y-coord0
x-coord1 spaces y-coord1
x-coord2 spaces y-coord2
empty line
```

Das Ende der Daten ist durch eine Zeile mit dem String "eof" gekennzeichnet (an Stelle der Leerzeile). Die Datenfiles *astro1.dat*, *astro2.dat*, *astro3.dat*, *astro4.dat* und *astro5.dat* können gepackt von <http://www.tigerjython.ch/download/astro.zip> heruntergeladen werden und müssen sich im gleichen Verzeichnis wie das Programm befinden.

2.2.1.1 Einfang eines interstellaren Kometen

Ein aus dem interstellaren Raum einfallender Komet kann in einem Planetensystem eingefangen werden, falls er sich nahe an einem Planeten vorbei bewegt. Wie man mit einer Computersimulation herausgefunden hat, besitzt in unserem Sonnensystem allerdings nur Jupiter eine genügend grosse Masse, um Kometen einzufangen.

```
# astrol.py

from gpanel import *
import time
import copy

xcenter = 0
ycenter = 0
size = 4
tit = "Comet Capture."
dat = "astrol.dat"

def step():
    global pOld
    for k in range(3): # index of body
        data = f.readline().split()
```

```

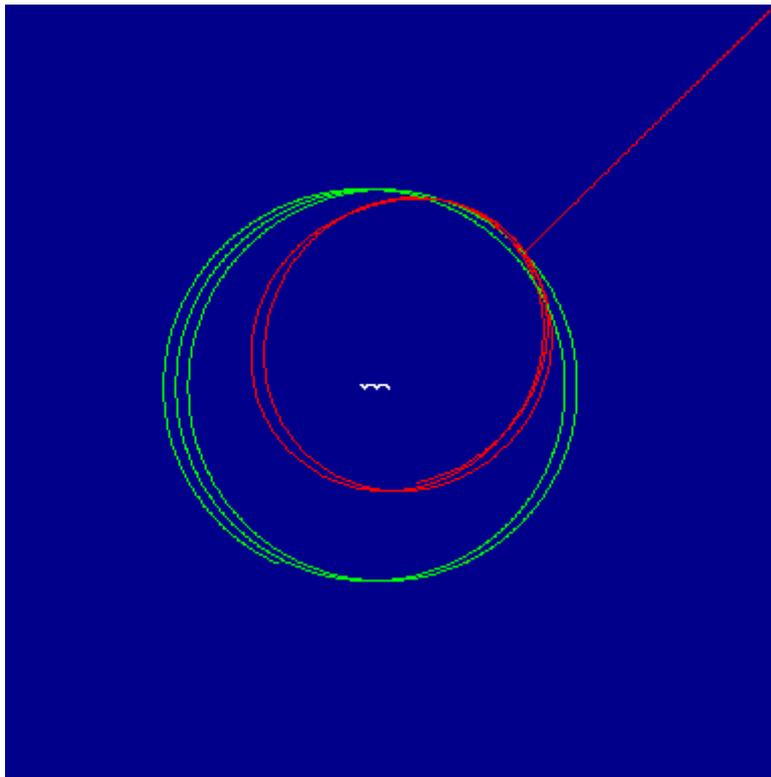
    p[k][0] = float(data[0])
    p[k][1] = float(data[1])
    data = f.readline() # empty line
    if "eof" in data: # End of file
        return True
    if pOld != None:
        for k in range(3): # index of body
            setColor(colors[k])
            line(pOld[k], p[k])
    pOld = copy.deepcopy(p) # save old coordinates
    return False

colors = ["white", # trace 1st body
          "green", # trace 2nd body
          "red"] # trace 3rd body

makeGPanel(xcenter - size/2, xcenter + size/2,
           ycenter - size/2, ycenter + size/2)
title(tit)
bgColor("darkblue")

p = [[0, 0], # [x, y] 1st body
      [0, 0], # [x, y] 2nd body
      [0, 0]] # [x, y] 3rd body
pOld = None
f = open(dat)
eof = False # end of file
dt = 0.015
while not eof and not isDisposed():
    t = time.clock()
    eof = step()
    while time.clock() - t < dt:
        delay(1)
f.close()
title(tit + " Done.")

```



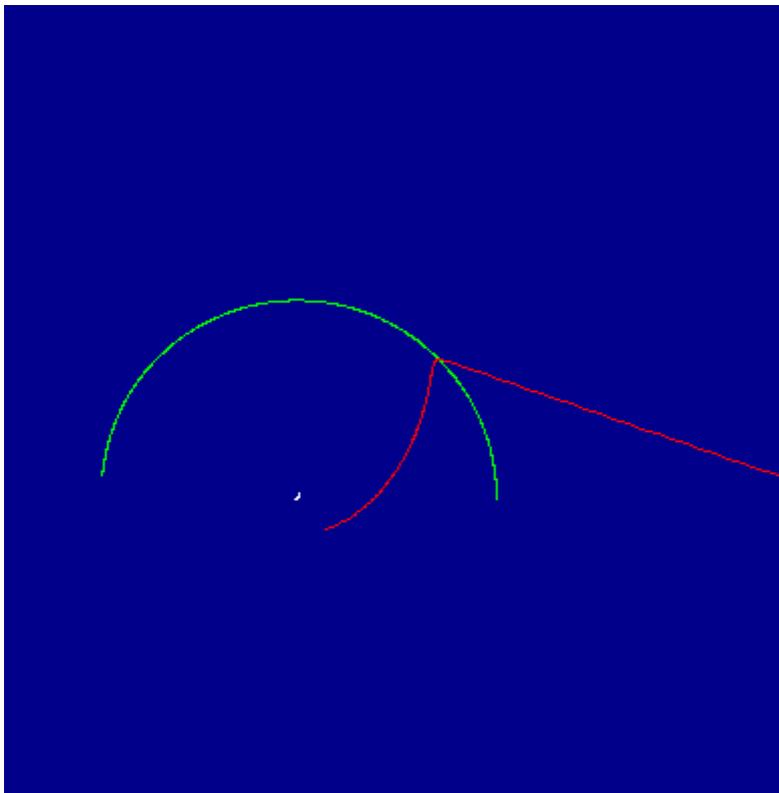
[Ausführen](#) mit Webstart

Die verschiedenen Programme unterscheiden sich nur durch die Wahl des Koordinatensystems und des Datenfiles.

2.2.1.2 Swing-by

Der Swing-by wird oft in der Raumfahrt verwendet, um die Bahn und die Geschwindigkeit eines Flugkörpers zu verändern, insbesondere um ihm eine grössere Schnelligkeit zu verleihen.

```
xcenter = 0.5  
ycenter = 0.5  
size = 4  
tit = "Swing-by."  
dat = "astro2.dat"
```

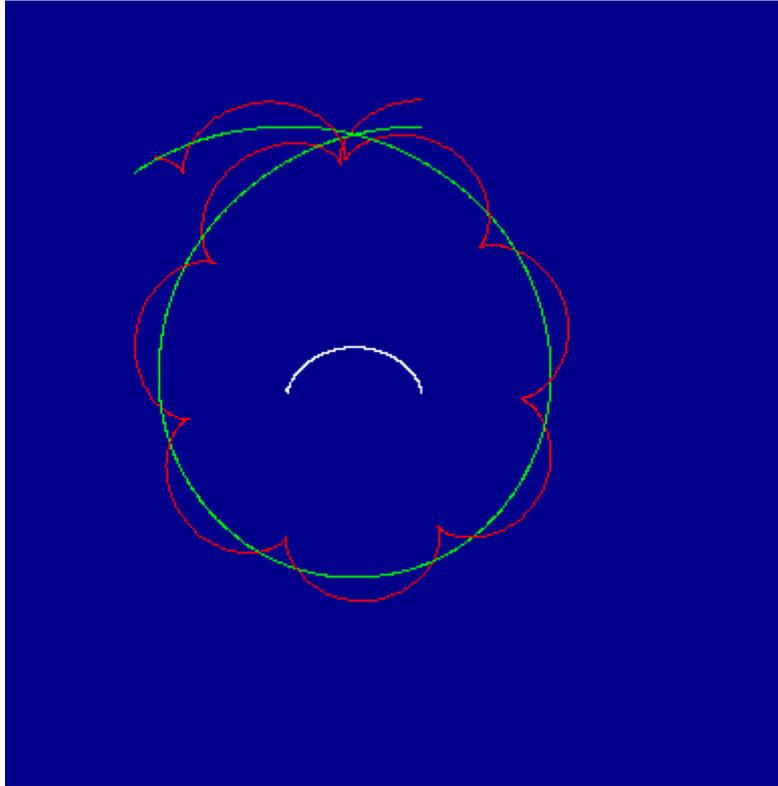


[Ausführen](#) mit Webstart

2.2.1.3 Sonne-Erde-Mond

Die Effekte werden durch eine andere Wahl der Massen verstärkt gezeigt.

```
xcenter = -0.1  
ycenter = 0  
size = 3  
tit = "Sun-Earth-Moon."  
dat = "astro3.dat"
```

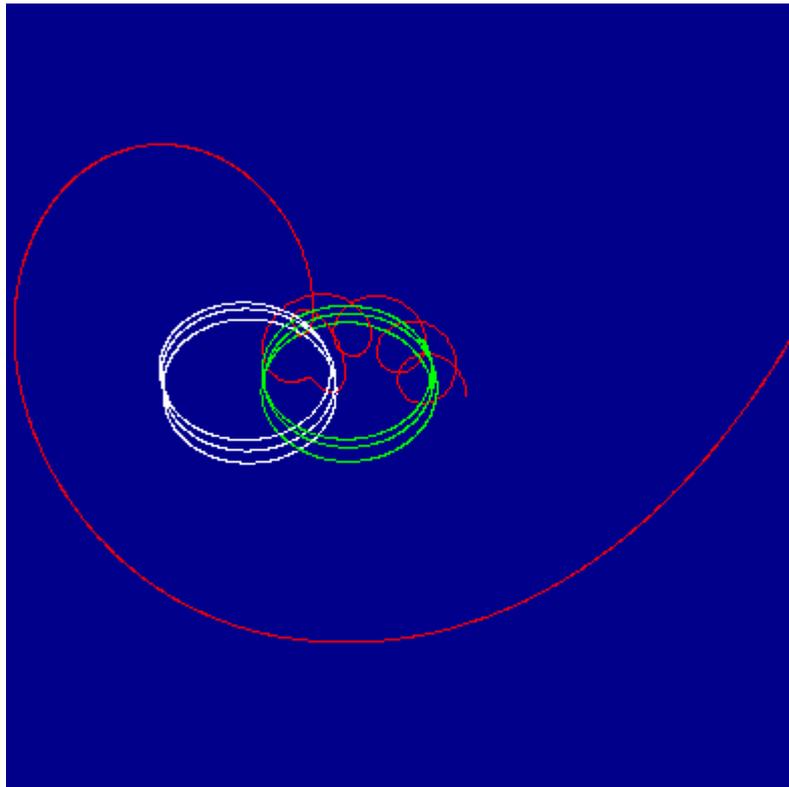


[Ausführen](#) mit Webstart

2.2.1.4 Doppelstern mit instabilem Planet

Die Stabilität eines Planetensystems ist kritisch. Es ist eigentlich verwunderlich, dass unser Sonnensystem offenbar stabil ist. Die Frage hat aber die Astronomen während Jahrhunderten beschäftigt. Im folgenden Beispiel wird eine Planet aus dem Doppelsternsystem hinauskatapultiert.

```
xcenter = 0.8  
ycenter = 0  
size = 6  
tit = "Double-Star (unstable planet)."  
dat = "astro4.dat"
```

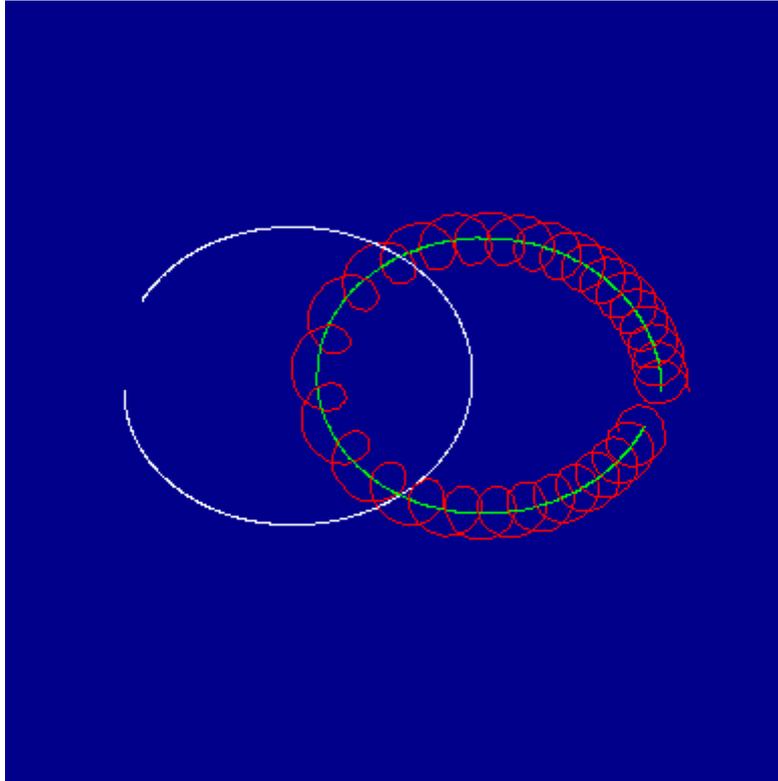


[Ausführen](#) mit Webstart

2.2.1.5 Doppelstern mit stabilem Planet

Das Doppelsternsystem mit einem Planet kann aber durchaus auch stabil sein. Die Frage stellt sich natürlich, ob dies auch in ferner Zukunft so sein wird.

```
xcenter = 0  
ycenter = 0  
size = 4  
tit = "Double-Star (stable planet)."  
dat = "astro5.dat"
```



[Ausführen](#) mit Webstart

2.2.2 Hubble-Gesetz

Die heute allgemein akzeptierte Entwicklungsgeschichte des Universums geht davon aus, dass es vor langer Zeit einen Urknall (Big Bang) gegeben hat, und sich seitdem das Universum ausdehnt. Im Vordergrund steht die Frage, wie lange der Urknall zurückliegt, was man als Alter des Universums bezeichnet. Der Astronom Hubble hat 1929 seine weltberühmten Untersuchungen publiziert, in denen er feststellte, dass es einen linearen Zusammenhang zwischen der Distanz d der Galaxien und ihrer Fluchtgeschwindigkeit v gibt. Das Hubble-Gesetz lautet:

$$v = H * d$$

wo H die Hubble-Konstante genannt wird.

Man kann die astrophysikalischen Überlegungen nachvollziehen, indem man von folgenden experimentellen Daten ausgeht, die vom Hubble-Weltraumteleskop stammen

Im Programm werden die Daten jeder Galaxie in einer Liste [Name, Distanz (Mpc), Geschwindigkeit (km/s)] und alle Galaxiendaten wiederum in einer Liste gespeichert. Mit einer linearen Regression wird dann H bestimmt. Geht man davon aus, dass die Geschwindigkeit v einer bestimmten Galaxie konstant geblieben ist, so ist ihre Distanz $d = v * T$, wo T das Alter des Universums ist. Mit dem Hubble-Gesetz $v = H * d$ folgt daraus $T = 1 / H$.

```

# hubble.py

from gpanel import *

# from Freedman et al, The Astrophysical Journal, 553 (2001) (Hubble Space
# Telescope)
data = [ ["NGC0300", 2.00, 133], ["NGC095", 9.16, 664], ["NGC1326A", 16.14,
1794],
["NGC1365", 17.95, 1594], ["NGC1425", 21.88, 1473], ["NGC2403", 3.22, 278],
["NGC2541", 11.22, 714], ["NGC2090", 11.75, 882], ["NGC3031", 3.63, 80],
["NGC3198", 13.80, 772], ["NGC3351", 10.0, 642], ["NGC3368", 10.52, 768],
["NGC3621", 6.64, 609], ["NGC4321", 15.21, 1433], ["NGC4414", 17.70, 619],
["NGC4496A", 14.86, 1424], ["NGC4548", 16.22, 1384], ["NGC4535", 15.78,
1444],
["NGC4536", 14.93, 1423], ["NGC4639", 21.98, 1403], ["NGC4725", 12.36,
1103],
["IC4182", 4.49, 318], ["NGC5253", 3.15, 232], ["NGC7331", 14.72, 999]]

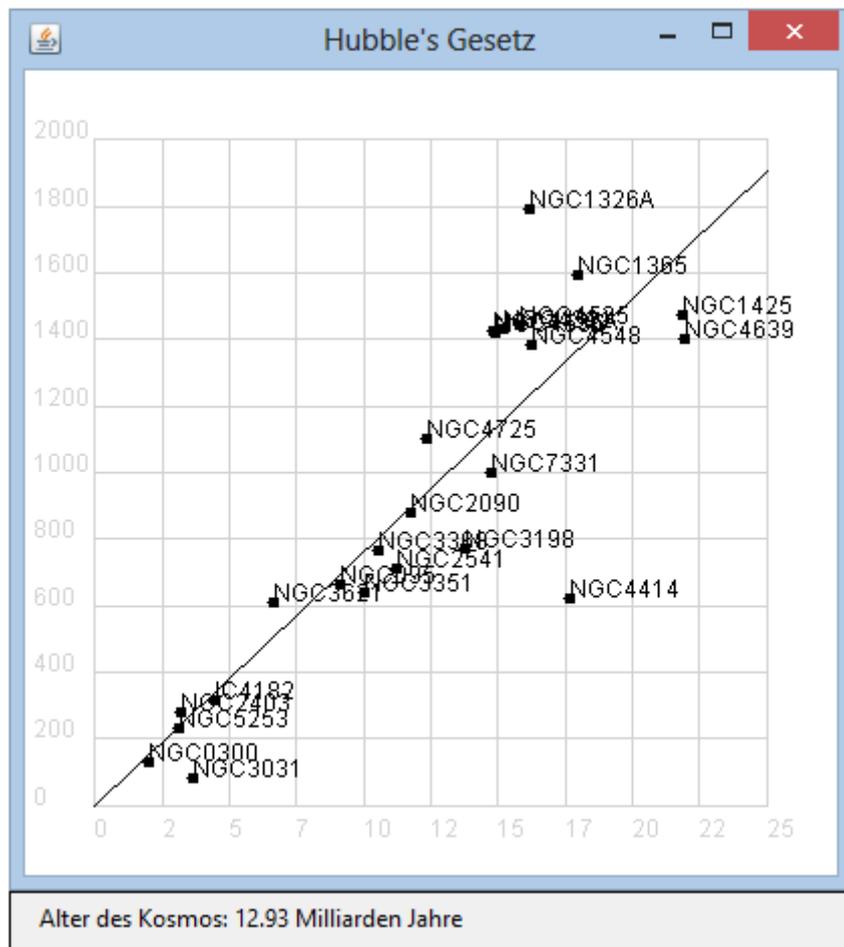
xval = []
yval = []

makeGPanel(-2.5, 27.5, -200, 2200)
title("Hubble's Gesetz")
addStatusBar(30)
drawGrid(0, 25, 0, 2000, "lightgray" )

for galaxy in data:
    d = galaxy[1]
    v = galaxy[2]
    xval.append(d)
    yval.append(v)
    move(d, v)
    fillCircle(0.2)
    text(galaxy[0])

H = linfit(xval, yval)[0]
line(0, 0, 25, H * 25)
mpc = 3.09E19 # 1 mpc in km
H = H / mpc # per s
t = 1 / H # age of cosmos in s
setStatusText("Alter des Kosmos: " + str(round(t / 3.14E16, 2)) + "
Milliarden Jahre")

```



[Ausführen](#) mit Webstart

2.3 Radioaktivität

Experimente mit radioaktiven Stoffen sind nicht gefahrlos und können nur unter Aufsicht durch eine entsprechend ausgebildete Lehrpersonen durchgeführt werden. Die Simulation hat zwar nie den gleichen Stellenwert wie das Experiment, ist aber gefahrlos und es wird kein Experimentiermaterial benötigt.

2.3.1 Radioaktiver Zerfall

Für radioaktive Nuklide gilt das fundamentale (von A. Einstein bestrittene) Naturgesetz, dass ein bestimmter Kern ohne Ursache zerfällt und die Wahrscheinlichkeit dp , dass dies im nächsten (infinitesimalen) Zeitintervall dt geschieht, unabhängig davon ist, wie lange es bereits vorhanden ist (gelebt hat). Es gilt

$$dp = \lambda * dt$$

wo man λ Zerfallskonstante nennt, die typisch für das Radionuklid ist.

Bei der Simulation einer Population von N_0 Radionukliden sollte man diese als Einzelindividuen betrachten und nicht sofort die kontinuierliche Approximation

$$dN = -N * \lambda * dt$$

betrachten, die auf eine Differentialgleichung führt, da man damit den typisch statistischen Charakter des radioaktiven Zerfalls, d.h. die Unvorhersehbarkeit für das Einzelindividuum und die Unregelmässigkeit der Anzahl Zerfälle in einem bestimmten Beobachtungsintervall verliert.

Für die Simulation löst man die Zeit in einzelne kleine Zeitschritte $dt = 10$ ms auf und unterwirft jedes einzelne (noch nicht zerfallenes) Isotop gemäss der Zerfallswahrscheinlichkeit einem möglichen Zerfall. Die Population wird in einer Liste z mit den Werten 0 für ein zerfallenes und 1 für eine noch nicht zerfallenes Isotop modelliert. Es lässt sich sogar das Knacken eines Zählgeräts simulieren, wenn ein vom Kern ausgesendetes γ -Quant registriert wird

```
# rad1.py

from soundsystem import *
import random
import time

def decay():
    global N
    for i in range(N0):
        if z[i] == 1:
            if random.random() < p * dt:
                z[i] = 0
                play()
    N = sum(z)

openSoundPlayer("wav/click.wav")
k = 0.005 # Decay constant (/s)
N0 = 1000 # Starting number of isotops
N = N0 # Current number of isotopes
z = [1] * N0 # Population
dt = 0.01 # Time interval for population check (s)

while True:
    currentTime = time.clock()
    decay()
    while time.clock() - currentTime < dt:
        pass
```

[Ausführen](#) mit Webstart

Man hört das typische unregelmässige Knacken des radioaktiven Strahlung.

2.3.2 Statistik des radioaktiven Zerfalls

In einer typischen experimentellen Anordnung wird mit einem elektronischen Zähler gezählt, wieviele Nuklide in einer festen Messzeit, beispielsweise 1 Sekunde zerfallen. Dabei ist nicht nur der Mittelwert interessant, sondern auch die Häufigkeitsverteilung. In der Simulation wird die Verteilung laufend aufgezeichnet.

```
# rad2.py
```

```

from soundsystem import *
import random
import time
from gpanel import *

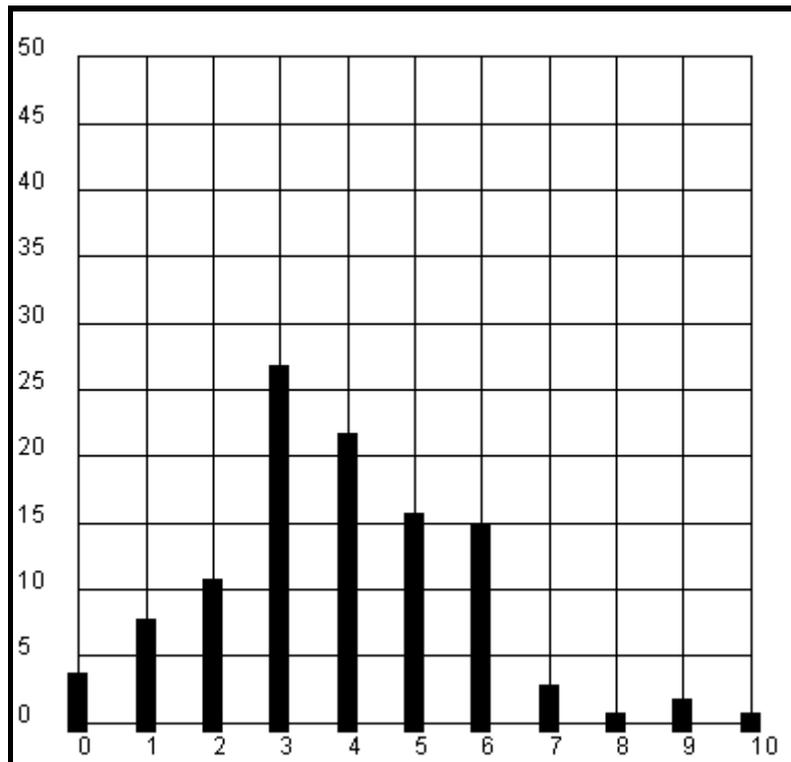
def decay():
    for i in range(N0):
        if z[i] == 1:
            if random.random() < k * dt:
                z[i] = 0
                play()

def updateDistribution():
    global N
    decayed = N - sum(z)
    if decayed <= 100:
        h[decayed] += 1
    for i in range(11):
        line(i, 0, i, h[i])
    N = sum(z)

openSoundPlayer("wav/click.wav")
makeGPanel(-1, 11, -5, 55)
drawGrid(0, 10, 0, 50)
lineWidth(10)
title("Number of decays in 1 sec")
p = 0.005      # Decay constant (/s)
N0 = 1000     # Starting number of isotopes
N = N0        # Current number of isotopes
z = [1] * N0  # Population
dt = 0.01     # Time interval for population check (s)
t = 0         # Current time
tSec = 0      # Current seconds
h = [0] * 101 # Decay distribution

while max(h) < 50:
    currentTime = time.clock()
    decay()
    tSecNew = int(t)
    if tSecNew != tSec: # every second
        updateDistribution()
        tSec = tSecNew
    while time.clock() - currentTime < dt:
        pass
    t += dt

```



[Ausführen](#) mit Webstart

Da die Population hier klein ist, spielt es eine grosse Rolle, dass die Zahl der Individuen im Lauf der Zeit abnimmt.

Es zeigt sich zwar eine glockenförmige Verteilung, die allerdings keiner bekannten Verteilung aus der Statistik entspricht, da die Zahl der Nuklide mit der Zeit abnimmt. Anders ist es, wenn man von einer grossen Zahl von Radionukliden mit kleiner Zerfallswahrscheinlichkeit ausgeht. In diesem Fall ergibt sich bekanntlich eine Poisson-Verteilung.

```
# rad3.py

import random
from gpanel import *

def decay():
    decayed = 0
    for i in range(N0):
        if random.random() < k * dt:
            decayed += 1
    return min(100, decayed)

def updateDistribution(decayed):
    h[decayed] += 1
    for i in range(101):
        line(i, 0, i, h[i])

def getMean():
    total = 0
    for i in range(101):
        total += h[i] * i
    return total / sum(h)
```

```

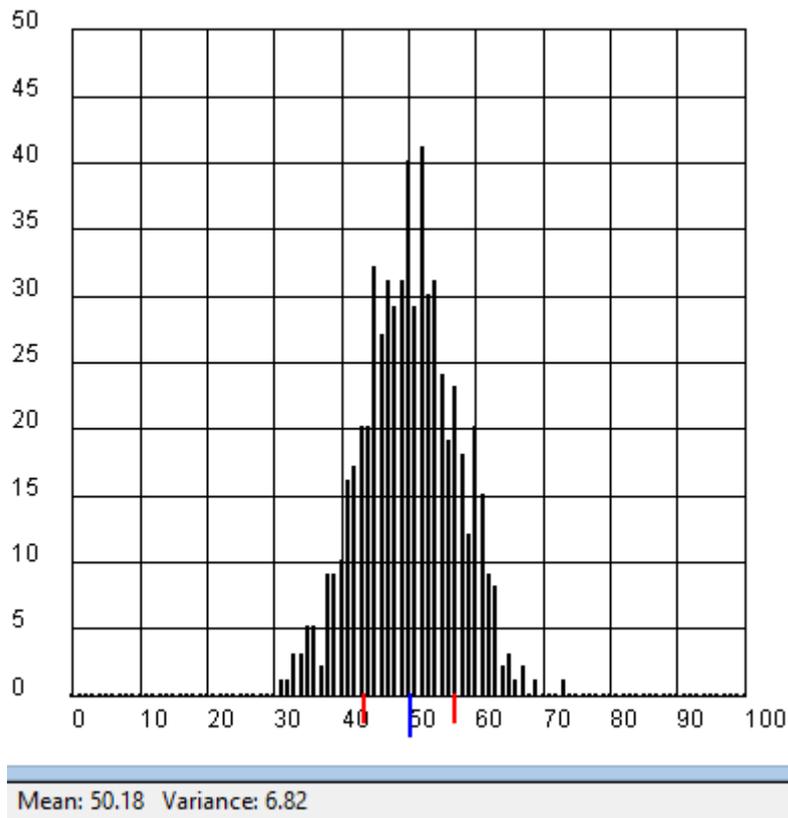
def getVariance(m):
    u = sum(h)
    total = 0
    k = int(m)
    while True:
        total += h[k]
        k += 1
        if total > 0.34 * u:
            break
    return k - m

makeGPanel(-10, 110, -5, 55)
drawGrid(0, 100, 0, 50)
lineWidth(2)
addStatusBar(20)
setStatusText("Running...")
title("Number of decays in 0.1 sec")
k = 0.05      # Decay constant (/s)
N0 = 10000    # Number of isotopes
dt = 0.1      # Time interval for population check (s)
t = 0         # Current time
h = [0] * 101 # Decay distribution

while t < 60:
    decayed = decay()
    updateDistribution(decayed)
    t += dt

m = getMean()
s = getVariance(m)
setColor("blue")
line(m, -3, m, 0)
setColor("red")
line(m - s, -2, m - s, 0)
line(m + s, -2, m + s, 0)
setStatusText("Mean: %0.2f" % m + "    Variance: %0.2f" % s)

```



[Ausführen](#) mit Webstart

Der theoretische Mittelwert beträgt $m = N_0 * \lambda * \Delta t$ und die Standardabweichung $s = \sqrt{m}$.

2.3.3 Zeitlicher Verlauf der Zahl radioaktiver Nuklide

Im nächsten Computereperiment wird der Verlauf der Zahl der Nuklide im Laufe der Zeit untersucht und grafisch dargestellt. Es zeigt sich innerhalb der statistischen Schwankungen ein typischer exponentieller Abfall.

```
# rad4.py

from soundsystem import *
from gpanel import *
import random
import time

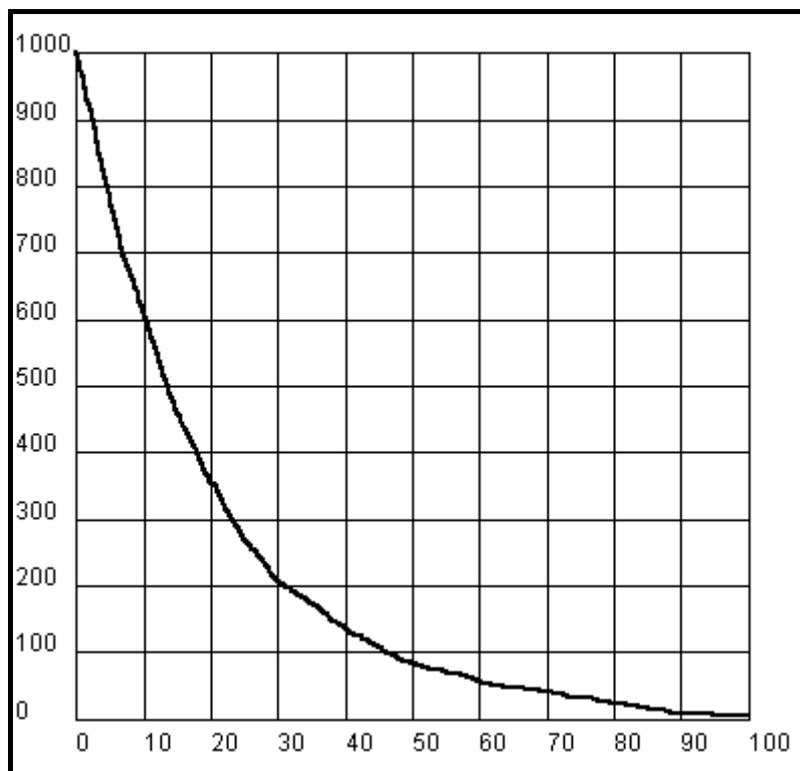
def decay():
    global N
    for i in range(N0):
        if z[i] == 1:
            if random.random() < k * dt:
                z[i] = 0
                play()
    N = sum(z)

openSoundPlayer("wav/click.wav")
makeGPanel(-10, 110, -100, 1100)
drawGrid(0, 100, 0, 1000)
```

```

title("Radioaktiver Zerfall (N versus t)")
lineWidth(2)
k = 0.05      # Decay constant (/s)
N0 = 1000    # Starting number of isotops
z = [1] * N0 # Population
dt = 0.01    # Time interval for population check (s)
t = 0        # Current time
move(0, N0)
while t <= 100:
    currentTime = time.clock()
    decay()
    draw(t, N)
    while time.clock() - currentTime < dt:
        pass
    t += dt
title("All done")

```



[Ausführen](#) mit Webstart

2.3.4 Radioaktives Zerfallsgesetz

Bei der mathematische Behandlung des radioaktiven Zerfalls betrachtet man den zeitlichen Mittelwert und schreibt für die Abnahme dN der Zahl der Nuklide im infinitesimalen Zeitintervall dt :

$$dN = -N * \lambda * dt$$

Statt die daraus folgende Differentialgleichung

$$\frac{dN}{dt} = -N * \lambda$$

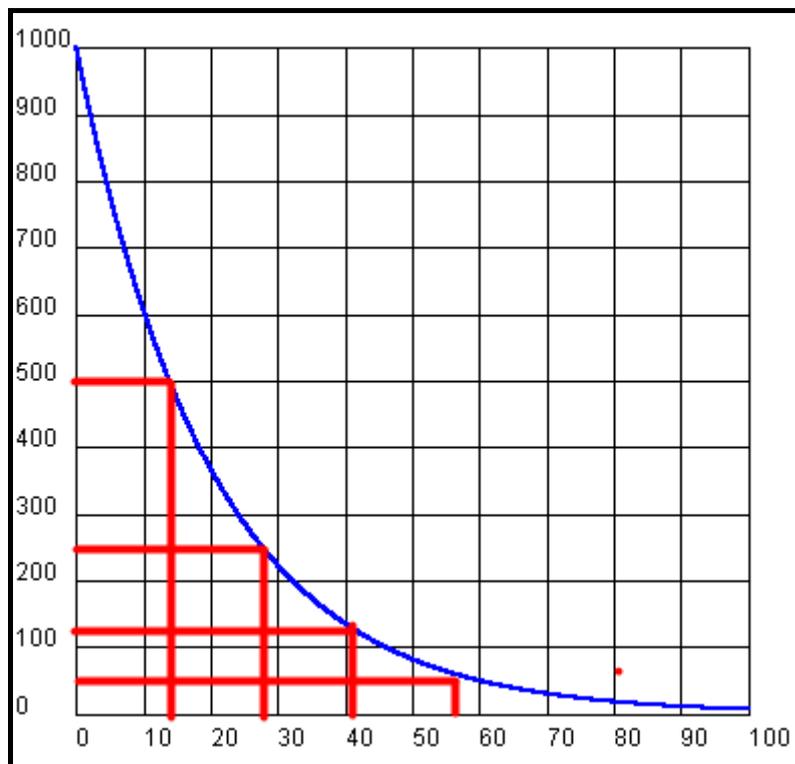
zu lösen, eine Computeriteration ausgeführt. Aus der Grafik erkennt man die Eigenschaft, dass immer nach der Halbwertszeit $T_{1/2} = \ln(2) / \lambda = 0.693 / \lambda = 13.86$ s die Zahl der Nuklide auf die Hälfte abgefallen ist.

```
# rad5.py

from gpanel import *

k = 0.05      # Decay constant (/s)
N0 = 1000    # Starting number of isotops
N = N0       # Current number of isotopes
dt = 0.01    # Time interval for population check (s)
t = 0        # Current time

makeGPanel(-10, 110, -100, 1100)
drawGrid(0, 100, 0, 1000)
title("Radioaktiver Zerfall (N versus t)")
setColor("blue")
lineWidth(2)
move(0, N)
while t <= 100:
    dN = -N * k * dt
    N = N + dN
    draw(t, N)
    t += dt
title("All done")
```



[Ausführen](#) mit Webstart

Zeichnet man zusätzlich den Funktionsgraph von

$$N = N_0 * e^{-\lambda t} \quad (\text{Radioaktives Zerfallsgesetz})$$

ein, so ergibt sich eine gute Übereinstimmung.

2.3.4 Das Mutter-Tochter-Problem

Beim klassischen Mutter-Tochter-Problem geht man davon aus, dass ein erstes Radionuklid (Mutter) mit der Zerfallskonstanten λ_1 zerfällt, aber das Zerfallsprodukt (Tochter), deren Nuklidzahl vorerst null ist, wiederum radioaktiv ist und mit der Zerfallskonstanten λ_2 zerfällt. Man untersucht den Verlauf des Mutter- und Tochterisotops. Im (infinitesimal) kleinen Zeitschritt dt gilt:

$$dN_1 = -\lambda_1 * N_1 * dt \quad \text{und} \quad dN_2 = \lambda_1 * N_1 * dt - \lambda_2 * N_2 * dt$$

Die daraus folgenden Differentialgleichungen sind für die Schule zu kompliziert, die Simulation ist aber nur eine kleine Erweiterung des vorhergehenden Problems.

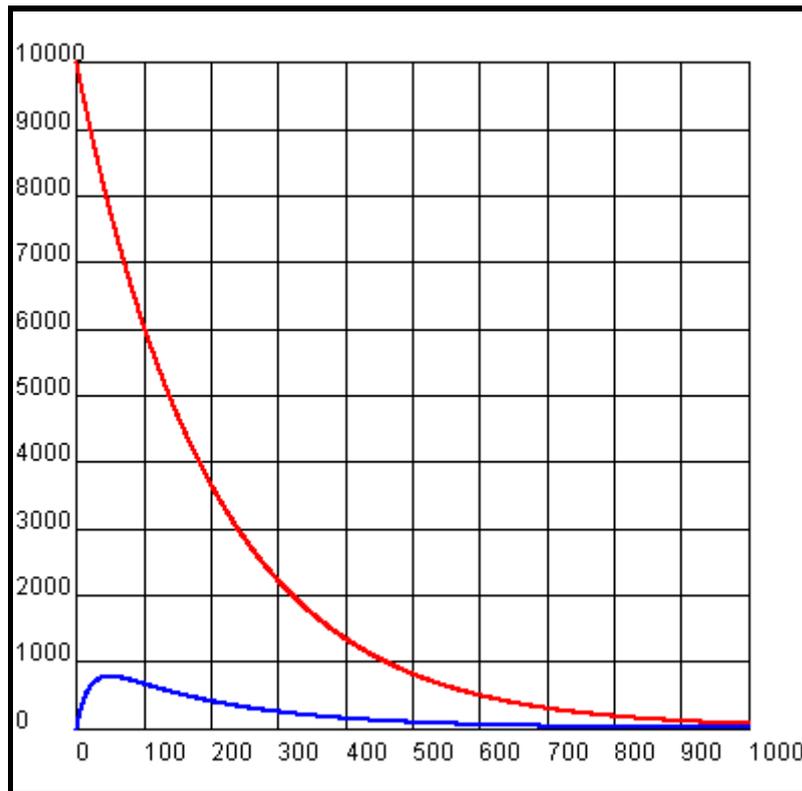
```
# rad6.py

from gpanel import *

k1 = 0.005 # Decay constant mother (/s)
N01 = 1000 # Starting number of mother isotops
N1 = N01 # Current number of mother isotopes
k2 = 0.05 # Decay constant daughter (/s)
N02 = 0 # Starting number of daughter isotops
N2 = N02 # Current number of daughter isotopes
dt = 0.01 # Time interval for population check (s)
t = 0 # Current time

makeGPanel(-100, 1100, -100, 1100)
drawGrid(0, 1000, 0, 1000)
title("Mutter-Tochter-Problem (N versus t)")
lineWidth(2)

while t <= 1000:
    N1New = N1 - N1 * k1 * dt
    N2New = N2 + N1 * k1 * dt - N2 * k2 * dt
    setColor("blue")
    line(t, N1, t, N1New)
    setColor("red")
    line(t, N2, t, N2New)
    N1 = N1New
    N2 = N2New
    t += dt
title("All done")
```



[Ausführen](#) mit Webstart

2.4 Thermodynamik

2.4.1 Verdunstungskälte

Der früher zur Narkose eingesetzte Äther (Diethylether) ist farblos und besitzt einen süßlichen Geruch. Bei Zimmertemperatur ist er flüssig (Siedetemperatur bei Normaldruck 35 °C). Er dient im Physikunterricht zur Demonstration der Verdunstungskälte. Giesst man ein paar Tropfen auf die Handfläche, so spürt man eine deutliche Abkühlung. Für die quantitative Behandlung geht man davon aus, dass bei der Verdunstung der Masse dm der Flüssigkeit die Verdampfungswärme $dQ = L_v dm$ entzogen wird und sich darum die Temperatur gemäss der Wärmekapazität $dQ = c m dT$ verringert. Es gilt daher

$$L_v * dm = c * m * dT \quad (L_v: \text{spez. Verdampfungswärme, } c: \text{spez. Wärmekapazität})$$

In einem isolierenden Behälter mit vernachlässigbarer Wärmekapazität befinden sich 2 kg Aether. Wir simulieren die Abnahme der Temperatur bis die Hälfte davon verdunstet ist (L_v und c werden als konstant angenommen).

```
# aether.py

from gpanel import *

Lv = 3.84E5      # Verdampfungswaerme von Aether in J/kg
c = 2.31E3      # Waermekapazitaet von Aether in J/kg/K
```

```

m0 = 2.0      # Anfangsmasse in kg
m1 = 1.0      # Endmasse in kg
dm = 0.001   # Massenschritt in kg
m = m0       # Aktuelle Masse

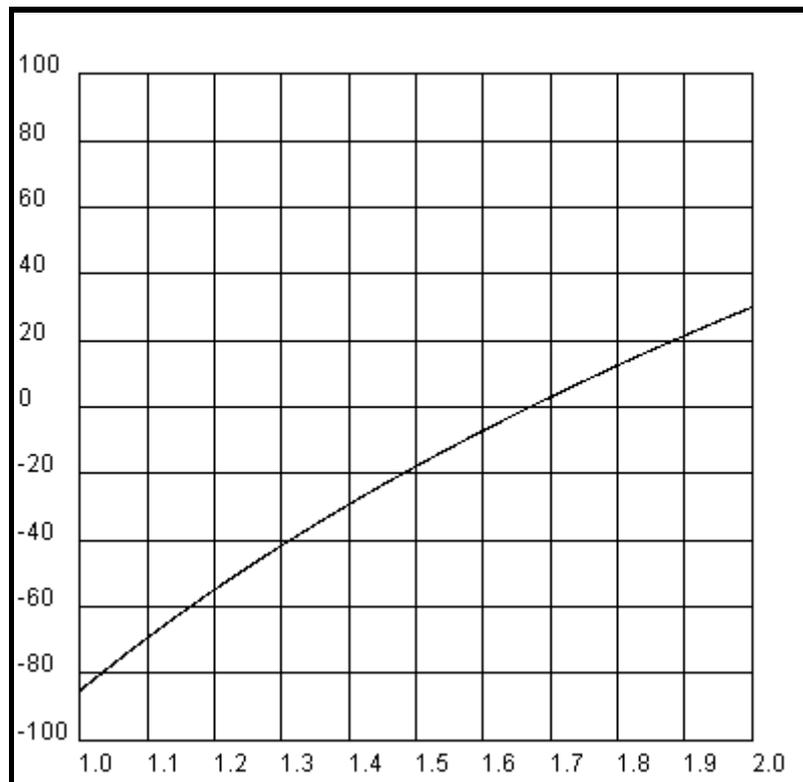
T0 = 30      # Anfangstemperatur in Grad Celsius
T = T0       # Aktuelle Temperatur in Grad Celsius

makeGPanel(0.9, 2.1, -120, 120)
drawGrid(1.0, 2.0, -100, 100)
title("Verdampfung von Aether (T versus m)")

move(m0, T0)
while m > m1:
    draw(m, T)
    dT = Lv * dm / c / m
    m -= dm
    T -= dT

print "Endtemperatur T = ", T, "°C"

```



[Ausführen](#) mit Webstart

2.4.2 Thermische Ausdehnung beim Quecksilber-Thermometer

Fast alle Festkörper und Flüssigkeiten dehnen sich mit zunehmender Temperatur wegen der thermischen Bewegung der Atome aus. Für Festkörper definiert man den (linearen) Ausdehnungskoeffizienten α mit

$$\frac{dL}{L} = \alpha(T) * dT .$$

In Worten ausgedrückt entspricht der Ausdehnungskoeffizient α der relativen Längenänderung pro Grad. Er ist von der Temperatur abhängig, aber für kleine Temperaturänderungen ΔT kann er in erster Näherung als konstant betrachtet werden und es folgt das bekannte Längenausdehnungsgesetz

$$\Delta L = \alpha * L * \Delta T .$$

Entsprechend gilt für die Volumenausdehnung von Festkörpern und Flüssigkeiten

$$\Delta V = \beta * V * \Delta T$$

mit dem Volumenausdehnungskoeffizienten $\beta = 3 \alpha$.

Eine interessante Anwendung aus dem täglichen Leben ist das Quecksilber-Glas-Thermometer, das aus einem Reservoir und einem daran verbundenen zylindrischen Rohr besteht, die aus Glas sind. Mit zunehmender Temperatur dehnt das Quecksilber aus und die Quecksilbersäule steigt nach oben. Da sich aber auch das Glas ausdehnt, ist man aber bei dieser Feststellung gar nicht so sicher. Eine Computersimulation bringt hier Klarheit, denn die durchgängig theoretische Abhandlung unter Beizug eines Taschenrechners ist mühsam und eine experimentelle Untersuchung mit eigener Herstellung von Quecksilberthermometern ist für die Schule undenkbar. Hier zeigt sich wiederum exemplarisch, wie die Computersimulation gerade in Zusammenhang mit schwierigen oder gefährlichen Experimenten sinnvoll eingesetzt wird. Im Gegensatz zu tabellenartigen Lösungsansätzen ist es zudem möglich, das Experiment mit einer interaktiven Benutzeroberfläche und einer grafischen Ausgabe auszustatten, was zur Anschaulichkeit beiträgt und die Motivation erhöht.

Man löst die Aufgabe am besten schrittweise, indem man zuerst davon ausgeht, dass sich nur das Quecksilber mit einem konstanten Volumenausdehnungskoeffizienten $\beta = 131 * 10^{-6} /K$ ausdehnt. Weiter wird angenommen, dass das Reservoir kugelförmig ist und man vernachlässigt die Kugelkappe bei der Ansatzstelle des Glasrohrs. In nachfolgenden Schritten kann man die Ausdehnung des Glases berücksichtigen (auskommentierte Zeilen), wobei sich zeigt, dass der Unterschied der Resultate bei 60 °C rund 1 mm beträgt. Zuletzt könnte man auch noch die Abhängigkeit der Ausdehnungskoeffizienten von der Temperatur auf Grund von Tabellenwerten oder einer Formel einbauen.

```

# thermometer.py

from gpanel import *

# Linear expansion: Length with given temperature (degrees centigrades)
def L(T):
    return L0 * (1 + alpha * T)

# Volume expansion: Volume with given temperature (degrees centigrades)
def V(T):
    return V0 * (1 + beta * T)

# Geometry: height with given volume
def h(V):
    return (V - 4 * math.pi / 3 * R * R * R) / (r * r * math.pi)

# height with given temperature
def x(T):
    return h(V(T))

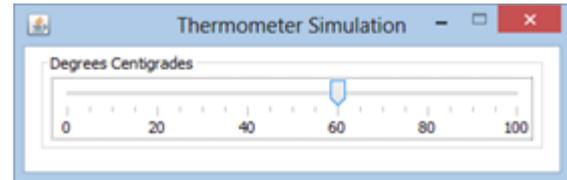
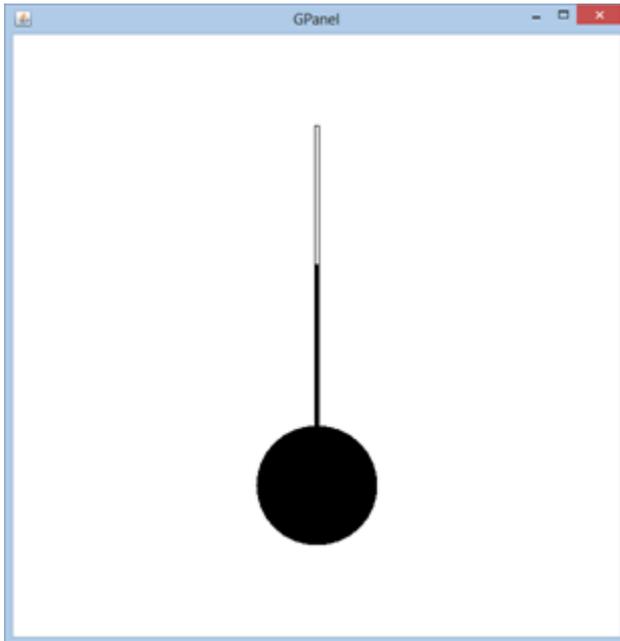
# Thermometer with given height of mercury
def drawThermometer(x):
    move(0, 0)
    fillCircle(R)
    move(0, R + H / 2)
    rectangle(r, H)
    move(0, R + x / 2)
    fillRectangle(r, x)

H = 20.0 # cm
R = 4.0 # cm at zero degrees C
r = 0.3 # cm at zero degrees C
V0 = 269 # cm^3 at zero degrees C
alpha = 4.78E-6 # Volume expansion coefficient for borosilicate glass (/K)
beta = 131E-6 # Volume expansion coefficient for mercury (/K)

makeGPanel(-20, 20, -10, 30)
drawThermometer(x(0))
slider = SliderEntry(0, 100, 0, 20, 5)
pane = EntryPane("Degrees Centigrades", slider)
dlg = EntryDialog(650, 200, pane)
dlg.setTitle("Thermometer Simulation")

while not dlg.isDisposed():
    if isDisposed():
        dlg.dispose()
    if dlg.isChanged():
        T = slider.getValue()
        print T, "°C ->", x(T), "cm"
        clear()
        # L0 = R
        # R = L(R)
        # L0 = r
        # r = L(r)
        drawThermometer(x(T))
dispose()

```



[Ausführen](#) mit Webstart

2.5 Wechselstromlehre

2.5.1 Impedanzen

Elektrische Schaltungen für sinusförmige Wechselspannungen und Wechselströme, die aus passiven Bauelementen (Widerständen, Kondensatoren, Spulen) aufgebaut sind, können wie Gleichstromschaltungen behandelt werden, wenn man für Spannungen, Ströme und Widerstände komplexe Größen verwendet. Ein allgemeiner komplexer Widerstand heisst auch **Impedanz** und wird oft mit Z und für rein imaginäre Widerstände mit X bezeichnet. Die Impedanz eines ohmschen Widerstands ist R , einer Spule $X_L = j\omega L$ (L : Induktivität) und eines Kondensators $X_C = 1 / j\omega C$ (C : Kapazität), wobei $\omega = 2\pi f$ (f : Frequenz) ist.

Eine komplexe Wechselspannung $u = u(t)$ läuft in der Gaußschen Zahlenebene gleichförmig auf einem Kreis. Wird sie an eine Impedanz Z angelegt, so fließt der Strom $i(t)$ und nach dem Ohmschen Gesetz gilt $u = Z * i$. Da bei der Multiplikation von komplexen Zahlen die Phasen addiert und die Beträge multipliziert werden, läuft u um die Phase von Z phasenverschoben vor i (bei positiver Phase von Z):

$$\text{phase}(u) = \text{phase}(Z) + \text{phase}(i)$$

Also läuft i auch auf einem Kreis. Für die Beträge (Amplituden) gilt:

$$|u| = |Z| * |i|$$

Diese Beziehung wird in der Gaußschen Zahlenebene für die Werte $|u| = 5V$, $Z = 2 + 3j$ und einer Frequenz von $f = 10$ Hz animiert dargestellt.

```

# impl.py

from gpanel import *
import math

def drawAxis():
    line(min, 0, max, 0) # Real axis
    line(0, min, 0, max) # Imaginary axis

def cdraw(z, color, label):
    oldColor = setColor(color)
    line(0j, z)
    fillCircle(0.2)
    z1 = z + 0.5 * z / abs(z) - (0.1 + 0.2j)
    text(z1, label)
    setColor(oldColor)

min = -10
max = 10
dt = 0.001

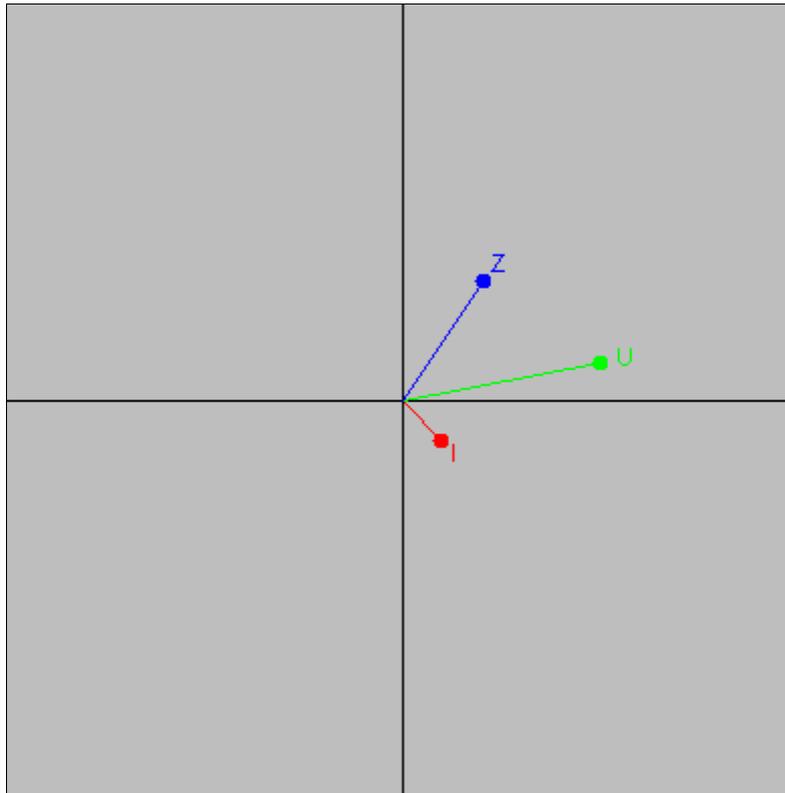
makeGPanel(min, max, min, max)
enableRepaint(False)
bgColor("gray")
title("Komplexe Spannungen und Ströme")

f = 10 # Frequency
omega = 2 * math.pi * f

t = 0
uA = 5
Z = 2 + 3j

while True:
    u = uA * (math.cos(omega * t) + 1j * math.sin(omega * t))
    i = u / Z
    clear()
    drawAxis()
    cdraw(u, "green", "U")
    cdraw(i, "red", "I")
    cdraw(Z, "blue", "Z")
    repaint()
    t += dt
    delay(100)

```



[Ausführen](#) mit Webstart

Elektrische Schaltungen mit Widerständen, Kondensatoren und Spulen können wie Gleichstromschaltungen behandelt werden, falls man Spannung, Strom und Widerstand als komplexe Zahlen (Impedanzen) auffasst. Dabei gilt (mit $\omega = 2\pi f$, f : Frequenz):

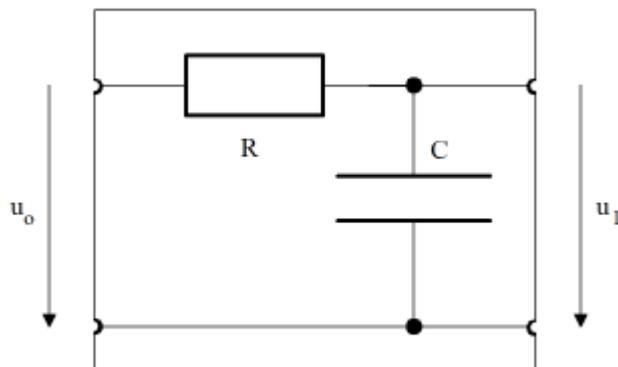
Bauelement	Grösse	Impedanz
Widerstand	R (Ohmscher Widerstand)	R
Kondensator	C (Kapazität)	$j / \omega C$
Spule	L (Induktivität)	$j \omega L$

2.5.2 Tiefpass-Filter

An ein RC-Glied legt man die Eingangswchelspannung u_o an. Wie gross ist die Ausgangspannung u_1 ?

Für die Serieschaltung von R und C ergibt sich die Impedanz $Z = R + X_C$ und damit der Strom $i = u_o / Z$, also wiederum mit dem Ohmschen Gesetz die Ausgangspannung

$$u_1 = X_C * i = \frac{X_C}{R + X_C} * u_o$$



oder

$$u_1 = v * u_o \quad \text{wo } v = \frac{X_c}{R + X_c} \quad \text{mit } X_c = \frac{j}{\omega C}$$

Im Bode-Plot wird vom komplexen Verstärkungsfaktor v Betrag und Phase als Funktion der Frequenz aufgetragen.

```
# imp2.py

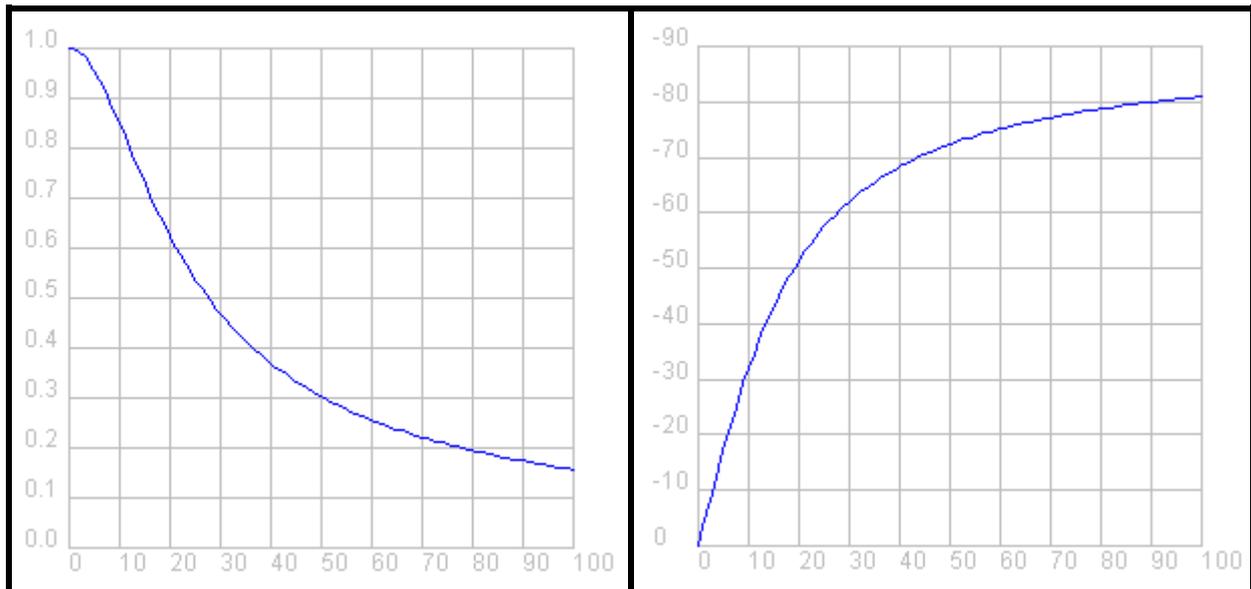
from gpanel import *
import math
import cmath

R = 10
C = 0.001

def v(f):
    if f == 0:
        return 1 + 0j
    omega = 2 * math.pi * f
    XC = 1 / (1j * omega * C)
    return XC / (R + XC)

p1 = GPanel(-10, 110, -0.1, 1.1)
drawPanelGrid(p1, 0, 100, 0, 1.0, "gray")
p1.title("Bode Plot - Low Pass, Gain")
p1.setColor("blue")
f = 0
while f <= 100:
    if f == 0:
        p1.move(f, abs(v(f)))
    else:
        p1.draw(f, abs(v(f)))
    f += 1

p2 = GPanel(-10, 110, 9, -99)
drawPanelGrid(p2, 0, 100, 0, -90, 10, 9, "gray")
p2.title("Bode Plot - Low Pass, Phase")
p2.setColor("blue")
f = 0
while f <= 100:
    if f == 0:
        p2.move(f, math.degrees(cmath.phase(v(f))))
    else:
        p2.draw(f, math.degrees(cmath.phase(v(f))))
    f += 1
```



[Ausführen](#) mit Webstart

Analog können Hochpass- und Bandpass-Filter untersucht werden.

2.6 Akustik

2.6.1 Überlagerung von Sinustönen, Schwebung

Der Computer kann als vielseitiger Audiogenerator eingesetzt werden. Damit kann man sich nicht nur die Anschaffung von Signalgeneratoren ersparen. Vielmehr lassen sich insbesondere Überlagerungsexperimente ausführen, die mit mehreren einzelnen Geräten nicht ausführbar sind, da diese nicht phasenstarr gekoppelt werden können. In TigerJython ist es nach dem Import des Moduls *soundsystem* sehr einfach, eine digitalen Audiostream auf dem linken und rechten Kanal gleichartig (mono) oder unterschiedlich (stereo) abzuspielen. Für einen monauralen Sound erstellt man dazu eine Liste mit den Abtastwerten des Sounds als 16 bit signed integers, also zwischen -32768 und 32767 und übergibt diese mit *openMonoPlayer()* einem *SoundPlayer*-Objekt. Mit *play()* wird der Sound über die Soundkarte abgespielt. Für einen binauralen Sound besteht die Liste abwechslungsweise aus dem Abtastwert für den linken und rechten Kanal und man verwendet *openStereoPlayer()*. Das Programm spielt einen Sinuston mit der Frequenz 500 Hz ab. Um es leicht zu erweitern, wird es mit den drei kommentierten Funktionen *sine()*, *createMonoSound()* und *showTimeSlice()* modularisiert. Die drei Funktionen werden in einer Datei *soundlib.py* abgespeichert, damit sie für alle weiteren Programme zur Verfügung stehen.

```
# sound1.py

from soundsystem import *
from gpanel import *

def sine(t, A, f, phi):
    ''' Sine wave
        t: current time (s)
        A: Amplitude (-33000..32000)
        f: frequency (Hz)
```

```

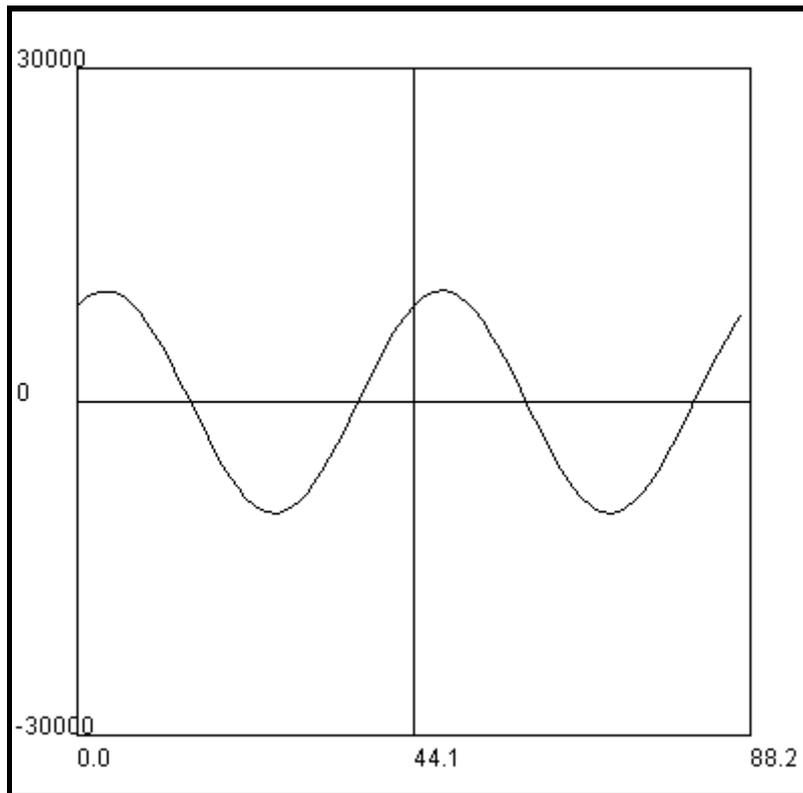
    phi: phase (degrees)
    '''
    y = A * math.sin(2 * math.pi * f * t + math.radians(phi))
    return y

def createMonoSound(g, A, f, T, fs, p):
    '''
    Returns a sound sample list:
    g: sound function g(t, A, f, p)
    A: amplitude (each sample within -32000 and 32000)
    f: fundamental frequency (Hz)
    T: duration (s)
    p: phase for each component (within 0..2pi)
    '''
    t = 0
    dt = 1.0 / fs
    nbFrames = int(T * fs)
    data = []
    for i in range(nbFrames):
        sample = int(g(t, A, f, p))
        data.append(sample)
        t += dt
    return data

def showMonoSlice(data, nbPeriods, f, fs):
    ''' Displays time trace of a mono signal
    data: sound sample list
    nbPeriods: number of periods to show
    f: signal base frequency (Hz)
    fs: sampling rate (Hz)
    '''
    makeGPanel()
    title("Signal vs Time")
    nbSamples = nbPeriods / f * fs
    window(-0.1 * nbSamples, 1.1 * nbSamples, -36000, 36000)
    drawGrid(0, nbSamples, -30000, 30000, 2, 2)
    for i in range(nbSamples):
        if i == 0:
            move(0, data[0])
        else:
            draw(i, data[i])

samplingRate = 22050 # Hz
duration = 10 # s
amplitude = 10000 # max 32000
frequency = 500 # Hz
makeGPanel()
title("Creating sound. Please wait...")
data = createMonoSound(sine, amplitude, frequency,
                      duration, samplingRate, 60)
openMonoPlayer(data, sampleRate)
play()
showTimeSlice(data, 2, frequency, samplingRate)

```



[Ausführen](#) mit Webstart

Die Überlagerung von zwei Sinusschwingungen mit fast gleicher Frequenz ergibt eine Schwebung mit einer An- und Abschwelperiode die gleich der Frequenzdifferenz $df = 1$ Hz ist.

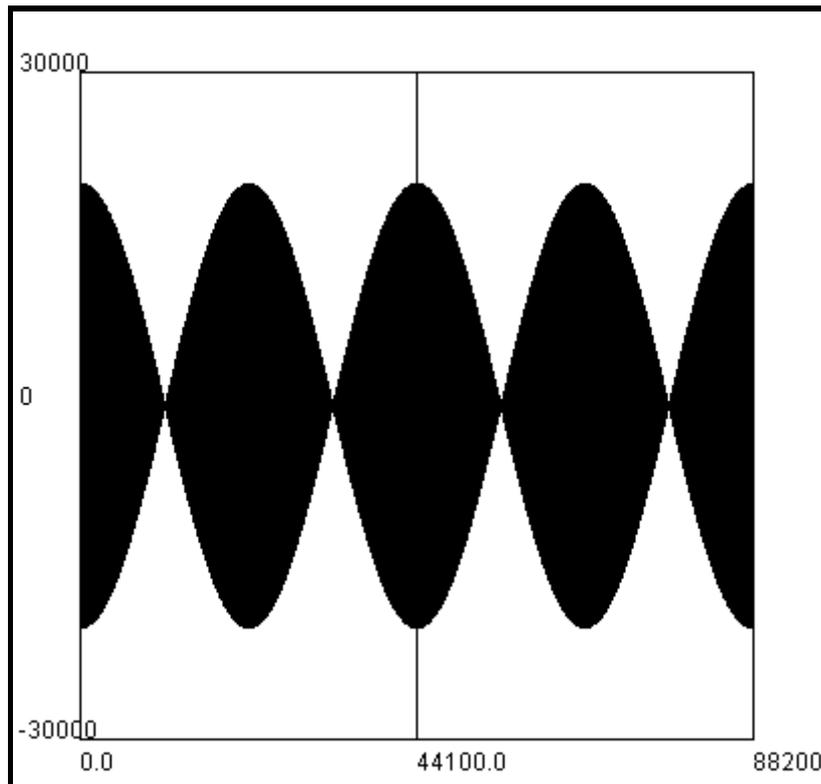
```
# sound2.py

from soundsystem import *
from gpanel import *
from soundlib import *

def tone(t, A, f, p):
    y = sine(t, A, f, 0) + sine(t, A, f + df, 0)
    return y

samplingRate = 22050 # Hz
duration = 10 # s
amplitude = 10000 # max 32000
frequency = 500 # Hz
df = 1

makeGPanel()
title("Creating sound. Please wait...")
data = createMonoSound(tone, amplitude, frequency,
                       duration, samplingRate, 60)
openMonoPlayer(data, sampleRate)
play()
showMonoSlice(data, 2000, frequency, samplingRate)
```



[Ausführen](#) mit Webstart

2.6.2 Grund- und Obertöne

2.6.2.1 Synthese von Klängen

Ein Klang ist ein periodisches Signal. Man kann dieses in einen sinusförmigen Grundton und sinusförmige Obertöne, die ein Vielfaches der Grundtonfrequenz haben, zerlegen (Fourierreihe) oder auch durch eine Überlagerung von Grund- und Obertönen synthetisieren. Hier wird der Sägezahn approximativ durch seinen Grundton und 9 Obertöne mit dem Amplituden $1/n$ erzeugt.

```
# sound3.py

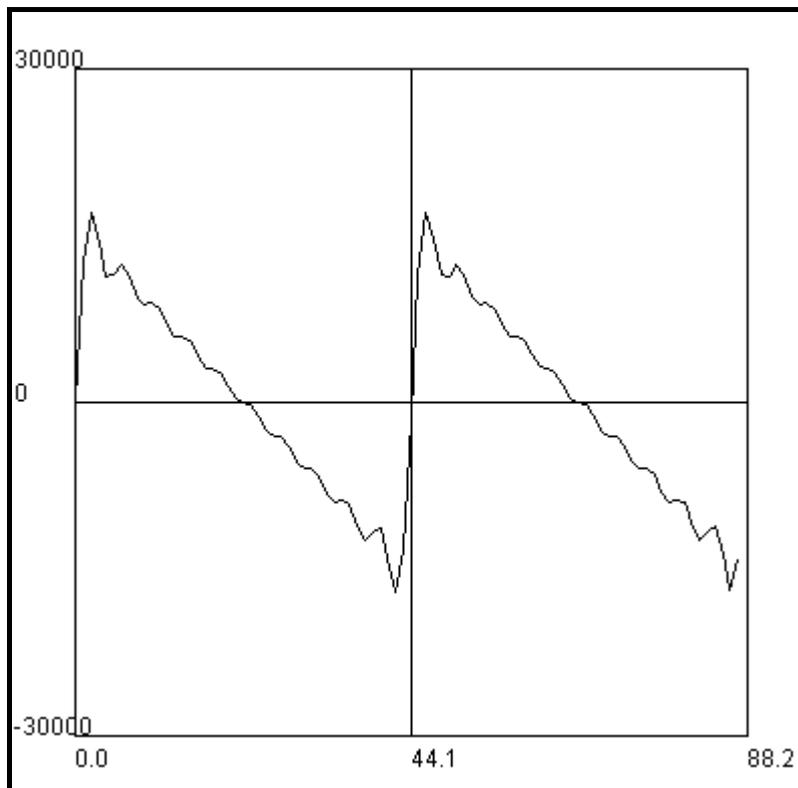
from soundsystem import *
from gpanel import *
from soundlib import *

def sound(t, A, f, p):
    y = 0
    for i in range(1, 11):
        y += sine(t, A / i, f * i, 0)
    return y

samplingRate = 22050 # Hz
duration = 10 # s
amplitude = 10000 # max 32000
frequency = 500 # Hz

makeGPanel()
title("Creating sound. Please wait...")
data = createMonoSound(sound, amplitude, frequency,
```

```
        duration, samplingRate, 0)
openMonoPlayer(data, sampleRate)
play()
showMonoSlice(data, 2, frequency, samplingRate)
```



[Ausführen](#) mit Webstart

2.6.2.2 Hörbarkeit von Phasen

Bereits seit den Untersuchungen durch H. Helmholtz ist bekannt, dass das menschliche Gehör für periodische Signale wie ein Spektrumanalyser wirkt, dass also die Amplituden von Grund- und Obertönen den Klangcharakter bestimmen, hingegen die Phasen der Obertöne keine Rolle spielt.

Seine vielzitierte Aussage lautet:

"Wir können demnach das wichtige Gesetz aufstellen, dass die Unterschiede der musikalischen Klangfarbe nur abhängen von der Anwesenheit und Stärke der Partialtöne, nicht von ihren Phasenunterschieden..." (aus H. Helmholtz, Die Lehre von den Tonempfindungen, Vieweg, 1863).

Dies ist sehr erstaunlich, da ja das Zeitsignal wesentlich verändert wird, wenn man die Phasen der Obertöne verschiebt. Im folgenden Programm werden die Phasen der Spektralanteile beim Erzeugen des Sägezahns zufällig im Bereich 0..360 Grad gewählt. Das Zeitsignal sieht damit gar nicht mehr wie ein Sägezahn aus. Trotzdem tönt der Klang genau gleich.

```
# sound4.py
from soundsystem import *
from gpanel import *
from soundlib import *
```

```

import random

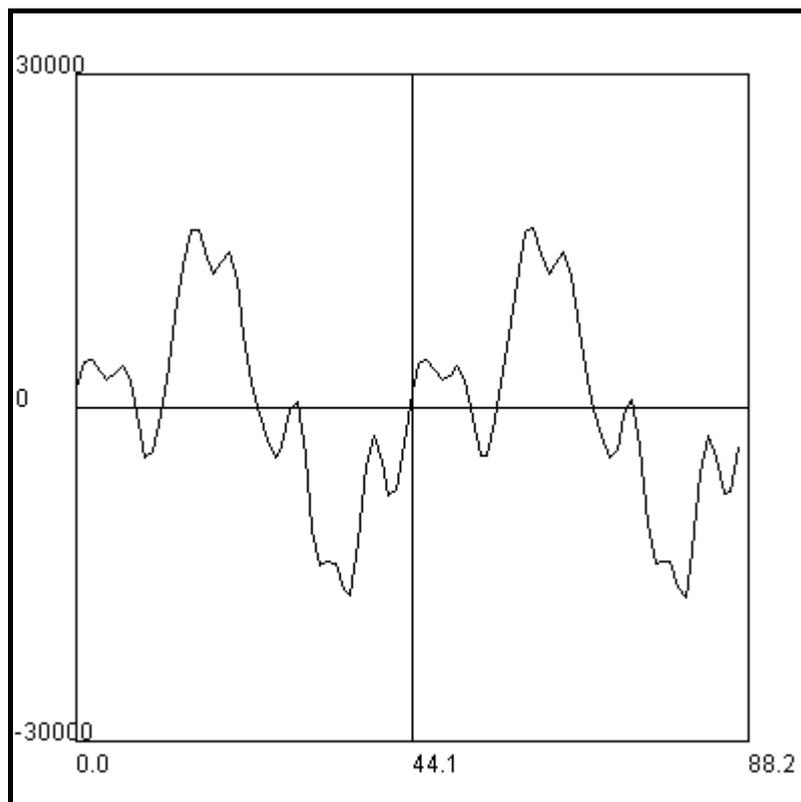
def sound(t, A, f, p):
    y = 0
    for i in range(1, 11):
        y += sine(t, A / i, f * i, phi[i])
    return y

samplingRate = 22050 # Hz
duration = 10 # s
amplitude = 10000 # max 32000
frequency = 500 # Hz

phi = [0] * 11
for i in range(1, 11):
    phi[i] = random.randint(0, 359)

makeGPanel()
title("Creating sound. Please wait...")
data = createMonoSound(sound, amplitude, frequency,
                       duration, samplingRate, 0)
openMonoPlayer(data, samplingRate)
play()
showTimeSlice(data, 2, frequency, samplingRate)

```



[Ausführen](#) mit Webstart

Mit der Funktion `saveWave()` können die Sounds auch sehr einfach auf der Festplatte gespeichert werden, damit sie mit irgendeinem Sound-Player abspielbar werden.

```

def saveWave(data, fs, filename):
    openMonoRecorder(fs)
    if writeWavFile(data, filename):

```

```
print filename + " successfully saved"
else:
    print "Can't save " + filename
```

2.6.2.3 Stereophones Hören

Obschon wir gerade vorhin feststellten, dass das Ohr normalerweise die Phasen der Obertöne nicht hört, kann es die Phaseninformation trotzdem verwerten. Trifft nämlich ein Klang von einer Schallquelle ein, die sich nicht gerade in Blickrichtung befindet, so weisen die Signale, die auf die beiden Ohren auftreffen, eine Phasenverschiebung auf, aus der sich die Richtung zur Quelle bestimmen lässt. Zudem haben sie auch noch unterschiedliche Amplituden. Das Programm erzeugt einen stereophonen Sinusklang, bei dem die Phasen des linken Kanals um 120 Grad verschoben ist. Dies entspricht einem früheren Eintreffen des linken Signals, also einer kleineren Distanz des linken Ohrs zur Schallquelle. In der Tat hört man (vorzugsweise mit einem Kopfhörer bei kleiner Lautstärke) die Schallquelle auf der linken Seite.

```
# sound5.py

from soundsystem import *
from gpanel import *
from soundlib import *

samplingRate = 22050 # Hz
duration = 10 # s
frequency = 500 # Hz

leftAmplitude = 10000
leftPhase = 120
rightAmplitude = 10000
rightPhase = 0

makeGPanel()
title("Creating sound. Please wait...")
t = 0
dt = 1.0 / samplingRate
nbFrames = int(duration * samplingRate)
data = createStereoTone(leftAmplitude, rightAmplitude, frequency,
    duration, samplingRate, leftPhase, rightPhase)
title("Time slice (2 periods)")
showStereoSlice(data, 2, frequency, samplingRate)
openStereoPlayer(data, samplingRate)
play()
```

Zur Erzeugung des stereophonen Klangs und zu seiner Darstellung wurden noch die beiden Funktionen *createStereoTone()* und *showStereoSlice()* in *soundlib.py* kopiert.

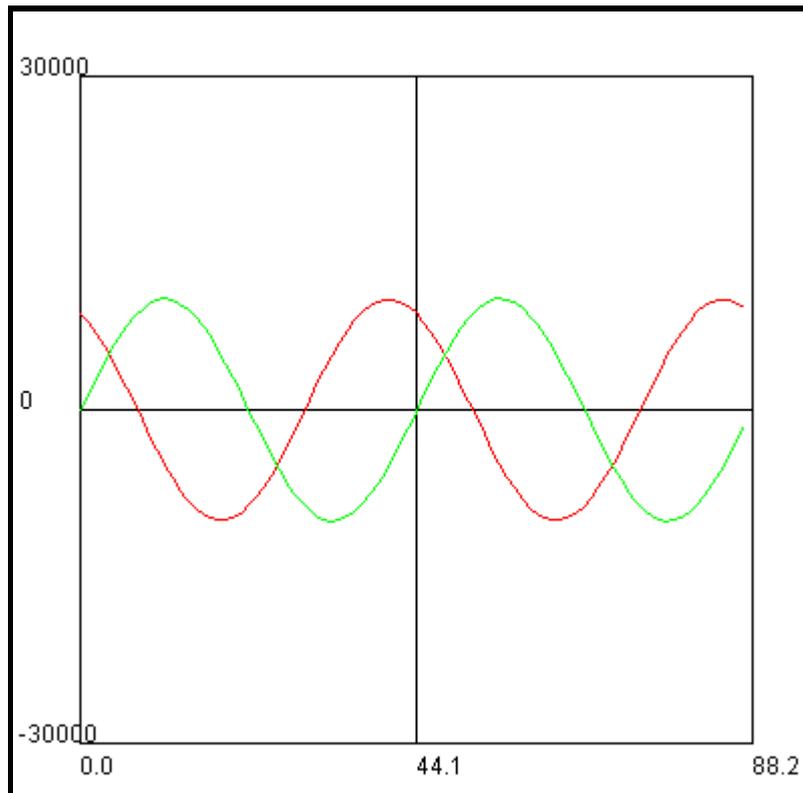
```
def createStereoTone(lA, rA, f, T, fs, lp, rp):
    """
    Creates a sound list with given:
    A: amplitude (each sample within -32000 and 32000)
    f: fundamental frequency (Hz)
    T: duration (s)
    pLeft: phase for left sine
```

```

    pRight: phase for right sine
    '''
    t = 0
    dt = 1.0 / fs
    nbFrames = int(T * fs)
    data = []
    for i in range(nbFrames):
        left = int(sine(t, lA, f, lp))
        right = int(sine(t, rA, f, rp))
        data.append(left)
        data.append(right)
        t += dt
    return data

def showStereoSlice(data, nbPeriods, f, fs):
    ''' Displays time trace of a stereo signal
        data: sound sample list
        nbPeriods: number of periods to show
        f: signal base frequency (Hz)
        fs: sampling rate (Hz)
    '''
    makeGPanel()
    title("Signal vs Time")
    nbSamples = nbPeriods / f * fs
    window(-0.1 * nbSamples, 1.1 * nbSamples, -36000, 36000)
    drawGrid(0, nbSamples, -30000, 30000, 2, 2)
    setColor("red")
    for i in range(0, nbSamples):
        if i == 0:
            move(0, data[0])
        else:
            draw(i, data[2 * i])
    setColor("green")
    for i in range(0, nbSamples):
        if i == 0:
            move(0, data[1])
        else:
            draw(i, data[2 * i + 1])

```



[Ausführen](#) mit Webstart

2.6.3 Spektralanalyse

2.6.3.1 Fast-Fourier-Transform (FFT)

In TigerJython können Soundsignale sehr einfach in ihre Spektralanteile zerlegt werden. Dazu wird die bekannte Fast Fourier Transform (FFT) verwendet. Man übergibt dazu der Funktion `fft(samples, n)` die Liste mit den (monauralen) Abtastwerten (als 16 bit signed integer) und einen Integer n , der als Ordnung der FFT bezeichnet wird (n muss gerade sein). Die Transformation verwendet dann n Abtastwerte aus `samples` und gibt $n / 2$ Amplitudenwerte des Spektrums mit dem Frequenzabstand $r = \text{samplingRate} / n$ zurück. (r nennt man die Auflösung.) Der spektrale Bereich reicht also bis zur maximalen Frequenz $f_{\text{max}} = r * n/2 = \text{samplingRate} / 2$.

Die Funktion `showSpectrum()` wird wiederum in unsere Modulbibliothek `soundlib.py` gefügt, um sie auch folgenden Programmen zugänglich zu machen.

```
def showSpectrum(a, fs, n):
    makeGPanel(-1000, 11000, -0.2, 1.2)
    drawGrid(0, 10000, 0, 1.0, 10, 5, "blue")
    title("Audio Spectrum")
    lineWidth(2)
    r = fs / n # Resolution
    f = 0
    for i in range(n // 2):
        line(f, 0, f, a[i])
        f += r
```

Das Programm analysiert einen Klang mit zwei Sinuskomponenten bei 500 Hz und 1500 Hz.

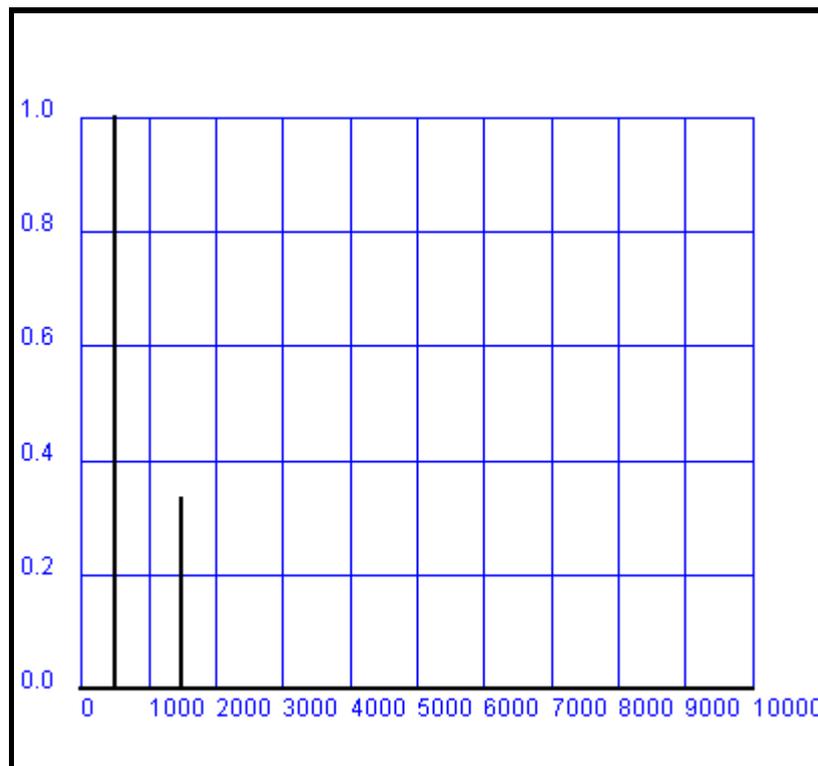
```
# sound6.py

from soundsystem import *
from gpanel import *
from soundlib import *

def sound(t, A, f, p):
    y = sine(t, A, f, 0) + sine(t, A / 3, 3 * f, 0)
    return y

samplingRate = 20000 # Hz
duration = 1 # s
amplitude = 10000 # max 32000
frequency = 500 # Hz
order = 10000

makeGPanel()
title("Creating sound. Please wait...")
data = createMonoSound(sound, amplitude, frequency,
                        duration, samplingRate, 0)
amp = fft(data, order)
showSpectrum(amp, samplingRate, order)
```



[Ausführen](#) mit Webstart

Da in TigerJython auch ein Soundrecorder eingebaut ist, kann man nun leicht Klänge, die mit einem Mikrophon aufgezeichnet werden, spektral analysieren. Beispielsweise der Gesang des Buchstabens 'a'.

```
# sound7.py
```

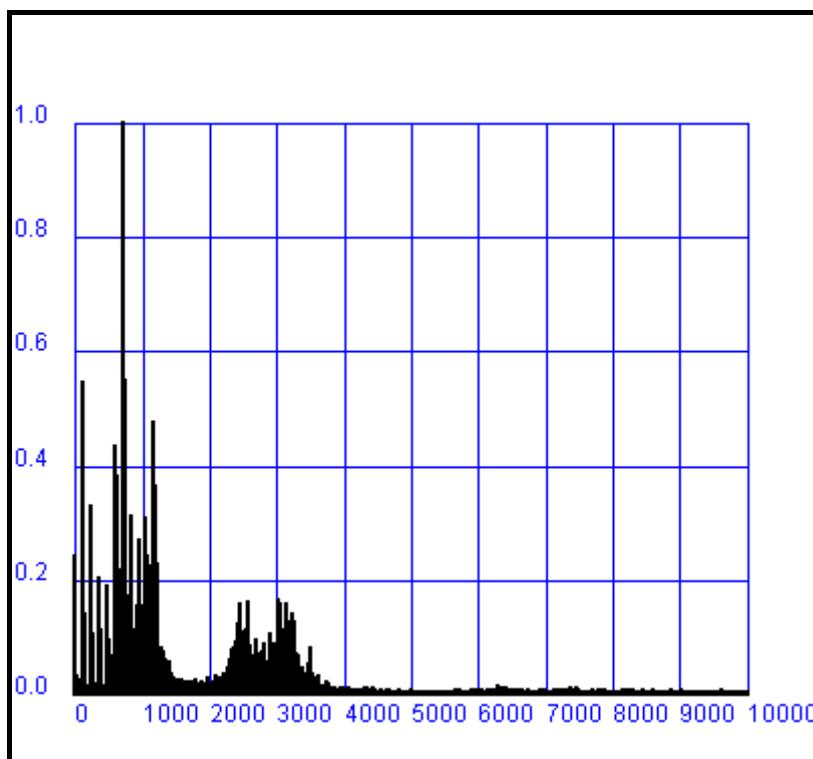
```

from soundsystem import *
from gpanel import *
from soundlib import *

samplingRate = 20000 # Hz
order = 10000
duration = 1 # s

makeGPanel()
title("Recording...")
openMonoRecorder(samplingRate)
capture()
delay(1000 * duration)
stopCapture()
data = getCapturedSound()
amp = fft(data, order)
showSpectrum(amp, samplingRate, order)

```



[Ausführen](#) mit Webstart

Gut sichtbar sind die hohen Spektralanteil im Bereich von 2 - 3 kHz. Es handelt sich um die *Formaten* der menschlichen Stimme, die wesentlich für ihre Verständlichkeit verantwortlich sind. Einmal mehr wird auf die extreme Einfachheit des Programms hingewiesen.

2.6.3.2 Sonogramm

Die FFT ist auch geeignet, Kurzzeitspektren aufzunehmen. Dabei wird das ganze Soundsignal in einzelne überlappende Blöcke mit kurzer Dauer zerlegt und in jedem Block ein Spektrum berechnet. Dieses kann in einem Sonogramm als Farbwerte aufgetragen werden. Da die Ordnung der FFT der Anzahl Abtastwerte entspricht, welche die FFT benötigt, haben mit $n = 2000$ und einer Abtastrate von 20000 Hz die

Blöcke eine Länge von 0.1 s. Sie werden überlappend im Abstand von 50 Samples analysiert.

Der Sound wird während 5 s über das Mikrofon aufgenommen. Die Analyse wird aber erst gestartet, wenn der Soundpegel einen bestimmten Wert übersteigt (Triggerung). Für die Umsetzung der Amplituden in eine Farbe, kann auf die Methode `wavelengthToColor()` zurückgegriffen werden, es lässt sich aber auch ein eigenes Farbmapping verwenden.

```
# sound8.py

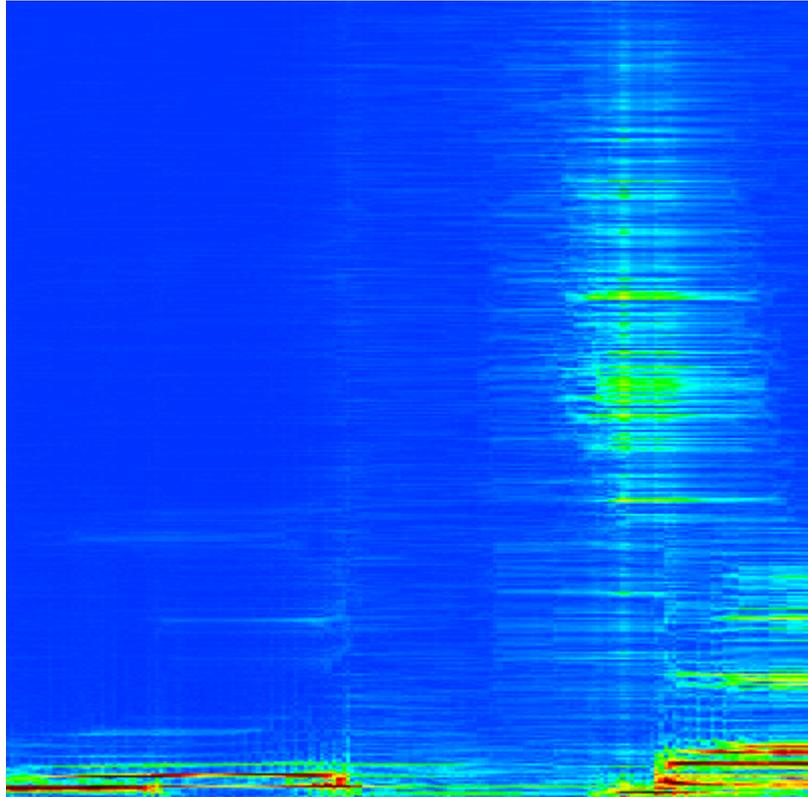
from soundsystem import *
from gpanel import *
from soundlib import *

def toColor(z):
    w = int(450 + 300 * z)
    c = X11Color.wavelengthToColor(w)
    return c

def drawSonogram(samples, n):
    makeGPanel(0, 190, 0, 1000)
    title("Sonogramm")
    lineWidth(4)
    # Analyse blocks every 50 samples
    for k in range(191):
        a = fft(samples[k * 50:], n)
        for i in range(n / 2):
            setColor(toColor(a[i]))
            point(k, i)

samplingRate = 20000
order = 2000
duration = 5 # s
triggerValue = 10000

makeGPanel()
title("Recording...")
openMonoRecorder(samplingRate)
capture()
delay(1000 * duration)
stopCapture()
data = getCapturedSound()
for i in range(len(data)):
    if data[i] > triggerValue:
        break
if i > 80000:
    title("Recording finished, but no trigger.")
else:
    print "Triggered at:", i / 20000, " s"
    drawSonogram(data[i:], order)
```



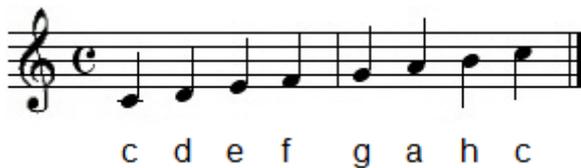
[Ausführen](#) mit Webstart

2.6.4 Tonleitern

Die wohltemperierte Tonleiter geht vom Standard-Kammerton mit der Frequenz 440 Hz aus und teilt die Oktave (Frequenzverhältnis 2) in 12 Halbtöne mit dem gleichem Frequenzverhältnis r . Demnach gilt

$$r^{12} = 2 \quad \text{oder} \quad r = \sqrt[12]{2} \approx 1.0594630943$$

Die C-Dur-Tonleiter wie folgt aus:



in Notenschrift



auf der Klaviertastatur

c-d, d-e, f-g, g-h, a-h sind Ganztonschritte (zwei Halbtonschritte), e-f, h-c sind Halbtonschritte. a ist der Kammerton. In der reinen oder natürlichen Tonleiter werden die Tonfrequenzen durch Multiplikation mit einfachen Brüchen aus dem Grundton gebildet. Für die 8 Töne einer Oktave lauten sie:

$$1, \frac{9}{8}, \frac{5}{4}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, \frac{15}{8}, 2$$

oder als Zahlenreihe 24, 27, 30, 32, 36, 40, 45, 48. Daraus ergeben sich ausgehend von a = 400 Hz die folgenden Frequenzen der einzelnen Töne:

Ton	wohltemperiert	natürlich
c	a / r^9	$3/5 * a$
d	a / r^7	$3/5 * a * 9/8$
e	a / r^5	$3/5 * a * 5/4$
f	a / r^4	$3/5 * a * 4/3$
g	a / r^2	$3/5 * a * 3/2$
a	a	a *
h	$a * r^2$	$3/5 * a * 15/8$
c	$a * r^3$	$3/5 * a * 2$

Um sie abzuspielen, kann man die Frequenzen in einer Liste speichern und diese `playTone()` übergeben. Zuerst werden die beiden Tonleitern einzeln abgespielt und man hört keinen Unterschied. Spielt man hingegen die Tonleitern gemeinsam, so tönt es wegen der auftretenden Schwebungen schrecklich.

```
r = 2**(1/12)
a = 440

scale_temp = [a / r**9, a / r**7, a / r**5, a / r**4, a / r**2,
              a, a * r**2, a * r**3]
scale_pure = [3/5 * a, 3/5 * a * 9/8, 3/5 * a * 5/4, 3/5 * a * 4/3, 3/5 *
              a * 3/2,
              a, 3/5 * a * 15/8, 3/5 * a * 2]

playTone(scale_temp, 1000)
playTone(scale_pure, 1000)

playTone(scale_temp, 1000, block = False)
playTone(scale_pure, 1000)
```

[Ausführen](#) mit Webstart

Referenzen:

Hans Bebie: [Signalanalyse](#) in der Akustik

Aegidius Plüss, [Overtone](#)

2.7 Optik

2.7.1 Brechungsgesetz

Minimumprinzipien spielen in der Natur eine grosse Rolle. Das bekannteste sagt aus, dass ein System die Tendenz hat, sich in einen möglichst tiefen Energiezustand zu

begeben. Das Brechungsgesetz $n = \frac{\sin \alpha}{\sin \beta} = \frac{c_1}{c_2}$ lässt sich bekanntlich aus einem

Minimumprinzip herleiten, das besagt, dass der Weg eines Lichtstrahls vom Punkt P_1 zu Punkt P_2 so verläuft, dass das Licht dafür die **kürzeste Zeit** braucht. Für eine theoretische Herleitung wird die Differentialrechnung vorausgesetzt, bei der Simulation sind jedoch nur elementare Kenntnisse aus der Trigonometrie nötig. Zudem ist die Simulation sehr anschaulich, da man sie leicht mit einem Läufer illustrieren kann, der zuerst auf dem schnellen Rasen mit $c_1 = 3$ m/s und dann auf dem langsamen Rasen mit $c_2 = 2$ m/s möglichst rasch von P_1 nach P_2 gelangen will. Es ist klar, dass er tendenziell möglichst lange auf dem schnellen Rasen laufen will. Zur Lösung wird der Übergangspunkt P mit der Maus so variiert, dass die Zeit minimal ist.

Das Programm präsentiert sich bereits professionell, dafür muss man allerdings einen etwas längeren Programmcode in Kauf nehmen.

```
# refraction.py

from gpanel import *
import math

def calc():
    global tmin, qmin
    s1 = math.sqrt((px - px1) * (px - px1) + py1 * py1)
    s2 = math.sqrt((px - px2) * (px - px2) + py2 * py2)
    t1 = s1 / c1
    t2 = s2 / c2
    t = t1 + t2
    alpha = abs(math.atan((px - px1) / py1))
    beta = abs(math.atan((px2 - px) / -py2))
    q = math.sin(alpha) / math.sin(beta)
    if t < tmin:
        tmin = t
        qmin = q
    return math.degrees(alpha), math.degrees(beta), q, t

def drawBoard():
    alpha, beta, q, t = calc()
    clear()
    setColor("green")
    fillRectangle(0, 50, 100, 0)
    setColor("darkgreen")
    fillRectangle(0, 0, 100, -50)
    setColor("black")
    lineWidth(4)
    line(0, 0, 100, 0)
    move(px1, py1)
    fillCircle(1)
    move(px2, py2)
    fillCircle(1)
    text(px + 1, 1, "(" + str(int(px)) + ",0)")
    move(px, py)
    if isDragging:
        setColor("red")
        fillCircle(1)
    else:
        fillRectangle(2, 2)
        lineWidth(1)
```

```

line(px, -20, px, 20)
setColor("black")
text(3, -48, "Drag square!")
text(px1, py1 + 2, "P1(10,40)")
text(px2, py2 - 3, "P2(90,-40)")
setColor("blue")
line(px1, py1, px, py)
line(px, py, px2, py2)
setStatusText("alpha = %4.1f, beta = %4.1f, \
q = sin(alpha)/sin(beta) = %4.2f, \
time = %4.4f s, tmin = %4.4f s, \
qmin = %4.2f" %(alpha, beta, q, t, tmin, qmin))

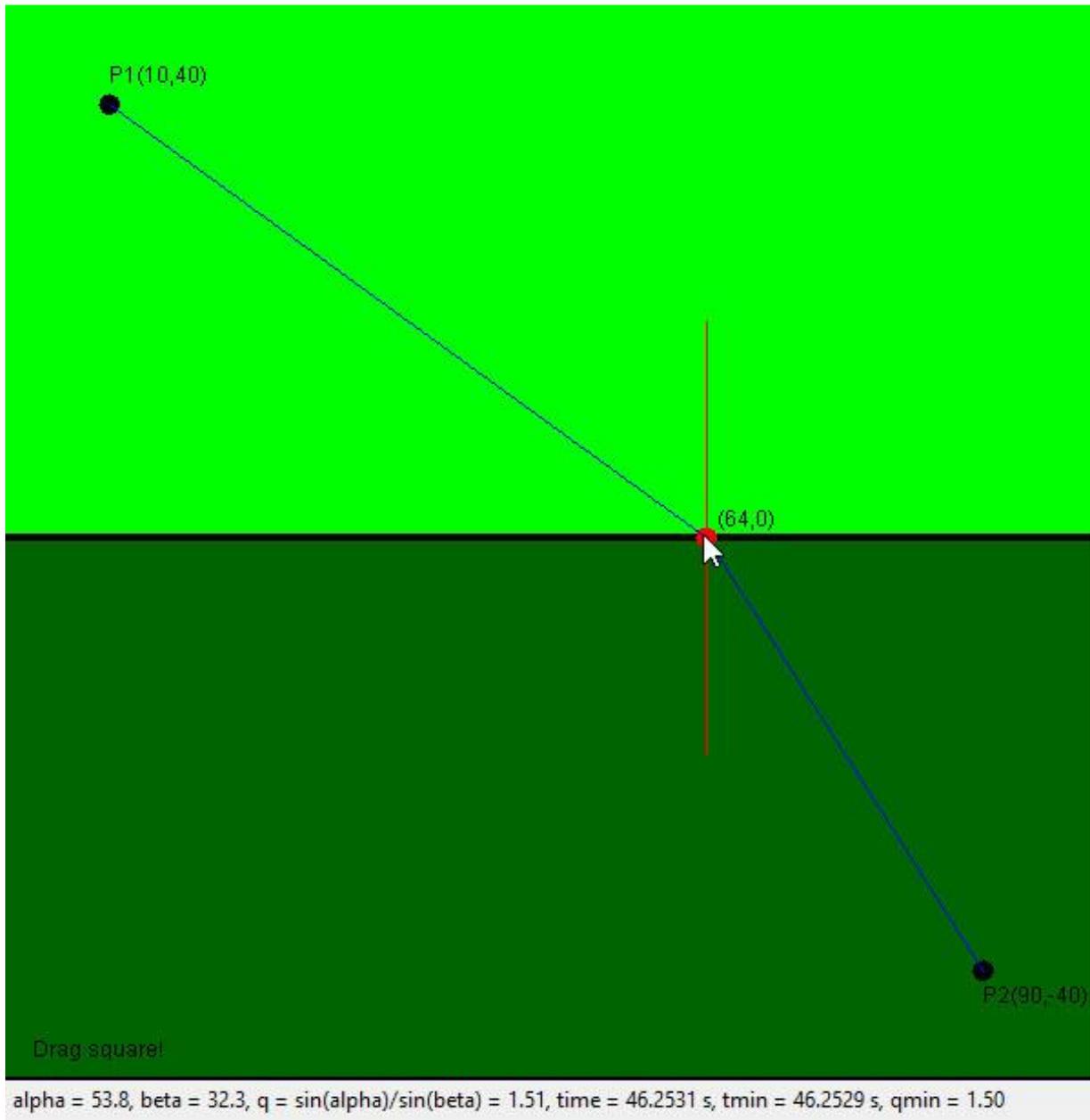
def onMousePressed(x, y):
    global isDragging
    if (x - px) * (x - px) + (y - py) * (y - py) < 2:
        isDragging = True
        drawBoard()

def onMouseDragged(x, y):
    global px
    if not isDragging:
        return
    if x > px1 + 1 and x < px2 - 1:
        px = x
        drawBoard()

def onMouseReleased(x, y):
    global isDragging
    isDragging = False
    drawBoard()

makeGPanel(0, 100, -50, 50,
           mousePressed = onMousePressed,
           mouseDragged = onMouseDragged,
           mouseReleased = onMouseReleased)
title("Law of Refraction, c1 = 3 m/s, c2 = 2 m/s, n = 1.5")
addStatusBar(30)
px1 = 10
py1 = 40
px2 = 90
py2 = -40
px = 50
py = 0
n = 1.5
c1 = 3
c2 = c1 / n
tmin = 1000
isDragging = False
drawBoard()

```



[Ausführen](#) mit Webstart

2.7.2 Interferenz

Treffen an einem bestimmten Ort Wellen ein, so überlagern sich die daraus entstehenden Schwingungen. Handelt es sich um zwei Sinuswellen gleicher Wellenlänge und sind diese gegeneinander phasenstarr, so ergibt sich eine resultierende Schwingung mit grosser Amplitude dort, wo zwei Wellenberge aufeinander treffen und eine Schwingung mit kleiner Amplitude dort, wo ein Wellenberg auf ein Wellental treffen. Man kann diese Interferenz veranschaulichen, indem man die Wellen zu einem bestimmten Zeitpunkt als schwarze Kreisscharen darstellt, wo die gezeichneten Kreise die Lage der Wellenberge darstellen. Zeichnet man zwei Kreisscharen mit einem leicht unterschiedlichen Mittelpunkt, so gibt es Gebiete, wo schwarz-weiss der beiden Scharen etwa übereinstimmen, was zu schwarz-weissen Streifen führt (Interferenzmaxima) und solche, wo schwarz der einen Schar auf weiss der anderen (und umgekehrt) fallen, was zu schwarzen

Gebieten führt (Interferenzminima). Um die Pixelstruktur zu verfeinern, ist es von Vorteil, wenn die Grafik als hochauflösender Druckerprint gerendert wird, beispielsweise auf einem virtuellen Drucker (zum Beispiel als PDF).

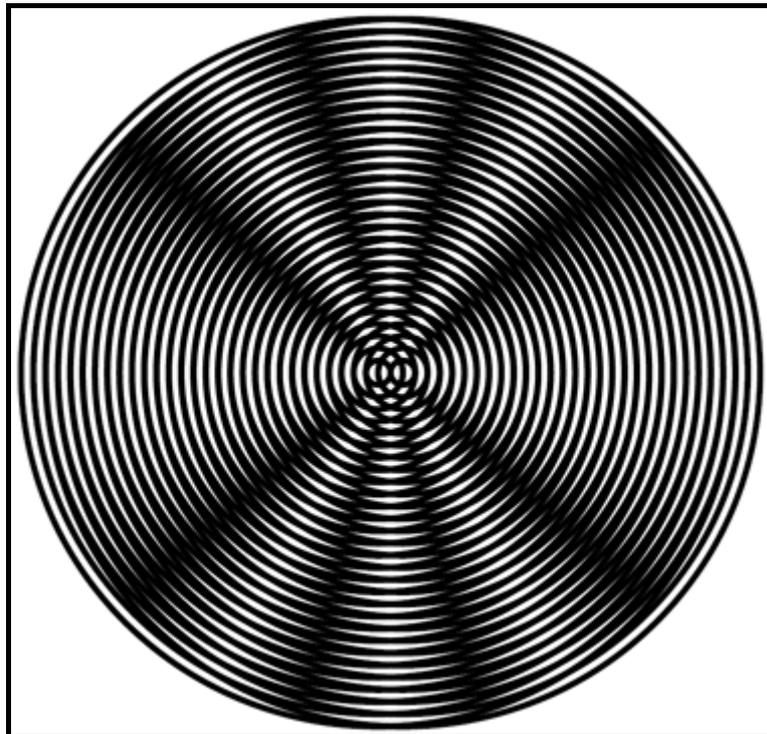
```
#interference.py

from gpanel import *

def wave(center, wavelength):
    nbCrests = 30
    lineWidth(wavelength // 2)
    move(center)
    for i in range(nbCrests):
        circle(i * wavelength)

def doIt():
    wave([250, 250], w)
    wave([250 + d, 250], w)

w = 8      # wavelength
d = 2 * w # separation
makeGPanel(Size(501, 501))
window(0, 500, 0, 500)
doIt()
printerPlot(doIt)
```



[Ausführen](#) mit Webstart

2.8 Zufallsexperimente

2.8.1 Random Walk

Mathematiker und Physiker kleiden das Problem oft in folgende, nicht allzu erstzunehmende Geschichte ein. Darum heisst das Problem auch Drunken Man's Walk:

"Ein Betrunkener versucht nach einer Wirtshaustour nach Hause zurückzukehren. Da er die Orientierung verloren hat macht er dabei immer einen Schritt in zufälliger Richtung. Wie weit bewegt er sich dabei (im Mittel) vom Wirtshaus weg?"

Der Betrunkene wird hier durch die Turtle gespielt, die gleich lange Schritte in zufälliger Richtung macht. Man wählt eine bestimmte feste Schrittzahl t (was bei fester Schrittperiode einer bestimmten Zeit entspricht). Mit diesem t führt man die Simulation 1000x aus und bestimmt zuletzt die mittlere Entfernung d . Die Zeit t wird in hunderter Schritten zwischen 100 und 1000 schrittweise erhöht, um herauszufinden, wie d von t abhängt. Dabei zeichnet man nicht den ganzen Weg, sondern nur die Endposition der Turtle bei jedem Simulationslauf als Punkt ein. Gleichzeitig trägt man d und t in einem GPanel-Diagramm auf.

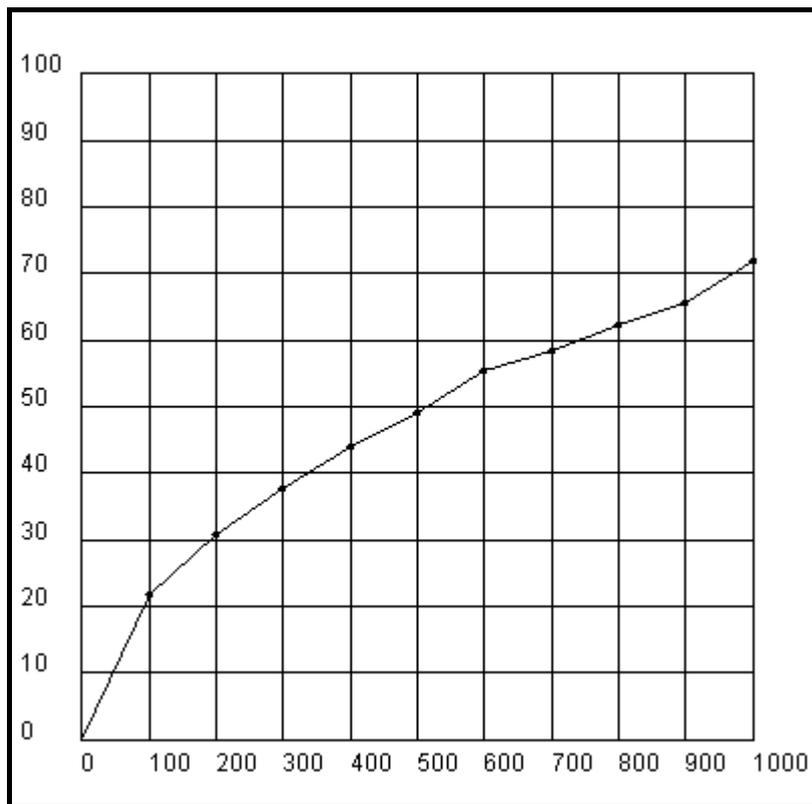
```
from gturtle import *
from gpanel import *
import math

makeTurtle()
makeGPanel(-100, 1100, -10, 110)
windowPosition(850, 10)
drawGrid(0, 1000, 0, 100)
title("Mean distance versus time")
ht()
pu()

for t in range(100, 1100, 100):
    sum = 0
    clean()
    repeat 1000:
        repeat t:
            fd(2.5)
            setRandomHeading()
        dot(3)
        r = math.sqrt(getX() * getX() + getY() * getY())
        sum += r
        home()
    d = sum / 1000
    print "t =", t, "d =", d, "q = d / sqrt(t) =", d / math.sqrt(t)
    draw(t, d)
    fillCircle(5)
    delay(2000)
```

```
print "all done"
```

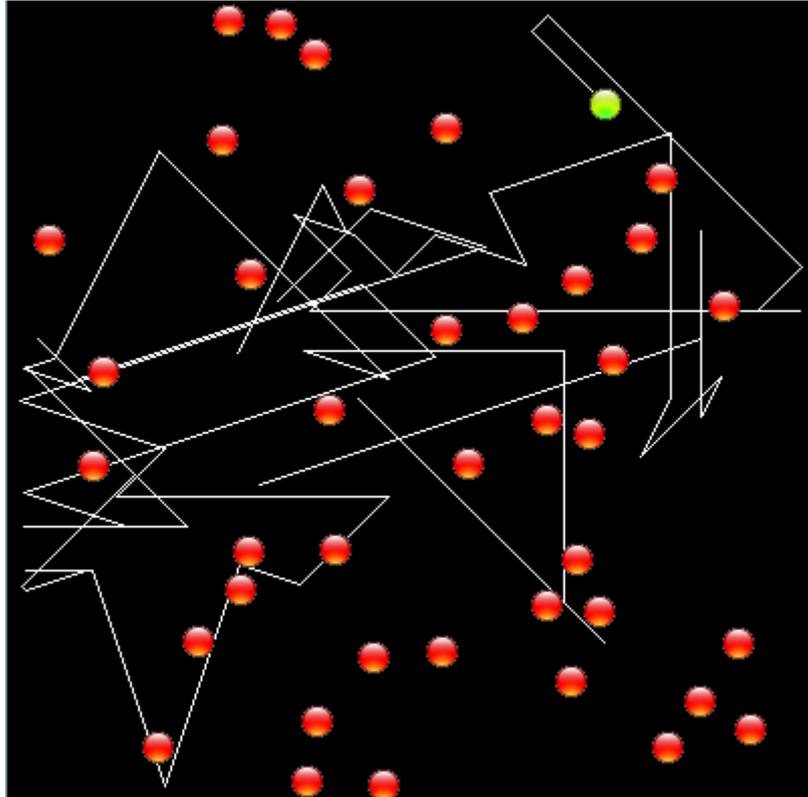
Da der Quotient $q = d / \sqrt{t}$ ungefähr konstant bleibt, nimmt offenbar der mittlere Abstand mit der Wurzel aus der Zeit zu. Der Betrunkene hat also trotz zufälliger Bewegungsrichtungen sehr wohl eine Chance, einmal nach Hause zu kommen. Diese Gesetzmässigkeit hat kein geringerer als Albert Einstein 1905 in seiner berühmten Abhandlung "Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen" hergeleitet, und damit eine theoretische Erklärung für die Brownsche Bewegung gegeben. Ein Jahr später hat M. Smoluchowski mit einer anderen Überlegung dasselbe Resultat gefunden.



[Ausführen](#) mit Webstart

2.8.2 Brownsche Bewegung

Die Brownsche Bewegung lässt sich sehr schön in einem Computerexperiment demonstrieren, wo die Moleküle als kleine Kugeln modelliert werden, die beim Zusammenstoss ihre Geschwindigkeit austauschen. Mit der Gamelibrary JGameGrid fasst man die Kugeln als Actors auf, denen eine Spritebild zugeordnet ist. Die Zusammenstösse werden als Kollisionsevents aufgefasst und brauchen nicht explizit berechnet zu werden. Der Programmcode ist trotzdem etwas länger und kann [hier](#) eingesehen werden. Er ist auch in der Programmsammlung zu diesem Artikel enthalten.



[Ausführen](#) mit Webstart

2.8.3 Diffusion, Entropie

Überall in der Natur beobachten wir Phänomene, die uns nahe legen, dass ein sich selbst überlassenes System die Tendenz zeigt, von einem geordneten Zustand in einen ungeordneten überzugehen. Beispiele dafür sind:

- Passagiere in einem Eisenbahnwagen verteilen sich über den ganzen Wagen
- Zigarettenrauch verteilt sich im Zimmer
- Tintenklecks verteilt sich im Wasserglas
- Temperatur zwischen Kaffee und Tasse gleicht sich aus

Der grosse Physiker Ludwig Boltzmann hat für die Unordnung eine messbare Grösse eingeführt. Die statistische Definition der Entropie lautet:

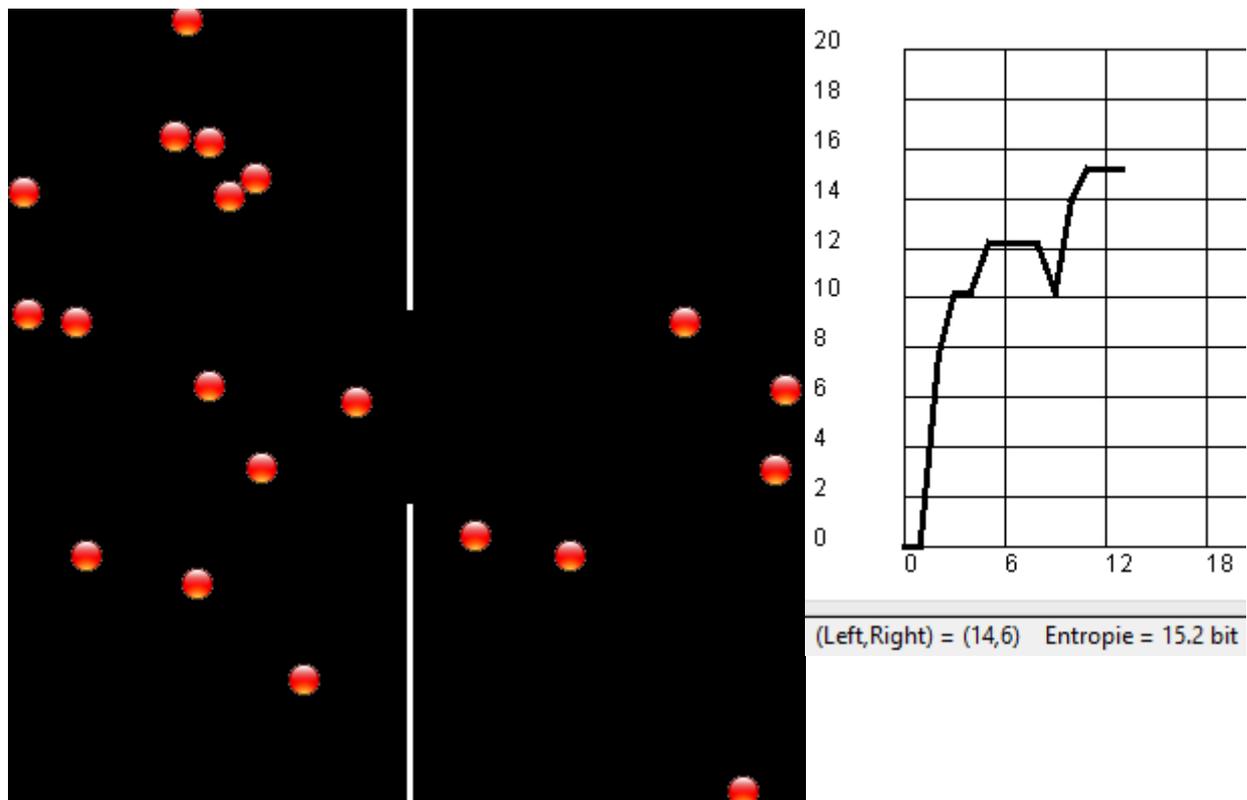
*Unter der **Entropie** (in bit) verstehen wir den Informationsmangel I , den wir bei einer makroskopischen Beschreibung des Systems gegenüber jemandem besitzen, der den mikroskopischen Zustand kennt.*

Es ist intuitiv klar, dass ein System mit grösserer Unordnung auch einen grösseren Informationsmangel, also mehr Entropie hat. Boltzmann konnte damit die Tendenz zur Unordnung als zweiten Hauptsatz der Thermodynamik) so formulieren, dass in einem sich selbst überlassenen System **die Entropie im Laufe der Zeit zunimmt**.

Eine bekannte, sehr einfache Demonstration des Zweiten Hauptsatzes ist das Ausströmen eines Gases in einem Doppelbehälter. Zuerst befinden sich alle Gasteilchen im linken Behälter. Öffnet man ein kleines Loch zwischen den Behältern,

so strömt das Gas aus und verteilt sich über beide Behälter. Es ist offensichtlich sehr unwahrscheinlich, aber nicht unmöglich, dass sich auch wieder einmal alles Teilchen im linken Behälter wiederfinden. Man spricht beim Ausströmen des Gases auch von einem **Diffusionsprozess**.

Das Programm ist sehr ähnlich wie bei der Demonstration der Brownschen Bewegung aufgebaut und kann [hier](#) eingesehen werden. Er ist ebenfalls in der Programmsammlung zu diesem Artikel enthalten.



[Ausführen](#) mit Webstart

2.9 Quantenphänomene

2.9.1 Energieniveaux, Boltzmann-Verteilung

Atomare Partikel besitzen gemäss der Quantentheorie üblicherweise ein diskretes Energiespektrum. Es ist von fundamentaler Bedeutung, dass ein System mit wechselwirkenden Partikeln, die ein nach unten begrenztes diskretes Energiespektrum haben, eine exponentielle Energieverteilung h aufweist. Es gilt die Boltzmann-Verteilung

$$h(E) = A * e^{-\beta E}$$

Aus $\beta = \frac{1}{kT}$ ergibt sich die statistische Definition der absoluten Temperatur T .

In der einfachen, aber durchaus realistischen Simulation geht man von folgenden Modellannahmen aus:

40 linear angeordnete Teilchen haben ein nach unten begrenztes Energiespektrum mit äquidistanten Energieniveaux mit den Energien 0, 1, 2,...,10. In jedem Zeitschritt wird zufällig ein Teilchen i und mit gleicher Wahrscheinlichkeit sein linker oder sein rechter Nachbar k ausgewählt. (Für das linke Randteilchen $i = 0$, ist der Nachbar $k = 1$, für das rechte Randteilchen $i = 39$ ist der Nachbar $k = 38$). Die Paare tauschen die Energie nach folgender Regel aus:

Falls sich das Teilchen i nicht im Grundzustand mit der Energie 0 befindet, so gibt es eine Energieeinheit an seinen Partner k ab, fällt also auf das nächst tiefere Energieniveau zurück. Dafür springt k auf das nächst höhere Energieniveau

Anfangsbedingungen: Zur Zeit $t = 0$ befinden sich zufällige 30 Teilchen im ersten angeregtem Niveau, alle anderen im Grundzustand.

```
# boltzmann.py
# E0 = 0, dE = 1

from gpanel import *
import random

def initSystem():
    width = (nb + 2) * d
    height = (eMax + 2) * d
    makeGPanel(Size(width, height))
    title("Particle System")
    setXORMode("white")
    window(-1, nb, -1, eMax + 1)
    n = 0
    while n < nbStart:
        k = random.randint(0, nb - 1)
        if e[k] == 0:
            e[k] = 1
            n += 1
    h[0] = nb - nbStart
    h[1] = nbStart
    drawSystem()

def drawSystem():
    # Draw energy niveaux
    for E in range(eMax + 1):
        line(-1, E, nb + 1, E)
    # Draw particles
    for i in range(nb):
        move(i, e[i])
        fillCircle(0.5)

def pair():
    i = random.randint(0, nb - 1)
    if i == 0: # left border
        k = 1 # right neighbor
    elif i == nb - 1: # right border
        k = nb - 2 # left neighbor
    else: # not at border
        k = i - 1 + 2 * random.randint(0, 1) # left or right neighbor
    if e[i] > 0 and e[k] < eMax: # i not in ground state
```

```

        # i jumps down, k jumps up
        drawPair(i, k)
        modifyDistribution(i, k)
        e[i] -= 1
        e[k] += 1
        drawPair(i, k)

def modifyDistribution(i, k):
    h[e[i]] -= 1
    h[e[k]] -= 1
    h[e[i] - 1] += 1
    h[e[k] + 1] += 1

def drawPair(i, k):
    move(i, e[i])
    fillCircle(0.5)
    move(k, e[k])
    fillCircle(0.5)

def initDistribution():
    global p
    p = GPanel(-1, 11, -5, 55)
    p.title("Energy Distribution")
    p.enableRepaint(False)

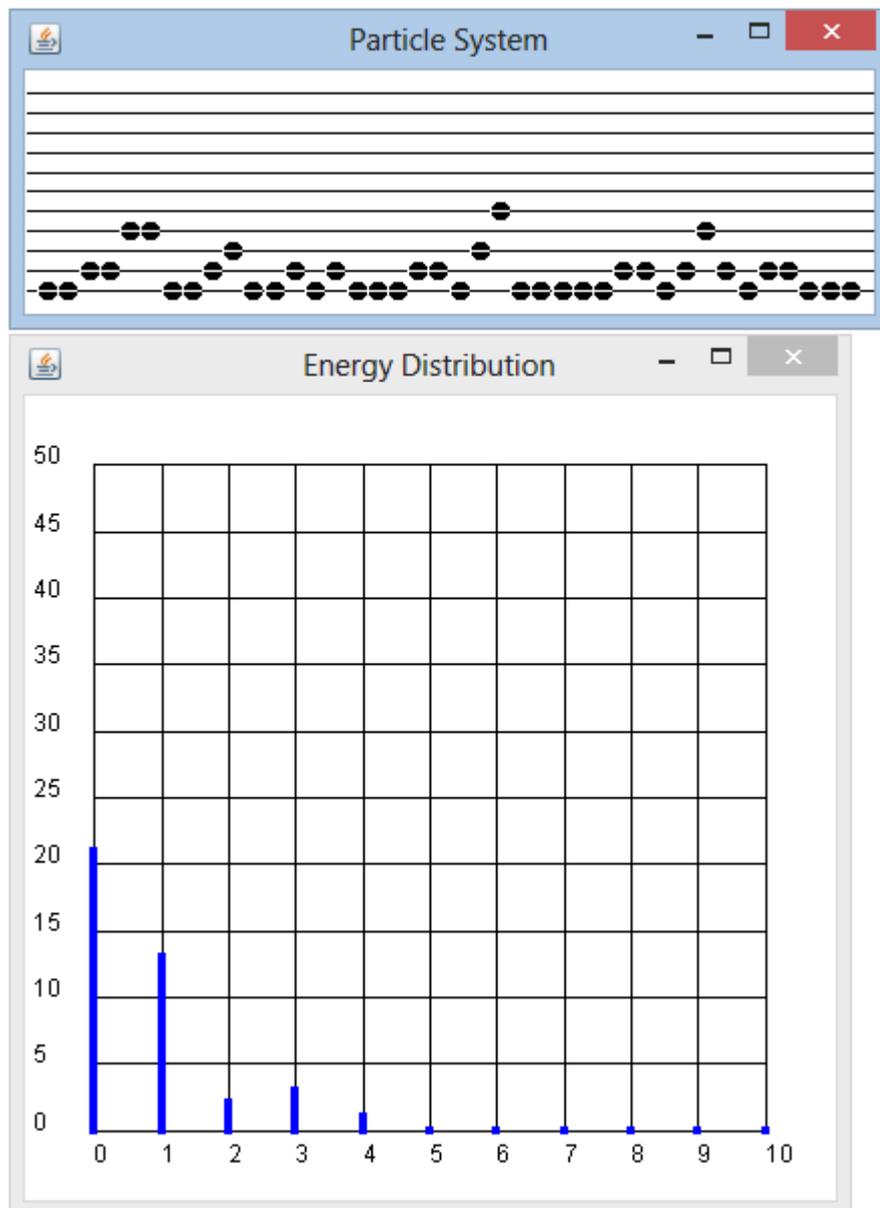
def showDistribution():
    p.clear()
    p.setColor("black")
    p.lineWidth(1)
    drawPanelGrid(p, 0, 10, 0, 50)
    p.setColor("blue")
    p.lineWidth(4)
    for t in range(11):
        p.line(t, 0, t, h[t])
    p.repaint()

nb = 40 # Number of particles
nbStart = 30
eMax = 10 # Energy levels 0..10
d = 10 # Diameter of particles
dt = 0.01 # Simulation period (s)
e = [0] * nb # Energy of each particle
h = [0] * 11 # Energy distribution

initSystem()
initDistribution()

while not isDisposed():
    pair()
    showDistribution()
    delay(10)

```



[Ausführen](#) mit Webstart

2.9 Kurvenanpassung (Fitten, Interpolation)

Die Anpassung wegen Messfehler streuender Messdaten ist eine wichtige Anwendung mathematischer Methoden im Physikunterricht. Man spricht allgemein vom "Fitten" von Messdaten. Da eine Messserie meist in Tabellenform vorliegt, wird das Fitten oft mit Tabellenkalkulationsprogrammen durchgeführt, was in einfachen Fällen auch sinnvoll ist. Immer häufiger werden aber die Messungen direkt in Echtzeit mit dem Computer durchgeführt, gemäss physikalischer Gesetze ausgeglichen und anschliessend weiter verarbeitet (beispielsweise aus Geschwindigkeitsverläufen Ortskurven berechnet). Die Programmierung in Python ist dazu hervorragend geeignet, da mit einigen wenigen Zeilen Code das Problem elegant und professionell gelöst werden kann. Dazu kommen im Physikunterricht Fitting-Funktionen aus einer Software-Bibliothek zum Einsatz, die nur dem Prinzip nach erläutert werden, ohne den meist komplexen Library-Code zu betrachten.

2.9.1 Linearer Fit

Um die Demonstration des Fittens interaktiv zu gestalten, wird die Datenreihe als (x, y)-Paare mit der Maus aufgenommen, indem diese in einem GPanel im Bereich x,y = -10..10 mit gedrückter Taste gezogen wird. Dabei verhindert der Code, dass es zum gleichen x-Wert mehrere y-Werte gibt, indem nur steigende x-Werte zugelassen werden. Für die Datenreihen eignen sich Listen xdata, ydata gleicher Länge mit Float-Werten hervorragend, da man die Anzahl Datenwerte nicht von vorne herein kennt. Die x-Werte sind allerdings nicht äquidistant, da je nach der Geschwindigkeit der Mausbewegung der Drag-Event mehr oder weniger oft ausgelöst wird. Falls nötig, können die Werte mit der Funktion `xdata, ydata = toAequidistant(xrawdata, yrawdata, deltax)` durch lineare Interpolation in äquidistante Daten mit der Schrittweite `deltax` umgewandelt werden. Die Funktion `linfit(xdata, ydata)` liefert `a, b` der (least squares) gefitteten Geraden

$$y(x) = a * x + b$$

```
# FitLin.py

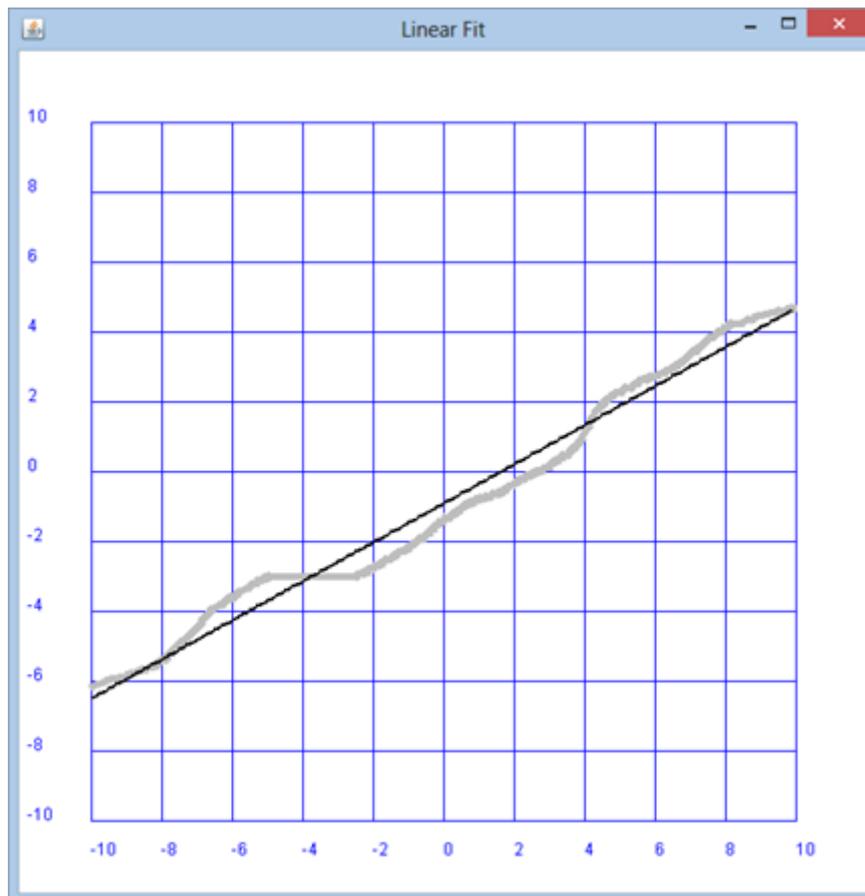
from gpanel import *
from fitter import *

def onMousePressed(x, y):
    global xrawdata, yrawdata
    xrawdata = [x]
    yrawdata = [y]
    lineWidth(5)
    setColor("gray")
    move(x, y)

def onMouseDragged(x, y):
    if x <= xrawdata[-1] + 0.01 or x > 10:
        return
    xrawdata.append(x)
    yrawdata.append(y)
    draw(x, y)

def onMouseReleased(x, y):
    xdata, ydata = toAequidistant(xrawdata, yrawdata, deltax)
    a, b = linfit(xdata, ydata)
    lineWidth(2)
    setColor("black")
    for i in range(len(xdata)):
        if i == 0:
            move(xdata[i], a * xdata[i] + b)
        else:
            draw(xdata[i], a * xdata[i] + b)

deltax = 0.1
makeGPanel(-12, 12, -12, 12, mousePressed = onMousePressed,
           mouseDragged = onMouseDragged, mouseReleased = onMouseReleased)
drawGrid(-10, 10, -10, 10, "blue")
title("Linear Fit")
```



[Ausführen](#) mit Webstart

2.9.2 Polynom-Fit

Das Fitten mit Polynomen mit höheren Graden n

$$y(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

ist zwar naheliegend, führt aber vielfach zu Kurven, die den Schwankungen der Datenwerte zu genau folgen. Weiss man allerdings, dass es sich um eine gleichförmig beschleunigte Bewegung handelt, so ist sie mit $n = 2$ angebracht und liefert mit $a = a_2$ die Beschleunigung. In der Demo wird ein Polynom 5-ten Grades verwendet. Die Funktion `polynomfit(xdata, ydata, order)` nimmt die Datenlisten `xdata` und `ydata` und liefert in `ydata` die ausgeglichenen Werte zu den `xdata`-Werten zurück. Zudem ist der Funktionsrückgabewert eine Liste mit den Polynomkoeffizienten.

```
# FitPolynom.py

from gpanel import *
from fitter import *

def onMousePressed(x, y):
    global xrawdata, yrawdata
    xrawdata = [x]
    yrawdata = [y]
    lineWidth(5)
    setColor("gray")
    move(x, y)

def onMouseDragged(x, y):
    if x <= xrawdata[-1] + 0.01 or x > 10:
        return
```

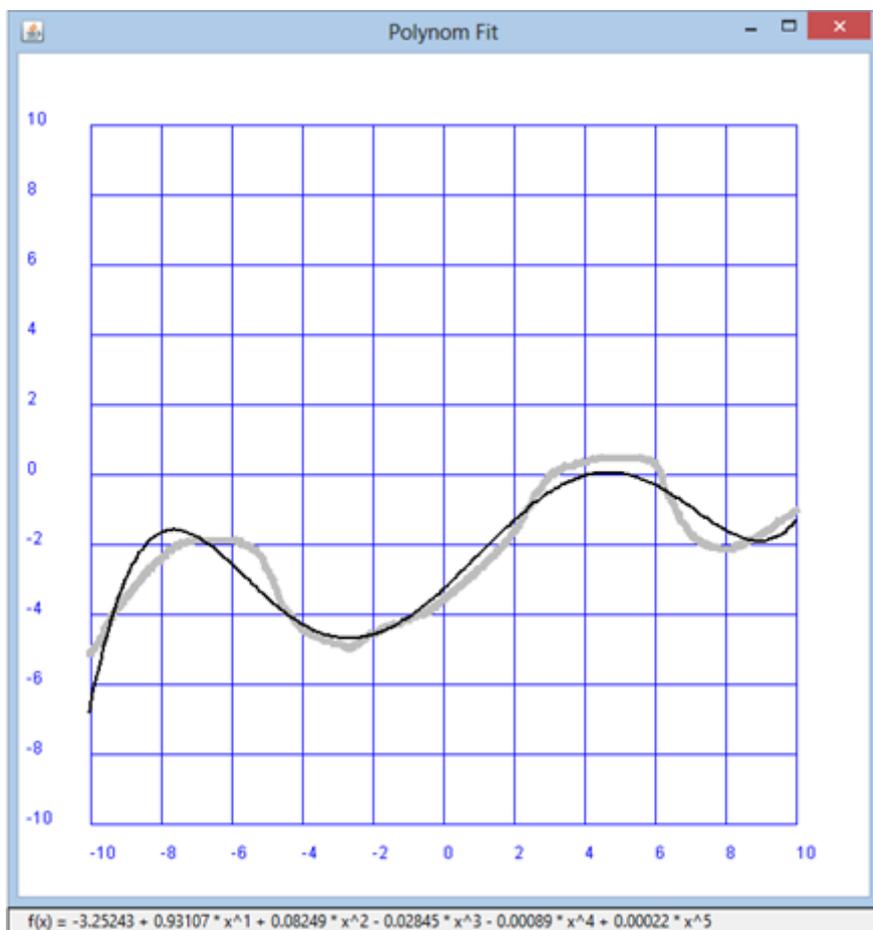
```

xrawdata.append(x)
yrawdata.append(y)
draw(x, y)

def onMouseReleased(x, y):
    xdata, ydata = toAequidistant(xrawdata, yrawdata, deltax)
    coeffs = polynomfit(xdata, ydata, order)
    lineWidth(2)
    setColor("black")
    for i in range(len(xdata)):
        if i == 0:
            move(xdata[i], ydata[i])
        else:
            draw(xdata[i], ydata[i])
    f = "f(x) = " + str(round(coeffs[0], 5))
    for i in range(1, order + 1):
        if coeffs[i] > 0:
            f += " + " + str(round(coeffs[i], 5)) + " * x^" + str(i)
        else:
            f += " - " + str(round(-coeffs[i], 5)) + " * x^" + str(i)
    setStatusText(f)

deltax = 0.1
order = 5
makeGPanel(-12, 12, -12, 12, mousePressed = onMousePressed,
            mouseDragged = onMouseDragged, mouseReleased = onMouseReleased)
drawGrid(-10, 10, -10, 10, "blue")
title("Polynom Fit")
addStatusBar(20)

```



[Ausführen](#) mit Webstart

Instruktiv ist die Darstellung des angepassten Polynoms durch Eingabe von einzelnen Datenpunkten mit einem Mausklick. Für ein Polynom n-ten Grades sind mindestens $n + 1$ Datenpunkte nötig, bevor mit einem Doppelklick das Polynom gezeichnet wird.

```
# FitPolynom1.py

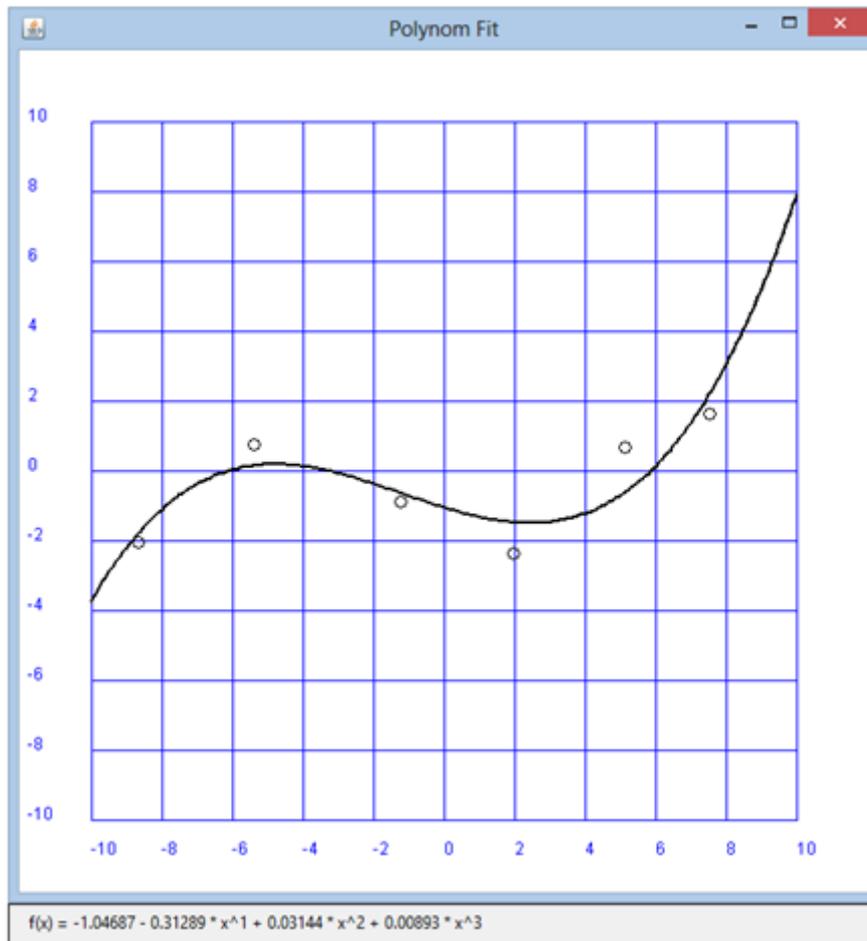
from gpanel import *
from fitter import *

def func(x, coeffs):
    y = 0
    for i in range(len(coeffs)):
        y += coeffs[i] * x**i
    return y

def onMouseSingleClicked(x, y):
    xdata.append(x)
    ydata.append(y)
    move(x, y)
    circle(0.15)

def onMouseDoubleClicked(x, y):
    global xdata, ydata
    if len(xdata) < 4:
        setStatusText("Not enough data points (needed: 4).")
        return
    coeffs = polynomfit(xdata, ydata, order)
    lineWidth(2)
    setColor("black")
    x = -10
    dx = 0.1
    while x <= 10:
        if x == -10:
            move(x, func(x, coeffs))
        else:
            draw(x, func(x, coeffs))
        x += dx
    f = "f(x) = " + str(round(coeffs[0], 5))
    for i in range(1, order + 1):
        if coeffs[i] > 0:
            f += " + " + str(round(coeffs[i], 5)) + " * x^" + str(i)
        else:
            f += " - " + str(round(-coeffs[i], 5)) + " * x^" + str(i)
    setStatusText(f)
    xdata = []
    ydata = []

order = 3
xdata = []
ydata = []
makeGPanel(-12, 12, -12, 12, mouseSingleClicked = onMouseSingleClicked,
            mouseDoubleClicked = onMouseDoubleClicked)
drawGrid(-10, 10, -10, 10, "blue")
title("Polynom Fit")
addStatusBar(30)
setStatusText("Click to create a data point. " +
              "Double click to fit polynom of order 3")
```



[Ausführen](#) mit Webstart

2.9.3 Spline-Fit

Splines sind eigentlich dünne, biegbare Latten. Anschaulich gesprochen wählt man einige Stützstellen des Datensatzes (Knotenpunkte genannt) und schlägt dort zwei engliegende Nägel ein. Dann verbindet man diese Punkte mit der Latte. Es ergeben sich harmonische Kurven ohne Ecken. Mathematisch handelt es sich um eine stückweise polynomiale Kurve (gewöhnlich dritten Grades) mit übereinstimmenden Ableitungen an den Knotenpunkten. `splinefit(xdata, ydata)` erhält die Knotenpunkte und liefert eine Funktion `spline(x)` zurück. Es sind mindestens 3 Knotenpunkte nötig.

```
# FitSpline.py

from gpanel import *
from fitter import *

def onMousePressed(x, y):
    global xrawdata, yrawdata
    xrawdata = [x]
    yrawdata = [y]
    lineWidth(5)
    setColor("gray")
    move(x, y)
```

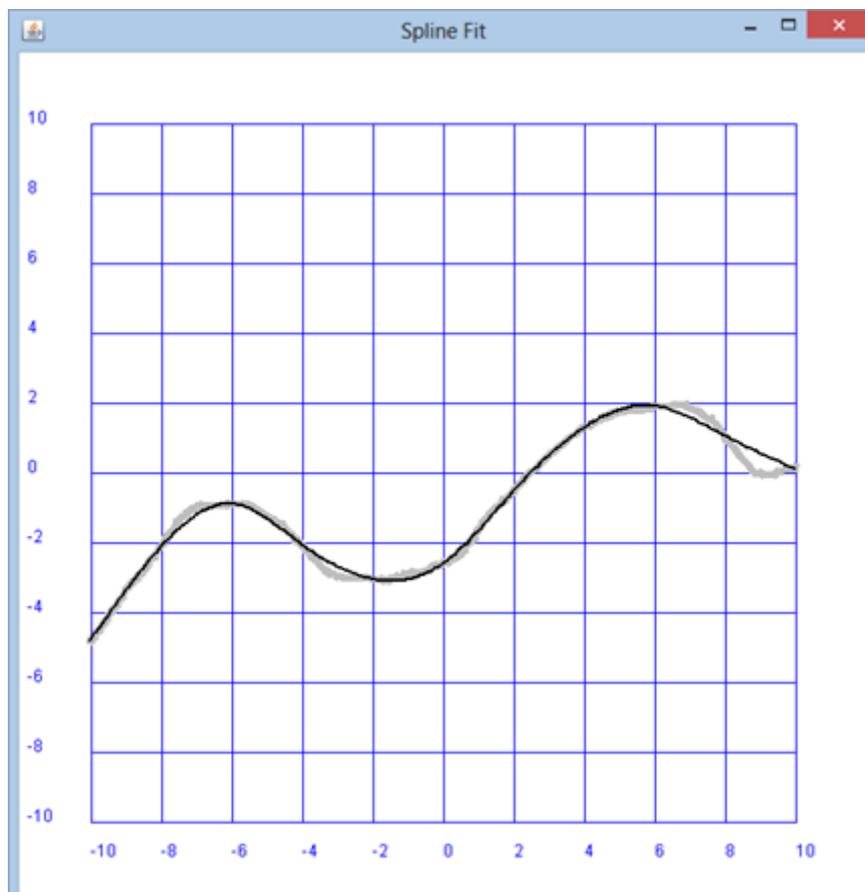
```

def onMouseDragged(x, y):
    if x <= xrawdata[-1] + 0.01 or x > 10:
        return
    xrawdata.append(x)
    yrawdata.append(y)
    draw(x, y)

def onMouseReleased(x, y):
    xdata, ydata = toAequidistant(xrawdata, yrawdata, deltax)
    xdata.append(x)
    ydata.append(y)
    spline = splinefit(xdata, ydata)
    lineWidth(2)
    setColor("black")
    start = min(xdata)
    end = max(xdata)
    x = start
    dx = 0.1
    while x <= end:
        if x == start:
            move(x, spline(x))
        else:
            draw(x, spline(x))
        x += dx

deltax = 2
makeGPanel(-12, 12, -12, 12, mousePressed = onMousePressed,
            mouseDragged = onMouseDragged, mouseReleased = onMouseReleased)
drawGrid(-10, 10, -10, 10, "blue")
title("Spline Fit")

```



Ausführen mit Webstart

Auch hier ist es instruktiv, wenn die Knotenpunkt interaktiv mit Mausklicks erzeugt werden können. Das Vorgehen ist dasselbe wie beim Polynomfit.

```
# FitSpline.py

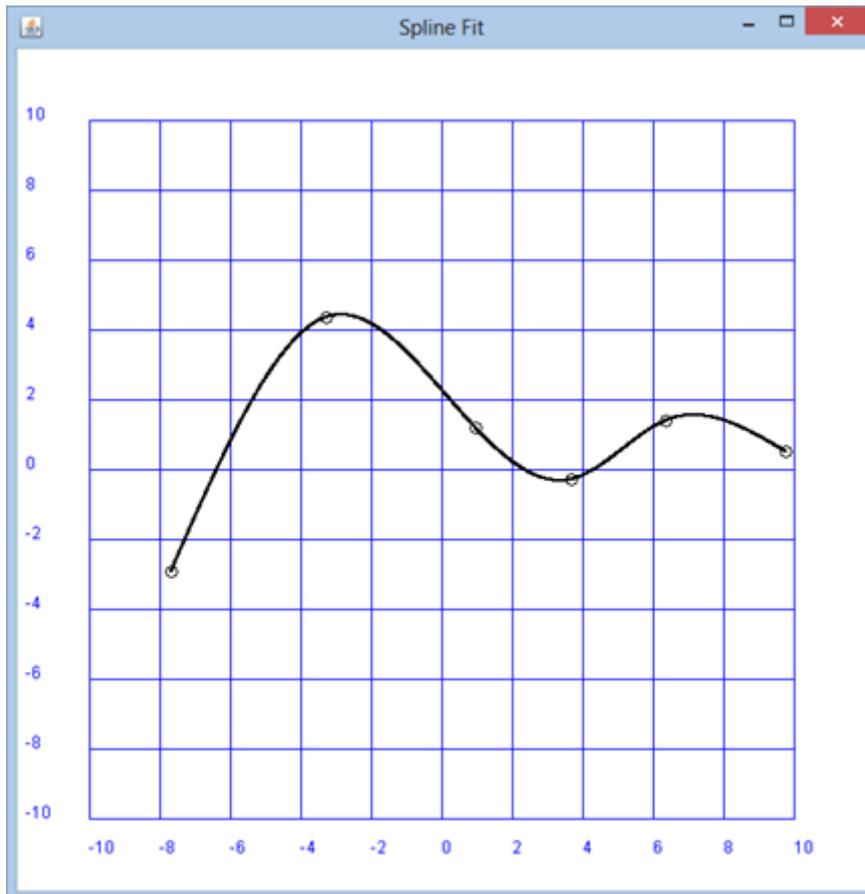
from gpanel import *
from fitter import *

def onMousePressed(x, y):
    global xrawdata, yrawdata
    xrawdata = [x]
    yrawdata = [y]
    lineWidth(5)
    setColor("gray")
    move(x, y)

def onMouseDragged(x, y):
    if x <= xrawdata[-1] + 0.01 or x > 10:
        return
    xrawdata.append(x)
    yrawdata.append(y)
    draw(x, y)

def onMouseReleased(x, y):
    xdata, ydata = toAequidistant(xrawdata, yrawdata, deltax)
    xdata.append(x)
    ydata.append(y)
    spline = splinefit(xdata, ydata)
    lineWidth(2)
    setColor("black")
    start = min(xdata)
    end = max(xdata)
    x = start
    dx = 0.1
    while x <= end:
        if x == start:
            move(x, spline(x))
        else:
            draw(x, spline(x))
        x += dx

deltax = 2
makeGPanel(-12, 12, -12, 12, mousePressed = onMousePressed,
           mouseDragged = onMouseDragged, mouseReleased = onMouseReleased)
drawGrid(-10, 10, -10, 10, "blue")
title("Spline Fit")
```



[Ausführen](#) mit Webstart

2.9.4 FFTFilterFit

Fasst man die Datenreihe als ein Signal in Funktion der Zeit auf, so kann man dieses fouriertransformieren. Die schnellen zeitlichen Schwankungen führen zu höherfrequenten Spektralanteilen. Schneidet man diese im Spektrum weg und macht eine Rücktransformation in den Zeitbereich, so erhält man eine geglättete Kurve. Diese Verfahren eignet sich vor allem bei quasi-periodischen Signalen. Sie entspricht einer Datenfilterung durch ein Tiefpass-Filter. Wie man sieht, ist der Code genauso einfach wie vorher, obschon der dahinter stehende Algorithmus, der auf der Fast-Fourier-Transformation basiert, wesentlich komplexer ist.

Die FFT-Routinen befinden sich im Modul `soundsystem`, das importiert werden muss. Die Funktion `fft_filter(ydata, cutoff, isLowpass)` führt mit `isLowpass = True` eine Tiefpassfilterung und mit `isLowPass = False` eine Hochpassfilterung auf dem Datensatz `ydata` durch und liefert die gefilterten Daten in der gleichen Liste wieder zurück.

`cutoff` muss im Bereich `0..n` liegen, wo `n` die Länge des Datensatzes `n = len(ydata)` ist. Ist die Sampling-Frequenz `fs` (hier `1 / deltax = 10 Hz`), so sind verläuft das Spektrum bis 5 Hz, was `cutoff = n` entspricht.

```
# FitFFT.py
```

```

from gpanel import *
from soundsystem import *

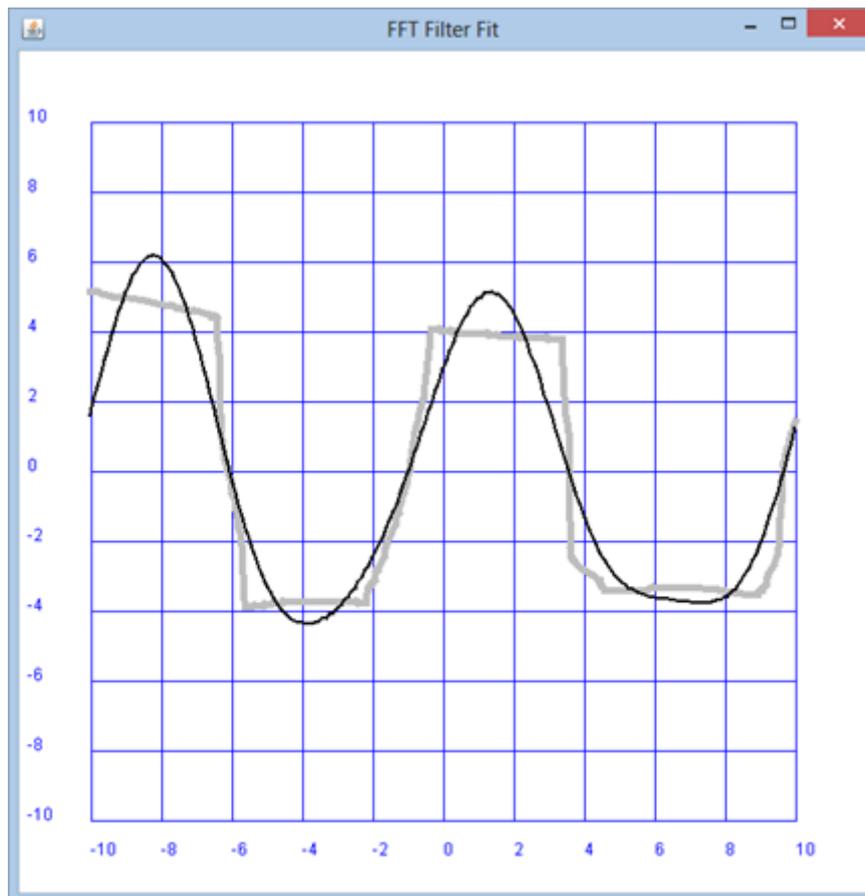
def onMousePressed(x, y):
    global xrawdata, yrawdata
    xrawdata = [x]
    yrawdata = [y]
    lineWidth(5)
    setColor("gray")
    move(x, y)

def onMouseDragged(x, y):
    if x <= xrawdata[-1] + 0.01 or x > 10:
        return
    xrawdata.append(x)
    yrawdata.append(y)
    draw(x, y)

def onMouseReleased(x, y):
    xdata, ydata = toAequidistant(xrawdata, yrawdata, deltax)
    fft_filter(ydata, cutoff, True)
    lineWidth(2)
    setColor("black")
    for i in range(len(xdata)):
        if i == 0:
            move(xdata[i], ydata[i])
        else:
            draw(xdata[i], ydata[i])

cutoff = 5
deltax = 0.1
makeGPanel(-12, 12, -12, 12, mousePressed = onMousePressed,
           mouseDragged = onMouseDragged, mouseReleased = onMouseReleased)
drawGrid(-10, 10, -10, 10, "blue")
title("FFT Filter Fit")

```



[Ausführen](#) mit Webstart

2.9.5 Funktionsfit

Der allgemeine Fall lässt sich wie folgt beschreiben: Es wird angenommen, dass die Listen xdata, ydata Messdaten einer physikalischen Gesetzmässigkeit

$$y = f_{a,b,c,\dots}(x)$$

mit den noch unbestimmten (freien) Parametern a, b, c, ... sind. Wie immer, weisen diese (eventuell unterschiedlich zu gewichtende) Messfehler auf. Man sucht die Parameter a,b,c,... so zu bestimmen, dass die Datenreihe das Gesetz möglichst gut wiedergibt.

Der Algorithmus bestimmt die Parameter durch Optimierung mit einem aufwendigen Last-Squares Verfahren. Dazu müssen die (partiellen) Ableitungen der Funktion f nach den Parametern angegeben werden. Wir zeigen dies an besten an einem Beispiel:

$$y = f(x) = a * x^b * e^{-cx}$$

Wie man von Hand oder mit einem symbolischen Algebraprogramm nachprüfen kann, lauten die Ableitungen:

$$\frac{\partial f}{\partial a} = x^b * e^{-cx} \quad \frac{\partial f}{\partial b} = a * x^b * e^{-cx} * \ln(x) \quad \frac{\partial f}{\partial c} = -a * x^{b+1} * e^{-cx}$$

Man definiert die Gesetzmässigkeit in der Funktion `func(x, params)`, wobei `params` eine Liste mit allen Parametern ist. Die partiellen Ableitungen werden in der Funktion `derivatives(x, params)` angegeben, die eine Liste mit den aus `x` und `params` berechneten Ableitungswerten zurückgibt. Die Datenreihe `xdata`, `ydata` ist hier willkürlich gewählt. In `weight` lassen sich auch noch relative Gewichte der Datenpunkte angeben. Das Fitten erfolgt mit `functionfit(func, derivatives, initialGuess, xdata, ydata)` mit einem optionalen letzten Parameter `weight`. `initialGuess` ist eine Liste mit vernünftig gewählten Vorschlägen für die Parameterwerte, damit das Fitten möglichst rasch erfolgt. Abgesehen von den partiellen Ableitungen ist der Code nicht wesentlich komplizierter als bei den anderen Fitting-Verfahren. Es ist leicht, eine interaktive Version mit der Erzeugung der Datenpunkte durch Mausklicks zu erstellen.

```
# FitFunc.py

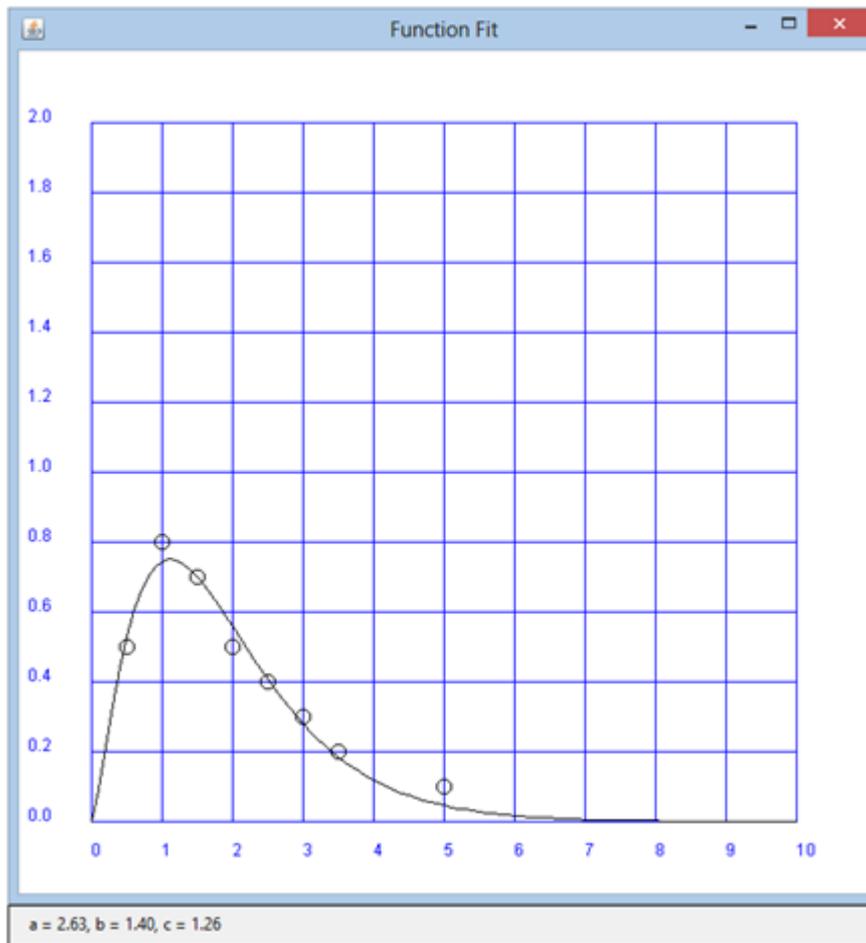
from gpanel import *
from fitter import functionfit

def func(x, params):
    # y = a * x**b * e**(-c*x)
    a = params[0]
    b = params[1]
    c = params[2]
    return a * math.pow(x, b) * math.exp(-c * x)

def derivatives(x, params):
    a = params[0]
    b = params[1]
    c = params[2]
    return [math.pow(x, b) * math.exp(-c * x),
            a * math.pow(x, b) * math.exp(-c * x) * math.log(x),
            -a * math.pow(x, b + 1) * math.exp(-c * x)]

makeGPanel(-1, 11, -0.2, 2.2)
drawGrid(0, 10, 0.0, 2, "blue")
title("Function Fit")
addStatusBar(30)
xdata = [0.5, 1, 1.5, 2, 2.5, 3, 3.5, 5]
ydata = [0.5, 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1]
weights = [1, 0.1, 1, 1, 1, 1, 1, 1]
for i in range(8):
    move(xdata[i], ydata[i])
    circle(0.1)
initialGuess = [1, 1, 1]
coeffs = functionfit(func, derivatives, initialGuess, xdata, ydata)
setStatusText("a = %2.2f, b = %2.2f, c = %2.2f"
              % (coeffs[0], coeffs[1], coeffs[2]))

dt = 0.1
t = 0
while t <= 10:
    if t == 0:
        move(t, func(t, coeffs))
    else:
        draw(t, func(t, coeffs))
    t += dt
```



[Ausführen](#) mit Webstart

2.10 Messgerätesteuerung, Sensorik und Robotik

Für viele moderne Messgeräte gibt es die Möglichkeit, die erfassten Daten über die USB-Schnittstelle mit einem spezifischen Anwenderprogramm auszulesen und Messgeräteeinstellungen vorzunehmen. In der Schule werden auch oft komplette Messerfassungssysteme eingesetzt, bei denen ein microprozessorbasiertes Messinterface mit dem Computer verbunden wird und die Daten mit einem proprietären Protokoll ausgetauscht werden. In manchen Fällen ist die mitgelieferte Software aber zu wenig flexibel, um die Daten direkt weiter zu verarbeiten, beispielsweise um sie zu filtern oder nichtlinear zu fitten, oder in einer für das Experiment angepassten Form grafisch darzustellen. Darum werden die Daten oft in Rohform in Dateien gespeichert und nachträglich weiterverarbeitet. Auch bei dieser Weiterverarbeitung kann natürlich eine höhere Programmiersprache wie Python gegenüber Anwenderprogrammen wie Excel grosse Vorteile bringen, insbesondere weil das Scripting bei Anwenderpaketen mühsam ist und wenig Allgemeingültigkeit hat, so dass sich der Einarbeitungsaufwand kaum lohnt.

Besonders interessant und lehrreich ist es, abgesetzte Messgeräte einzusetzen, die nicht mit einer Kabelverbindung mit dem Steuerungs- und Auswertesystem verbunden sind. Sie können auch als Messroboter bezeichnet werden und daher der Physik, Elektronik oder Robotik zugeordnet sein. Für die Schule ergibt sich beim

Einsatz der Messroboter ein interessantes fächerübergreifendes Thema. Als Roboterplattform eignet sich besonders gut der Raspberry Pi, da er über vielseitige Schnittstellen verfügt, aber auch mit einer modernen höheren Programmiersprache (v.a. Python) unter einem modernen Betriebssystem (Linux) programmiert wird.

Der bei Jugendlichen weit verbreitete Arduino ist für Bastelarbeiten gut geeignet, vermittelt aber durch die Verwendung eines stark abgemagerten C++ wenig Informatikgrundwissen und verleitet zu einer schludrigen Programmieretechnik.

Zur Messung physikalischer Grössen werden verschiedenartige Sensoren eingesetzt, die alle auf interessanten physikalischen Gesetzmässigkeiten aufbauen. Die Sensorik ist darum zu einem wichtigen Teilgebiete der Physik geworden und kann auch im Physikunterricht gewinnbringend eingesetzt werden, da es sich bei Sensoren um eine praxisnahe und nützliche Anwendung physikalischer Grundlagen handelt.

Insbesondere in der Robotik spielen neben Sensoren auch Aktoren eine wichtige Rolle. Dabei kann es sich um einfache Elektromotoren zum Antrieb von Rädern oder um LEDs in verschiedenen Anzeigeelementen handeln. Steuert man Aktoren mit einem Raspberry Pi an, so motiviert dies auch, ihre Funktionsweise zu verstehen wodurch sich weiterer Praxisbezug zum Physikunterricht ergibt.

Über die Sensorik und der Verwendung von Aktoren im Zusammenhang mit einem Mikrocontroller, z.B. dem Raspberry Pi, tauchen ganz natürlich viele fächerübergreifende Themen aus den Bereichen Physik, Informatik, Umwelt und Technik auf. Wie die Erfahrung der letzten Jahre zeigt, wählen viele Schülerinnen und Schüler in der Maturaarbeit tatsächlich Problemstellungen, bei denen ein Microcontroller für die Messung, Steuerung und Datenübertragung verwendet wird.

Wegen der besonderen Wichtigkeit der Sensorik und der Verwendung von Sensoren und Aktoren im Zusammenhang mit selbsterstellten Mess- und Steuerungsprogrammen ist dem interessanten Thema ein eigenes Online-Tutorial gewidmet (www.python-exemplarisch.ch/rpi).

3. Downloads und Literatur

- Download aller Programme von <http://www.tigerjython.ch/download/apluphy.zip>
- Bitte neuste Version von TigerJython verwenden. Download von <http://jython.tobiaskohn.ch/download2-de.html>
- Landau, Páez, Bordeianu, Computational Physics, Problem solving with Python, 3rd edition, Wiley-VCH
- Programmierkonzepte mit Python und der Lernumgebung TigerJython, <http://www.tigerjython.ch>
- Mikroprozessor-Programmierung und Sensorik mit dem Raspberry Pi <http://www.python-exemplarisch.ch/rpi>
- Downey, Programmieren lernen mit Python, O'Reilly