

Zusatzmaterial zu dem Buch: Mechanik verstehen mit Python

Das Zusatzmaterial enthält Matplotlib- und VPython-Programme, die einzelne Themenbereiche ergänzen und vertiefen sollen. Die Quelltexte finden Sie im Downloadbereich: <https://drsteinkamp.de>

1 Einführung

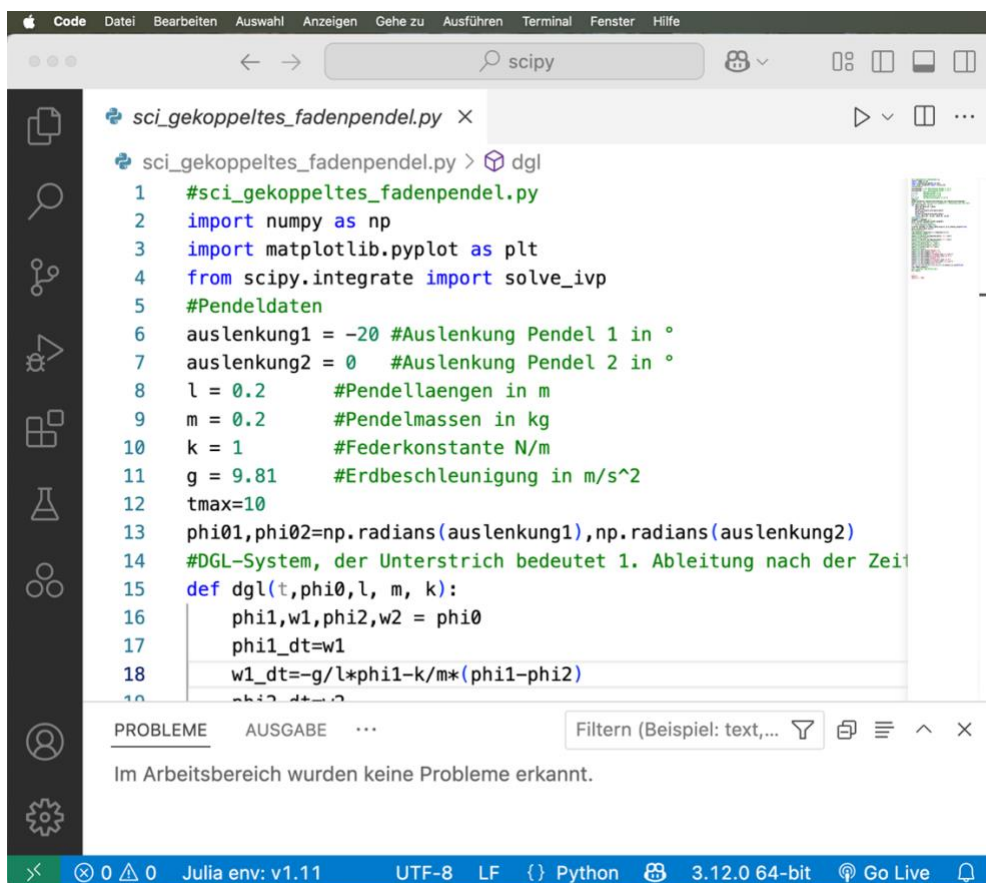
Der Visual Studio Editor von Microsoft ist eine Alternative zu den im Buch vorgestellten Entwicklungsumgebungen: <https://code.visualstudio.com>.

Über

Code → **Einstellungen** → **Erweiterungen** (oder shift + command + X) gelangen Sie zu dem Menü **Erweiterungen**. Oder Sie klicken auf das Icon



In dem Textsuchfeld für **Erweiterungen** müssen Sie "Python" eingeben. Die verfügbaren Optionen werden angezeigt und Sie können die **Python extension for Visual Studio Code** installieren.



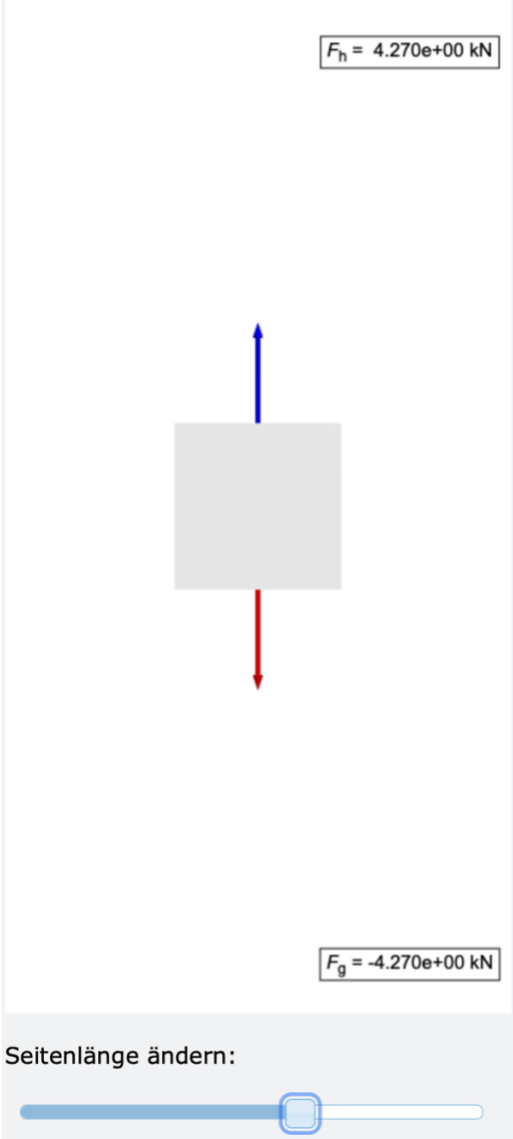
Anschließend müssen Sie eventuell unten rechts die gewünschte Python-Version (hier: 3.12.0 64-bit) auswählen. Durch einen Mausklick auf das Dreieck oben rechts können Sie ihr Python-Programm ausführen.

2 Statik

Kräftegleichgewicht

Ein statisches System ist im Gleichgewicht, wenn sich die Summe aller Kräfte und die Summe aller Drehmomente aufheben. Man denke an ein Haus: Damit die Wände des Hauses keine Risse bilden, muss das Fundament der Gewichtskraft des Hauses mit einer betragsmäßig gleichen Gegenkraft entgegenwirken.

Das VPython-Programm `vp_kraeftegleichgewicht.py` veranschaulicht das Kräftegleichgewicht.

Kräftegleichgewicht: Kraft = Gegenkraft 	Der Screenshot zeigt die Benutzeroberfläche des VPython-Programms. In der Mitte befindet sich ein grauer Würfel aus Beton. Der rote Pfeil, der in Richtung der negativen y-Achse zeigt, soll die Gewichtskraft repräsentieren. Um das Gleichgewicht zu halten, muss eine Hubkraft (blauer Pfeil) der Gewichtskraft entgegenwirken. Sie muss betragsmäßig gleich groß sein. Wenn Sie die Seitenlänge des Würfels ändern, ändert sich das Volumen des Würfels und damit auch die Gewichtskraft und die Hubkraft. Damit das statische System (grauer Würfel) im Gleichgewicht bleibt, müssen beide Kräfte betragsmäßig gleich groß sein. Für das Volumen des Würfels gilt: $V = a^3$ Mithilfe des Volumens und der Materialdichte kann die Masse berechnet werden: $m = \rho \cdot V$ Wenn die Masse bekannt ist, kann die Gewichtskraft berechnet werden: $F_g = m \cdot g$ Die Gewichtskraft wird unten rechts und die Hubkraft oben rechts angezeigt.
--	--

In Zeile 3 können Sie die Dichte des Materials ändern.

```
#vp_kraeftegleichgewicht.py
from vpython import *
rho=2000    #Dichte von Beton in kg/m^3
g=9.81      #m/s^2
#Dehnung ändern
def seite(s):
```

```

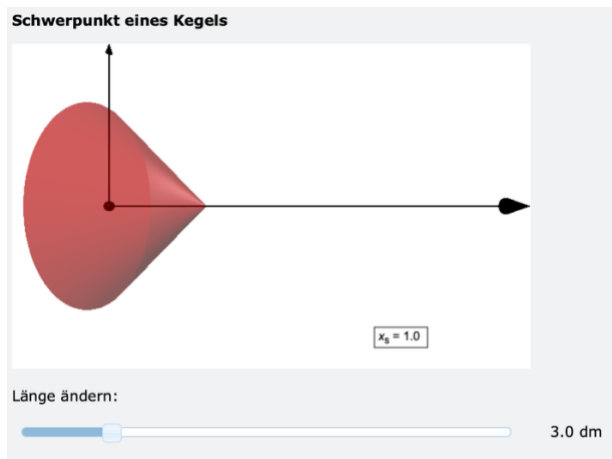
a=s.value
V=a**3          #Volumen
m=rho*V         #Masse
Fg=m*g*1e-6    #Umwandlung in kN
wuerfel.size=vec(a,a,a)
pfeilFg.pos=vec(0,-a/2,0)
pfeilFh.pos=vec(0, a/2,0)
pfeilFg.axis.y = -Fg
pfeilFh.axis.y =  Fg
lblFh.text = '<i>F</i><sub>h</sub> = %2.3e kN' %Fg #actio
lblFg.text = '<i>F</i><sub>g</sub> = -%2.3e kN' %Fg #reactio
txtSeite.text = '\t %2.1f' %s.value
#Grafikbereich
scene.title="<h3>Kräftegleichgewicht: Kraft = Gegenkraft</h3>"
scene.width=400
scene.height=800
scene.center=vector(0,0,0)
scene.range=10
scene.background = color.white
scene.userzoom = False    #kein Zoom
scene.userspin = False
#System im Gleichewicht
wuerfel=box(length=3, height=3,width=3)
wuerfel.pos=vec(0,0,0)
#Pfeil Hubkraft
pfeilFh=arrow(axis=vec(0, 5, 0),color=color.blue,round=True)
pfeilFh.pos=vec(0,0,0)
pfeilFh.shaftwidth=0.2
#Pfeil Gewichtskraft
pfeilFg=arrow(axis=vec(0, -5, 0),color=color.red,round=True)
pfeilFg.pos=vec(0,0,0)
pfeilFg.shaftwidth=0.2
#Label für Ausgabe der Federkräfte
lblFh=label(pos=vec(6, 18,0),text='Fh  = %2.1f kN' %4.3)
lblFg=label(pos=vec(6,-18,0),text='Fg  = %2.1f kN' %-4.3)
#****Slider für Kraftänderung****
scene.caption = "\nSeitenlänge ändern: \n\n" #Beschriftung des Sliders
#Sliderobjekt für Kraftänderung erzeugen
sldSeite = slider(bind=seite,min=3, max=8,value=6,length=400, right=15) #2
#Textfeld für Ausgabe der Seitenlänge
txtSeite = wtext(text='\t %2.1f' %sldSeite.value)
#Einheit der Seitenlänge
scene.append_to_caption(" m\n")

```

In #1 wird der Wert der Slider-Einstellung des Sliders `sldSeite` ausgelesen und der Variablen `a` zugewiesen. In #2 wird die selbstdefinierte Python-Funktion `seite` aufgerufen (ohne Parameter `s`).

Schwerpunkt eines Kegels

Mit dem VPython-Programm `vp_slider_zyylinder.py` können Sie den Schwerpunkt eines Kegels veranschaulichen.



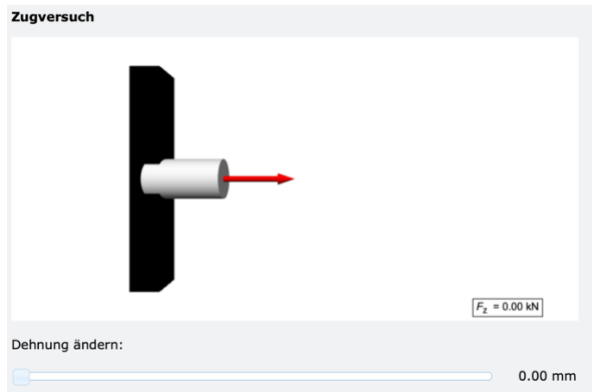
Wenn Sie den Slider-Knopf verschieben, verändern sich die Länge des Kegels. Der Schwerpunkt x_s wird aktualisiert.

```
#vp_slider_zylinder.py
from vpython import *
#Daten
l0=3
r0=3
x0 = -5
#Berechnungen durchführen
def schwerpunkt():
    l=sldLaenge.value
    xs=l/3
    y_achse.pos=vec(xs+x0,0,0)
    SP.pos=vec(xs+x0,0,0)
    kegel.length = l
    txtLaenge.text=text='\t %2.1f dm' %l
    lblXs.text = '<i>x</i><sub>s</sub> = %2.2f' %xs
#Eigenschaften der Zeichenfläche
scene.title="<h3>Schwerpunkt eines Kegels</h3>"
scene.background = color.white
scene.range=5
scene.userzoom = False      #kein Zoom
scene.userspin = False
#Zylinderobjekt erzeugen
kegel = cone(axis=vec(10,0,0),radius=r0,color=color.red,opacity=0.4)
kegel.pos=vec(x0,0,0)
y_achse=arrow(axis=vec(0,5,0),round=True,color=color.black)
y_achse.pos=vec(x0,0,0)
x_achse=arrow(axis=vec(13,0,0),round=True,color=color.black)
x_achse.pos=vec(x0,0,0)
x_achse.shaftwidth=0.05
y_achse.shaftwidth=0.05
SP=sphere(pos=vec(x0,0,0),radius=0.15,color=color.black)
#Slider: Länge ändern
scene.caption = "\nLänge ändern: \n\n" #Beschriftung des Sliders
sldLaenge =
slider(bind=schwerpunkt,min=1,max=12,step=0.1,value=l0,length=650)    #2
txtLaenge = wtext(text='\t %2.1f dm' %sldLaenge.value)
#Ergebnisse ausgeben
lblXs = label(pos=vec(4,-4,0),text='<i>x</i><sub>s</sub> = %2.1f ' %l)
```

In #1 wird der Wert des Sliders sldLaenge ausgelesen und in die Variable l gespeichert. in #2 wird die selbstdefinierte Python-Funktion schwerpunkt aufgerufen.

3 Mechanik deformierbarer Körper

Mit dem VPython-Programm *vp_zugversuch.py* können Sie einen Zugversuch animieren.



Die Dehnung des Stabes wird, damit sie überhaupt wahrgenommen werden kann, übertrieben dargestellt. Die Zugkraft steigt proportional zur Dehnung.

```
#vp_zugversuch.py
from vpython import *
#Daten
E=2.1e5      #Elastizitätsmodul in N/mm^2
A=10         #Querschnittsfläche der Probe in mm^2
l0 = 1000    #Stablänge in mm
#Dehnung ändern
def dehnung(s):
    dl=s.value #aktuelle Stablänge
    probe.axis.x = dl + 5
    pfeilFz.pos=probe.pos + vec(dl + 5,0,0)
    pfeilFz.axis.x = dl + 5
    epsilon=dl/l0      #Dehnung
    F=E*A*epsilon*1e-3  #Federkraft in kN
    lblFz.text = '<i>F</i><sub>z</sub> = %2.2f kN' %F #Zugkraft
    txtWeg.text = '\t %2.2f' %s.value
#Grafikbereich
scene.title="<h3>Zugversuch</h3>"
scene.width=800
scene.height=400
scene.center=vector(0,0,0)
scene.range=10
scene.background = color.white
scene.userzoom = False      #kein Zoom
scene.userspin = False
#Wand für Befestigung der Probe
wand=box(pos=vec(-10,0,0),length=2,height=15,width=2,color=color.black)
#Stab als Probe
probe=cylinder(axis=vector(5,0,0),radius=1.4)
probe.pos=wand.pos
#Pfeil für Zugkraft
pfeilFz=arrow(axis=vec(10, 0, 0),color=color.red,round=True)
pfeilFz.shaftwidth=0.4
pfeilFz.pos=probe.pos
#Label für Ausgabe der Federkräfte
lblFz=label(pos=vec(+15,-9,0),text='<i>F</i><sub>z</sub> = %2.2f kN' %0)
#****Slider für Wegänderung****
scene.caption = "\nDehnung ändern: \n\n" #Beschriftung des Sliders
#Sliderobjekt für Dehnung erzeugen
sldWeg = slider(bind=dehnung,min=0,max=10,value=0,length=700,right=15)
#Textfeld für Ausgabe der Dehnung
```

```
txtWeg = wtext(text='\t %2.2f' %sldWeg.value)
#Einheit des Wegs
scene.append_to_caption(" mm\n")
```

Das Programm können Sie im Unterricht einsetzen: Die Benutzeroberfläche des Programms wird mit dem Beamer für alle Schüler gut sichtbar auf ein Whiteboard projiziert. Sie starten das Programm und vergrößern schrittweise die Zugkraft. Die Schüler tragen die Werte für Dehnung und Zugkraft in eine Tabelle ein und zeichnen anschließend den Funktionsgraphen. Wenn Sie den Funktionsgraphen gezeichnet haben, können Sie die Steigung des Funktionsgraphen ermitteln. Sie erkennen, dass es bei der Steigung um das Elastizitätsmodul E handelt.

4 Kinematik

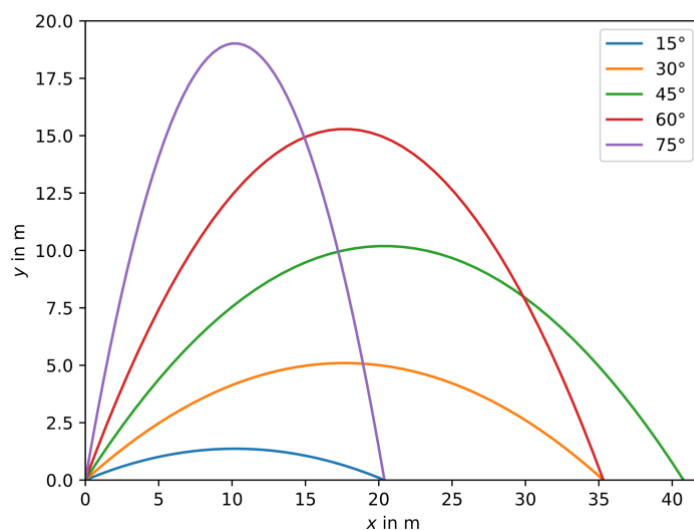
Parameterdarstellung des schiefen Wurfs

Für die x- und y-Komponenten des schiefen Wurfs gilt:

$$x = v_0 \cos \alpha \cdot t$$

$$y = v_0 \sin \alpha \cdot t - \frac{1}{2} g \cdot t^2$$

Das Bild zeigt Wurfparabeln für verschiedenen Abwurfwinkel (Parameterdarstellung).



Es wurde mit dem Matplotlib-Programm *plot_parabelboegen.py* erzeugt.

```
#plot_parabelboegen.py
#Parameterdarstellung für verschiedene Abwurfwinkel
import numpy as np
import matplotlib.pyplot as plt
#Daten
v0=20
g=9.81
#Funktionsdefinition
def f(t,alpha):
    x=v0*np.cos(alpha)*t
    y=v0*np.sin(alpha)*t - g*t**2/2
    return x,y
#Grafikbereich
```

```

fig,ax=plt.subplots()
ax.axis([0,42,0,20])
for winkel in [15,30,45,60,75]:
    alpha=np.radians(winkel)
    tw=2*v0*np.sin(alpha)/g
    t=np.linspace(0,tw)
    x,y=f(t,alpha)
    ax.plot(x,y,label=str(winkel)+'°')
ax.legend()
ax.set(xlabel='$x$ in m',ylabel='$y$ in m')
plt.show()

```

Der Funktionsplot der Wurfparabeln wird innerhalb einer `for`-Schleife ausgeführt. Die Abwurfwinkel sind in einer Liste gespeichert. Die Laufvariable `winkel` iteriert über eine Liste mit den verschiedenen Abwurfwinkeln (#1). In #2 wird die selbstdefinierte Python-Funktion `f(t, alpha)` aufgerufen.

Ableitungen der Wurfparabel

Für die y-Komponente gilt:

$$y = v_0 \sin \alpha \cdot t - \frac{1}{2} g \cdot t^2$$

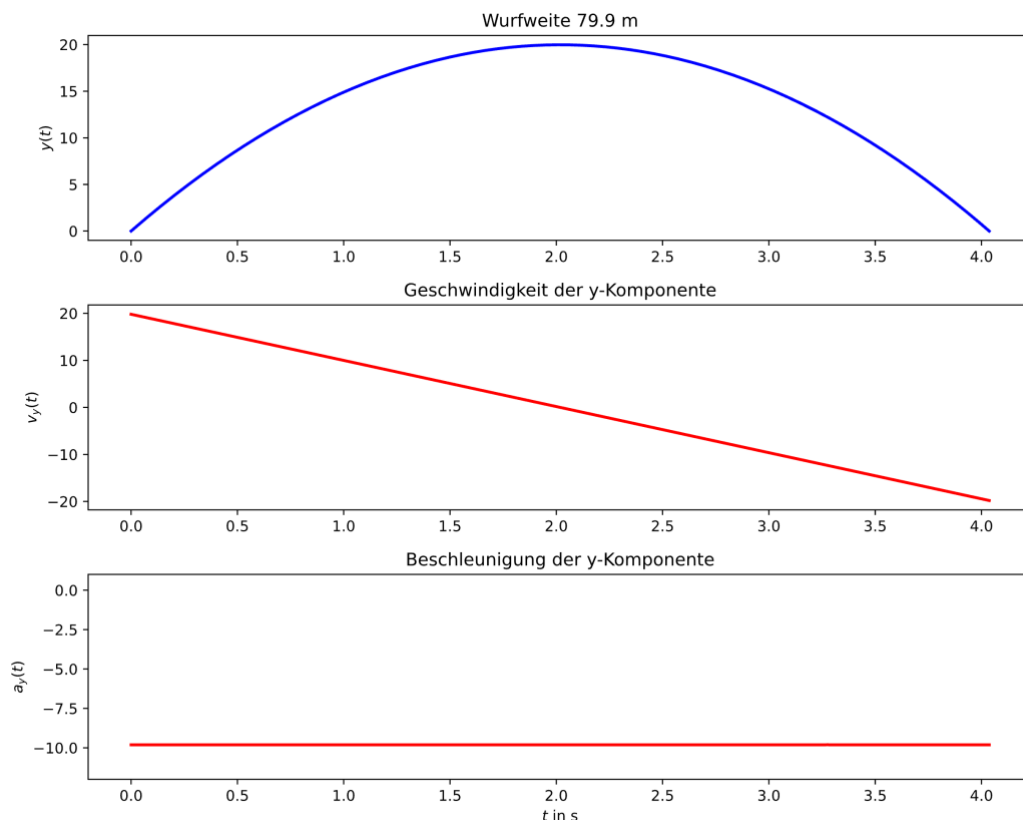
Mit der ersten Ableitung nach der Zeit erhält man die y-Komponente der Geschwindigkeit:

$$\dot{y} = v_y = v_0 \sin \alpha - g \cdot t$$

Die zweite Ableitung der Wurfparabel nach der Zeit ergibt die Erdbeschleunigung:

$$\ddot{y} = a_y = -g$$

Das Matplotlib-Programm `diff_wurfparabel.py` erzeugt die folgende Darstellung des schiefen Wurfs.



Die x-Komponente der Geschwindigkeit ist konstant.

```
#diff_wurfparabel.py
import numpy as np
import matplotlib.pyplot as plt
#Daten
alpha=45 #Abwurfwinkel
v0=28    #Afangsgeschwindigkeit
g=9.81   #Erdbeschleunigung
alpha=np.radians(alpha)
xw=v0**2*np.sin(2*alpha)/g      #Wurfweite
tw=2*v0*np.sin(alpha)/g        #Wurfzeit
hw=v0**2*np.sin(alpha)**2/(2*g) #Wurfhöhe
#zentraler Differenzenquotient
def diff(t,f,h=1e-6):
    return (f(t+h)-f(t-h))/(2*h) #1
#Wurfhöhe: y=f(t)
def y(t):
    return v0*np.sin(alpha)*t - 1/2*g*t**2
#Geschwindigkeit der y-Komponente
def v(t):
    return diff(t,y)
    #return v0*np.sin(alpha) - g*t #zur Kontrolle
#Beschleunigung
def a(t):
    return diff(t,v)
#Grafikbereich
fig,ax=plt.subplots(3,1,figsize=(10,8),label='Schiefer Wurf')
t=np.linspace(0,tw,500)
ax[0].plot(t,y(t),'b-',lw=2)
ax[1].plot(t,v(t),'r-',lw=2)
ax[2].plot(t,a(t),'r-',lw=2)
ax[2].set_ylim(-12,1) #2
#Beschriftungen
ax[0].set_title('Wurfweite %2.1f m' %xw)
ax[1].set_title('Geschwindigkeit der y-Komponente')
ax[2].set_title('Beschleunigung der y-Komponente')
ax[0].set_ylabel(r'$y\left( t \right)$')
ax[1].set_ylabel(r'$v_y\left( t \right)$')
ax[2].set_ylabel(r'$a_y\left( t \right)$')
ax[2].set_xlabel('$t$ in s')
fig.tight_layout()
plt.show()
```

Die Ableitungen werden mit dem zentralen Differenzenquotienten (#1) berechnet. Es ist wichtig den Darstellungsbereich der Beschleunigung mit

`ax[2].set_ylim(-12,1)`

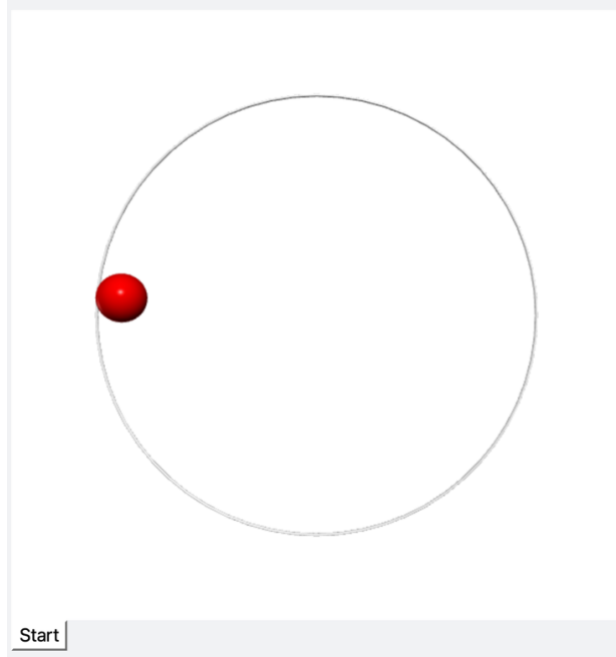
zu begrenzen (#2). Ansonsten würde man keine glatte Gerade für die Beschleunigung erhalten.

5 Dynamik

Erstes Newtonsche Axiom

Mit dem Programm *vp_traegheitsaxiom.py* kann das erste Newtonsche Axiom veranschaulicht werden.

Erstes Newtonsche Axiom: Trägheitsprinzip



Eine Kugel bewegt sich reibungsfrei auf einer Kreisbahn. Sie können die Animation jederzeit stoppen und wieder neu starten. Die Animation ist nicht realistisch, sie bildet ideale Verhältnisse nach. Denkt man aber an Planetenbewegungen, dann zeigt die bisherige Erfahrung, dass sich Planeten ungebremsst im Weltraum bewegen.

```
#vp_traegheitsaxiom.py
from vpython import *
#Daten
breite=5.  #Breite der Zeichenfläche
R=8.       #Radius der Umlaufbahn
r=1        #Radius der Kugel
w=1        #Winkelgeschwindigkeit
dt=1e-2    #Zeitschrittweite
#boolesche Variable
run = True
# Animation anhalten
def startStop(b):
    global run
    run = not run
    if run: b.text = "Stop"
    else: b.text = "Start"
#Grafikbereich
scene.title="<h3>Erstes Newtonsche Axiom: Trägheitsprinzip</h3>"
scene.width=600
scene.height=600
scene.range=2.5*breite
scene.background = color.white
scene.userzoom = False
#Bahn
orbit=ring(pos=vec(0,0,0),axis=vec(0,0,1),thickness=0.05)
orbit.radius=R + r
masse = sphere(radius=r,color=color.red)
# Start-Stop-Button
button(bind=startStop,text="Stop",pos=scene.caption_anchor )
#Animationsschleife
t=0
while True:
```

```

rate(100)
if run:
    x=R*sin(w*t)          #1
    y=R*cos(w*t)          #2
    masse.pos = vector(x,y,0) #3
    t+=dt

```

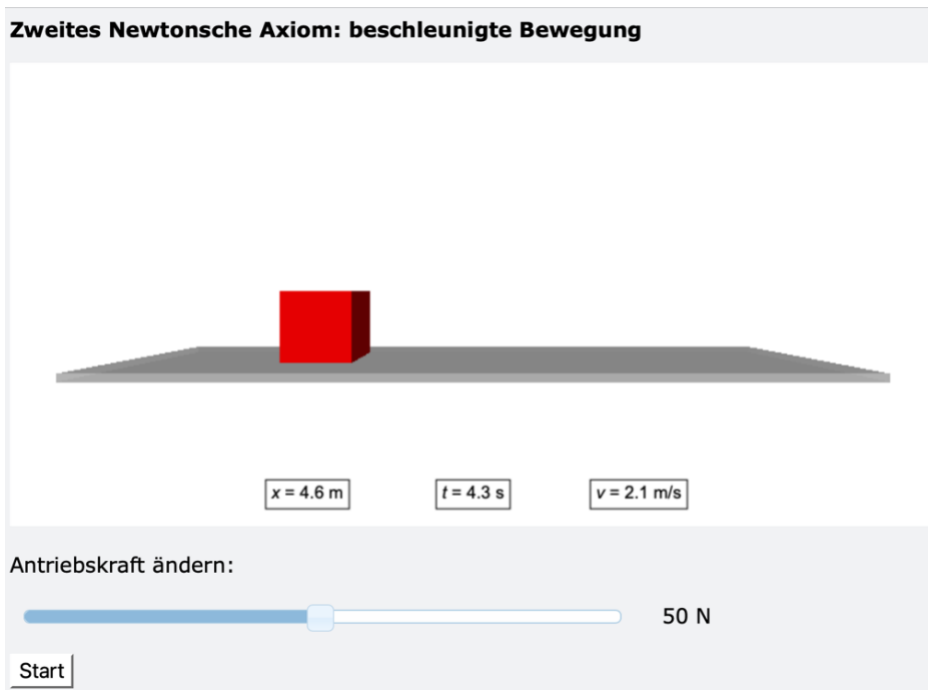
In #1 und #2 werden die x- und y-Komponenten einer Kreisbahn berechnet. In #3 wird die Position der Masse *masse* stetig geändert.

Zweites Newtonsche Axiom

Die Beschleunigung eines Körpers ist proportional zu der Antriebskraft F und umgekehrt proportional zu seiner Masse m :

$$a = \frac{F}{m}$$

Mit dem VPython-Programm *vp_actio.py* können Sie eine beschleunigte Bewegung veranschaulichen.



Wenn die Antriebskraft vergrößert wird, vergrößert sich auch die Beschleunigung. Mit dem Programm können Sie im Unterricht das Weg-Zeit-Gesetz für eine beschleunigte Bewegung experimentell nachweisen: Sie starten und stoppen das Programm zu bestimmten Zeitpunkten, tragen die Werte für die Zeit, den zurückgelegten Weg und die Geschwindigkeit in eine Tabelle ein. Mithilfe dieser Daten können Sie das Geschwindigkeits-Zeit und das Weg-Zeit-Diagramm erstellen oder von den Schülern erstellen lassen.

```

#vp_actio.py
from vpython import *
#Daten
m=100          #Masse in kg
dt = 1e-2      #Zeitschrittweite
breite=20      #Breite der Bodenplatte
dicke=0.2      #Dicke der Bodenplatte
tiefe=5        #Tiefe der Bodenplatte
kasten=2       #Breite des Wuerfels

```

```

#boolesche Variablen
run = False
#Animation anhalten
def startStop(e):
    global run
    run = not run
    if run: e.text = "Stop"
    else: e.text = "Start"
# Wert der Kraft anzeigen
def kraft(s):
    txtKraft.text = '\t %2.0f' %s.value
#Grafikbereich
scene.title="<h3>Zweites Newtonsche Axiom: beschleunigte Bewegung</h3>"
scene.width =800 #Standardwert: 640 Pixel
scene.height=400 #Standardwert: 400 Pixel
scene.center=vector(0,10*dicke,0)
scene.background = color.white
scene.userzoom = False      #kein Zoom
#scene.userspin = False
scene.range=0.35*breite
#Unterlage und Würfel, x0,y0,z0,Laenge, Hoehe, Tiefe
unterlage=box(size=vec(breite,dicke,tiefe),opacity=0.5)
unterlage.pos=vector(0,0,0) #Position der Unterlage
#zu bewogender Koerper
koerper=box(size=vector(kasten,kasten,kasten),color=color.red) #Masse
koerper.pos = vector(-breite/2+kasten/2,kasten/2+dicke/2,0)
#-----
#***Slider für Wegänderung***
scene.caption = "\nAntriebskraft &auml;ndern: \n\n" #Beschriftung Slider
#Sliderobjekt für Kraft erzeugen
sldKraft=slider(bind=kraft,min=1,max=100,value=50,length=550,right=15) #1
#Textfeld für Ausgabe der Kraft
txtKraft = wtext(text='\t %2.0f' %sldKraft.value)
#Einheit der Kraft
scene.append_to_caption(" N\n\n")
#-----
#***Button für Start, Stop***
button(bind=startStop,text="Start",pos=scene.caption_anchor)
#-----
#***Label für Ausgaben***
lblWeg = label()
lblZeit = label()
lblGeschwindigkeit = label()
lblWeg.pos = unterlage.pos + vector(-5,-4,0)
lblZeit.pos = unterlage.pos + vector(0,-4,0)
lblGeschwindigkeit.pos = unterlage.pos + vector(5,-4,0)
#*****
t=x=v=0
#***Animation***
while True:
    rate(1/dt)
    if run:
        F = sldKraft.value #2
        a = F/m            #3
        v = a*t            #4
        x = a*t**2/2        #5
        koerper.pos = vec(x - breite/2 + kasten/2,dicke/2 + kasten/2,0)
        t+=dt
    if x >= (breite - kasten):
        t=0
#Textausgabe
lblWeg.text = '<em>x</em> = %.1f m' %x
lblZeit.text = '<em>t</em> = %.1f s' %t

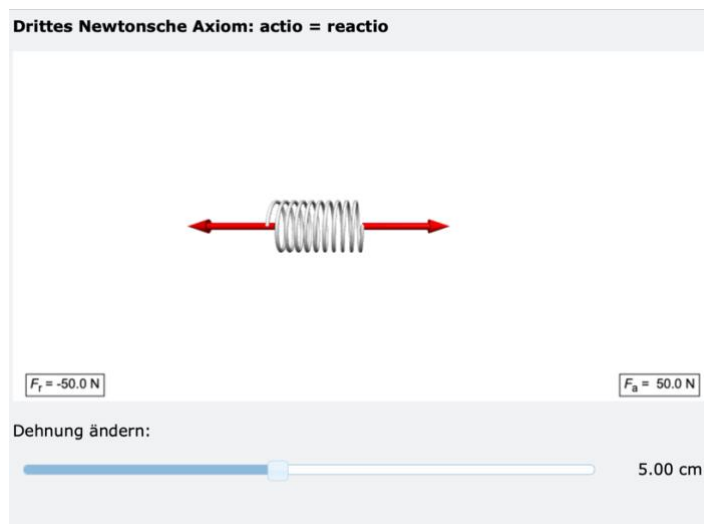
```

```
lblGeschwindigkeit.text = '<em>v</em> = %.1f m/s' %v
```

In #1 wird das Slider-Objekt `sldKraft` erzeugt. In #2 wird der Wert der Slider-Einstellung ausgelesen und in die Variable `F` gespeichert. In #3 wird die aktuelle Beschleunigung berechnet. In #4 wird die Geschwindigkeit berechnet und in #5 erfolgt die Berechnung des zurückgelegten Weges.

Drittes Newtonsche Axiom

Mit dem VPython-Programm `vp_actio_reactio.py` können Sie das dritte Newtonsche Axiom veranschaulichen.



Wenn Sie den Slider-Knopf nach rechts verschieben, vergrößern Sie die Kraft F_a (*actio*). Damit die Feder im Gleichgewicht bleibt, sich also nicht bewegt, muss eine in entgegengesetzter Richtung betragsmäßig gleiche Kraft F_r (*reactio*) wirken.

```
#vp_actio_reactio.py
from vpython import *
kf=10    #Federkonstante N/cm
x0=5     #Startwert für Dehnung
#Funktionsdefinition für Dehnung
def weg(s):
    pfeilFa.pos=feder.pos + vector(s.value,0,0)    #1
    pfeilFa.axis.x = s.value #nach rechts          #2
    pfeilFr.axis.x = -s.value #nach links
    feder.axis.x = s.value
    F=kf*s.value #Federkraft
    lblFa.text = '<i>F</i><sub>a</sub> = %2.1f N' %F #actio
    lblFr.text = '<i>F</i><sub>r</sub> = -%2.1f N' %F #reactio
    txtWeg.text = '\t %2.2f' %s.value
#Graphikbereich
scene.title="<h3>Drittes Newtonsche Axiom: actio = reactio</h3>"
scene.width=800
scene.height=400
scene.center=vector(0,0,0)
scene.range=10
scene.background = color.white
scene.userzoom = False    #kein Zoom
scene.userspin = False
#Feder
feder=helix(axis=vector(x0,0,0),radius=1.4)
feder.thickness=0.2
feder.coils=10
feder.pos=vector(-x0,0,0)
```

```

#Pfeil actio, zeigt nach rechts
pfeilFa=arrow(color=color.red,round=True)
pfeilFa.axis=vec(x0, 0, 0)
pfeilFa.shaftwidth=0.4
pfeilFa.pos=feder.pos + vec(x0,0,0)
#Pfeil reactio, zeigt nach links
pfeilFr=arrow(color=color.red,round=True)
pfeilFr.axis=vec(-x0, 0, 0)
pfeilFr.shaftwidth=0.4
pfeilFr.pos=feder.pos
#Label für Ausgabe der Federkräfte
lblFa=label(pos=vec(+17,-9,0),text='<i>F</i><sub>a</sub> = %2.1f N' %50)
lblFr=label(pos=vec(-17,-9,0),text='<i>F</i><sub>r</sub> = -%2.1f N' %50)
#****Slider für Wegänderung****
scene.caption = "\nDehnung ändern: \n\n"
#Sliderobjekt für Dehnung erzeugen
sldWeg=slider(bind=weg, min=1, max=10,value=x0,length=700) #3
#Textfeld für Ausgabe der Dehnung
txtWeg=wtext(text='\t %2.2f' %sldWeg.value)
#Einheit des Wegs
scene.append_to_caption(" cm\n")

```

In #3 erzeugt die VPython-Methode `slider(bind=weg, ...)` das Objekt `sldWeg`. Die selbstdefinierte Python-Funktion `weg` (#1) wird aufgerufen. Der vom Slider-Knopf eingestellte Zahlenwert wird in #2 dem `vector(s.value,0,0)` als Argument übergeben.

Schiefer Wurf mit Luftwiderstand

Der schiefe Wurf mit Luftwiderstand kann durch das folgende nichtlineare DGL-System beschrieben werden:

$$m \frac{dv_x}{dt} = -b \cdot v_x \sqrt{v_x^2 + v_y^2}$$

$$m \frac{dv_y}{dt} = -g \cdot m - b \cdot v_y \sqrt{v_x^2 + v_y^2}$$

Für die Abkürzung b gilt:

$$b = \frac{1}{2} \rho \cdot c_w \cdot A$$

Mit dem VPython-Programm `vp_realer_wurf.py` können Sie den schiefen Wurf mit Luftwiderstand animieren.



Sie können das Programm jederzeit stoppen, die Werte für die x- und y-Koordinaten in eine Tabelle eintragen, und mithilfe dieser Daten die Wurfbahn als x-y-Diagramm darstellen.

```
#vp_realer_wurf.py
from vpython import *
#Daten
v0=42      #Anfangsgeschwindigkeit
m=0.45     #Masse der Kugel in kg
d=0.22     #Durchmesser der Kugel in Meter
cw=0.2     #Stroemungswiderstandskoeffizient
rho=1.225  #Luftichte in kg/m^3
alpha0=45
dt=1e-2    #Zeitschrittweite
g = 9.81
laenge= 100 #Laenge der Bezugsebene
dicke = 1   #Bodenplatte
R = 1      #Radius fuer Ball (nur fuer Darstellung)
y0 = R + dicke/2
#Berechnungen
A=pi*(d/2)**2      #Querschnitt der Kugel
b=1/2*rho*cw*A     #Reibungskonstante
#boolesche Variable
run = False
#starten oder stoppen
def startStop(e):
    global run
    run = not run
    if run:
        e.text = "Stop"
    else:
        e.text= "Start"
#fuer Ausgabe des Abwurfwinkels
def winkel(e):
    txtA.text = "%2.0f" %e.value
#Grafikbereich
scene.title="<h3>Schiefer Wurf mit Luftwiderstand </h3>"
scene.background=color.white
```

```

scene.width= 800 #Standardwert: 640 Pixel
scene.height=500 #Standardwert: 400 Pixel
scene.userzoom = False      #kein Zoom
scene.center=vector(0,laenge/5,0) #Position des Beobachters
scene.range=0.5*laenge
#Ball
ball = sphere(radius=R,color=color.yellow)
ball.pos=vec(-laenge/2,y0,0)
#Ebene
ebene=box(size=vec(laenge,dicke,laenge/2),color=color.green)
ebene.pos=vec(0,0,0)
#Slider fuer Abwurfwinkel
scene.append_to_caption("\n\n")
scene.caption="\n  Abwurfwinkel &uml;ndern:\n\n"
sldWinkel=slider(bind=winkel,min=20,max=80,value=alpha0,length=780)
txtA=wtext(text="%2.1f" %sldWinkel.value)
scene.append_to_caption(" °\n\n") #Winkel in Grad
#Start-Stop-Button
cmdS=button(bind=startStop,pos=scene.caption_anchor)
cmdS.text="Start"
#Textausgaben
lblX=label()
lblY=label()
lblZeit=label()
lblGeschwindigkeit=label()
yu=-20 #Abstand der Ausgaben
lblZeit.pos = ebene.pos + vec(-30,yu,0)
lblX.pos = ebene.pos + vec(-10,yu,0)
lblY.pos = ebene.pos + vec(10,yu,0)
lblGeschwindigkeit.pos = ebene.pos + vec(30,yu,0)
#Animationsschleife
y=h=y0
t=x=tw=xw=v=0
alpha=radians(alpha0)
vx=v0*cos(alpha) #Anfangswert fuer vx
vy=v0*sin(alpha) #Anfangswert fuer vy
while True:
    rate(30)
    if not run:
        t=x=vx=vy=0
        y=y0
        alpha=radians(sldWinkel.value)
        vx=v0*cos(alpha)
        vy=v0*sin(alpha)
    if run:
        v=sqrt(vx**2 + vy**2)
        vx = vx - b*v*x*v*dt/m          #1
        vy = vy - g*dt - b*v*y*v*dt/m  #2
        x = x + vx*dt                    #3
        y = y + vy*dt                    #4
        ball.pos = vec(x - laenge/2,y,0)#5
        h=y          #Wurfhoehe
        xw=x          #Wurfweite
        tw=t          #Wurfzeit
        t+=dt
        if y < y0: #Ball erreicht den Boden
            t=x=vx=vy=0
            y=y0
            run=False
            if run == False:
                cmdS.text="Start"
#Ausgaben
lblZeit.text = '<em>t</em> = %.1f s' %tw

```

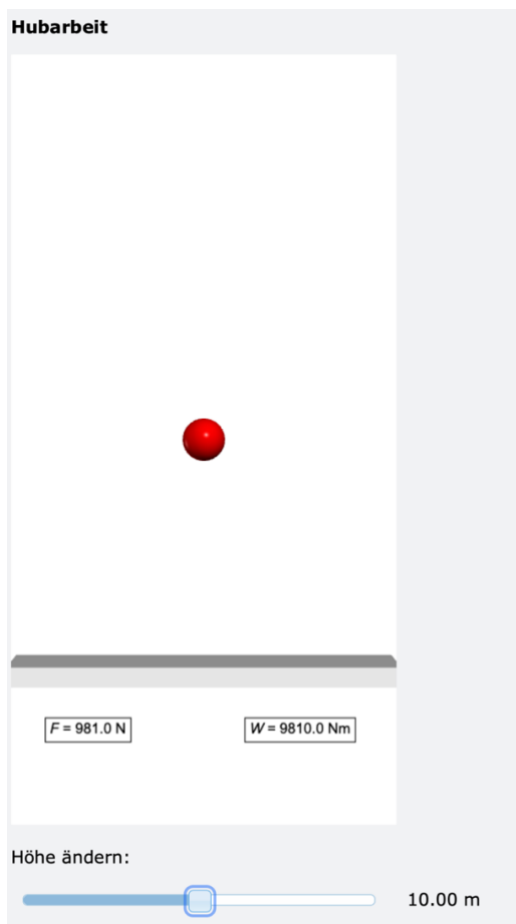
```
lblX.text = '<em>x</em> = %2.1f m' %xw
lblY.text = '<em>y</em> = %2.1f m' %abs(h-y0)
lblGeschwindigkeit.text = '<em>v</em> = %2.1f m/s' %
```

Das DGL-System wird mit dem Euler-Verfahren gelöst (#1 bis #4). In #1 und #2 werden die x- und y-Geschwindigkeitskomponenten durch numerische Integration (Summenalgorithmus) berechnet. In #3 und #4 werden mithilfe dieser Geschwindigkeitskomponenten die x- und y-Koordinaten des Wurfs durch numerische Integration berechnet. In #5 wird die Wurfbewegung auf den Bildschirm dargestellt.

6 Arbeit, Leistung und Energie

Hubarbeit

Hubarbeit wird immer dann verrichtet, wenn Körper angehoben werden. Man denke an einen Kran auf einer Baustelle.



Die Bodenplatte legt das Nullniveau fest. Die Erdanziehungskraft ist proportional zur Masse. Je höher der Körper angehoben wird, desto größer ist die an dem Körper verrichtete Arbeit.

Für die Hubarbeit gilt:

$$W_h = m \cdot g \cdot h$$

Die Animation des VPython-Programms *vp_hubarbeit.py* visualisiert die Hubarbeit.

Durch Verschieben des Slider-Knopfes können Sie die Höhe verändern. Es wird eine Masse von 100 kg gehoben.

Die Kraft F und der aktualisierte Wert der verrichteten mechanischen Arbeit W wird unten rechts auf der Benutzeroberfläche des Programms angezeigt.

In Zeile 3 können Sie die Masse ändern, um so den Zusammenhang von Masse und Hubarbeit quantitativ zu veranschaulichen.

```
#vp_hubarbeit.py
from vpython import *
m=100 #Masse in kg
g=9.81 #Erdbeschleunigung
#Höhe ändern
def hoehe(s):
    h=sldHoehe.value #Höhe ermitteln
    masse.pos.y = h #Masse anheben
```

```

F=m*g          #Kraft berechnen
W=F*h          #Arbeit berechnen
lblF.text = '<i>F</i> = %2.1f N' %F #Kraft
lblW.text = '<i>W</i> = %2.1f Nm' %W #Arbeit
txtHoehe.text = '\t %2.2f' %s.value
#Grafikbereich
scene.title="<h3>Hubarbeit</h3>"
scene.width=400
scene.height=800
scene.range=10
scene.center=vector(0,10,0)
scene.background = color.white
scene.userzoom = False    #kein Zoom
scene.userspin = False
#Boden, Bezugsebene
platte=box(length=20, height=1, width=2)
platte.pos=vec(0,-2,0)
#Masse
masse=sphere(radius=1.1,color=color.red)
masse.pos=vec(0,0,0)
#Label für Ausgabe der Kraft und Hubarbeit
lblF=label(pos=vec(-6,-5,0),text='F = %2.2f °' %0)
lblW=label(pos=vec( 5,-5,0),text='W = %2.2f °' %0)
#***Slider für Höhenänderung***
scene.caption = "\nHöhe ändern: \n\n" #Beschriftung des Sliders
#Sliderobjekt für Höhe erzeugen
sldHoehe = slider(bind=hoehe,min=0,max=20,value=0,length=400,right=15) #2
#Textfeld für Ausgabe der Dehnung
txtHoehe = wtext(text='\t %2.2f' %sldHoehe.value)
#Einheit des Wegs
scene.append_to_caption(" m\n")

```

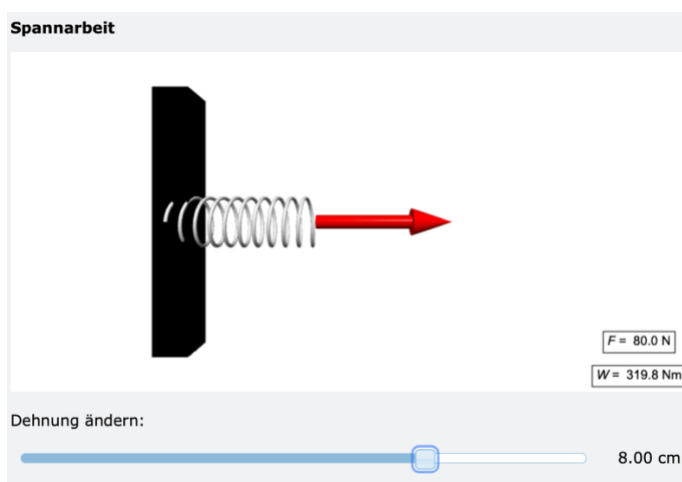
In #2 wird das Slider-Objekt `sldHoehe` erzeugt. Die selbstdefinierte Python-Funktion `hoehe` aus #1 wird aufgerufen (ohne den Parameter `s`). Die Höhe der Masse `masse` kann zwischen 0 und 20 m mithilfe des Slider-Knopfes verstellt werden.

Spannarbeit

Für die Spannarbeit gilt:

$$W_s = \frac{1}{2} k_f \cdot s^2$$

Mit dem VPython-Programm `vp_spannarbeit.py` kann die Spannarbeit veranschaulicht werden.



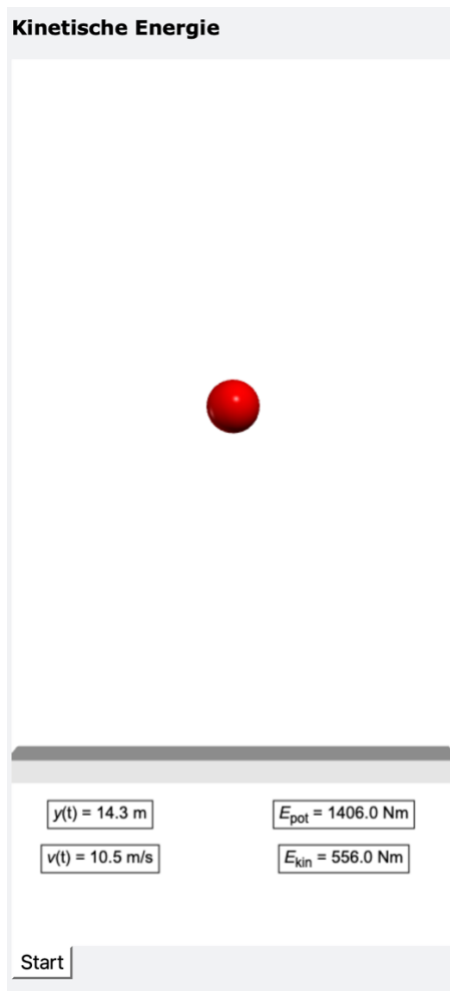
Durch Verschieben des Slider-Knopfes wird die Dehnung geändert. Die Werte für die Federkraft und der Spannarbeit werden aktualisiert.

```
#vp_spannarbeit.py
from vpython import *
kf=10    #Federkonstante N/cm
x0=5
#Dehnung ändern
def weg():                                #1
    s=sldWeg.value                        #2
    feder.axis.x = s                      #3
    pfeilF.pos=feder.pos + vector(s,0,0)
    pfeilF.axis.x = s
    F = kf*s    #Federkraft berechnen
    W = F*s/2    #Spannarbeit berechnen
    lblF.text = '<i>F</i> = %2.1f N' %F    #Kraft
    lblW.text = '<i>W</i> = %2.1f Nm' %W    #Spannarbeit
    txtWeg.text = '\t %2.2f' %s
#Zeichenfläche
scene.title="<h3>Spannarbeit</h3>"
scene.width=800
scene.height=400
scene.center=vector(0,0,0)
scene.range=10
scene.background = color.white
scene.userzoom = False    #kein Zoom
scene.userspin = False
#Wand für Befestigung der Schraubenfeder
wand=box(pos=vec(-10,0,0),length=2,height=15,width=2,color=color.black)
#Schraubenfeder
feder=helix(axis=vector(x0,0,0),radius=1.4)
feder.thickness=0.2
feder.coils=10
feder.pos=wand.pos
#Pfeil Zugkraft
pfeilF=arrow(pos=vec(0, 0, 0),axis=vec(x0, 0,
0),color=color.red,round=True)
pfeilF.pos=feder.pos + vector(x0,0,0)
#Label für Ausgabe der Federkräfte
lblF=label(pos=vec(17,-7,0),text='F = %2.2f °' %0)
lblW=label(pos=vec(17,-9,0),text='W = %2.2f °' %0)
****Slider für Wegänderung****
scene.caption = "\nDehnung ändern: \n\n" #Beschriftung des Sliders
#Sliderobjekt für Dehnung erzeugen
sldWeg = slider(bind=weg,min=3,max=10,value=x0,length=700,right=15) #4
#Textfeld für Ausgabe der Dehnung
txtWeg = wtext(text='\t %2.2f' %sldWeg.value)
#Einheit des Wegs
scene.append_to_caption(" cm\n")
```

In #4 erzeugt die VPython-Methode `slider(bind=weg, ...)` das Objekt `sldWeg`. Die selbstdefinierte Python-Funktion `weg` (#1) wird aufgerufen. In #2 wird die Dehnung `s` aktualisiert. In #3 wird das Federende um diesen Wert auf der x-Achse verschoben.

Kinetische Energie

Wenn ein Stein von einem Turm fallengelassen wird, wird seine potentielle Energie in kinetische Energie umgewandelt.



Das VPython-Programm *vp_kinetische_energie.py* animiert den freien Fall eines Körpers und berechnet seine potentielle und kinetische Energie.

Wenn Sie auf den Start-Button klicken, fällt der Körper aus einer Höhe von 20 m. Seine Geschwindigkeit steigt proportional zur Fallzeit:

$$v = g \cdot t$$

Die kinetische Energie steigt proportional zum Quadrat der Geschwindigkeit:

$$E_{kin} = \frac{1}{2} m \cdot v^2$$

Die Summe aus kinetischer und potentieller Energie muss zu jedem Zeitpunkt konstant sein:

$$m \cdot g \cdot y + \frac{1}{2} m \cdot v^2 = \text{const.}$$

Für den zurückgelegten Weg gilt:

$$y = h - \frac{1}{2} g \cdot t^2$$

Links unten werden der zurückgelegte Weg und die Momentangeschwindigkeit; rechts unten werden die aktuellen Werte für die potentielle und kinetische Energie angezeigt. Sie können die Animation jederzeit anhalten, um zu überprüfen, ob die Summe aus potentieller und kinetischer Energie konstant ist.

In Zeile 4 können Sie einen anderen Wert für die Masse eintragen.

```
#vp_kinetische_energie.py
from vpython import *
#Daten
m = 10 #Masse in kg
h = 20 #Fallhöhe
r = 1.2 #Radius der Kugel
g=9.81 #Erdbeschleunigung
# Animation anhalten
def startStop(b):
    global run
    run = not run
    if run: b.text = "Stop"
    else: b.text = "Start"
#Grafikbereich
scene.title="<h3>Kinetische Energie</h3>"
scene.width=400
scene.height=800
scene.range=10
scene.center=vector(0,10,0)
scene.background = color.white
scene.userzoom = False #kein Zoom
scene.userspin = False
#Boden, Bezugsebene
platte=box(length=20, height=1, width=2)
platte.pos=vec(0,-1.5*r,0)
#Masse
```

```

kugel=sphere(radius=r,color=color.red)
kugel.pos=vec(0,0,0)
#Label für Ausgabe der Kraft und Hubarbeit
lblY = label(pos=vec(-6,-4,0),text='')
lblV = label(pos=vec(-6,-6,0),text='')
lblEpot=label(pos=vec(5,-4,0),text='')
lblEkin=label(pos=vec(5,-6,0),text='')
# boolesche Variablen
run = True
# Start-Stop-Button
button(bind=startStop,text="Stop",pos=scene.caption_anchor )
dt=1e-6 #Zeitschrittweite
t=0
tmax= sqrt(2*h/g) #Fallzeit für Zeile 56
#Animationsschleife
while True:
    rate(1e3/dt)
    if run:
        y = h - g*t**2/2          #1
        v = g*t                    #2
        t+=dt
        kugel.pos.y = y            #3
        #Energie
        Epot=m*g*y                 #potentielle Energie berechnen
        Ekin = m*v**2/2            #kinetische Energie berechnen
        #Ausgaben
        lblY.text = '<i>y</i>(t) = %2.1f m' %y    #aktuelle Fallhöhe
        lblV.text = '<i>v</i>(t) = %2.1f m/s' %v    #aktuelle Geschwindigkeit
        lblEpot.text = '<i>E</i><sub>pot</sub> = %2.1f Nm' %Epot
        lblEkin.text = '<i>E</i><sub>kin</sub> = %2.1f Nm' %Ekin
        if t >= tmax: #alternativ y <= 0
            t=0

```

In #1 wird die y-Komponente des fallenden Körpers, also die senkrechte Bewegung des Körpers, berechnet. In #2 wird seine Geschwindigkeit berechnet. Die Anweisung in #3 animiert die Fallbewegung. Der Eiffelturm hat eine Höhe von 312 m. Welchen Wert hat die kinetische Energie für diese Fallhöhe.

Animation eines Kugelpendels

Die Bewegung einer Kugel, die sich einer Kreisschale hin und her bewegt, kann durch das folgende DGL-System beschrieben werden:

$$\frac{d\varphi}{dt} = \omega$$

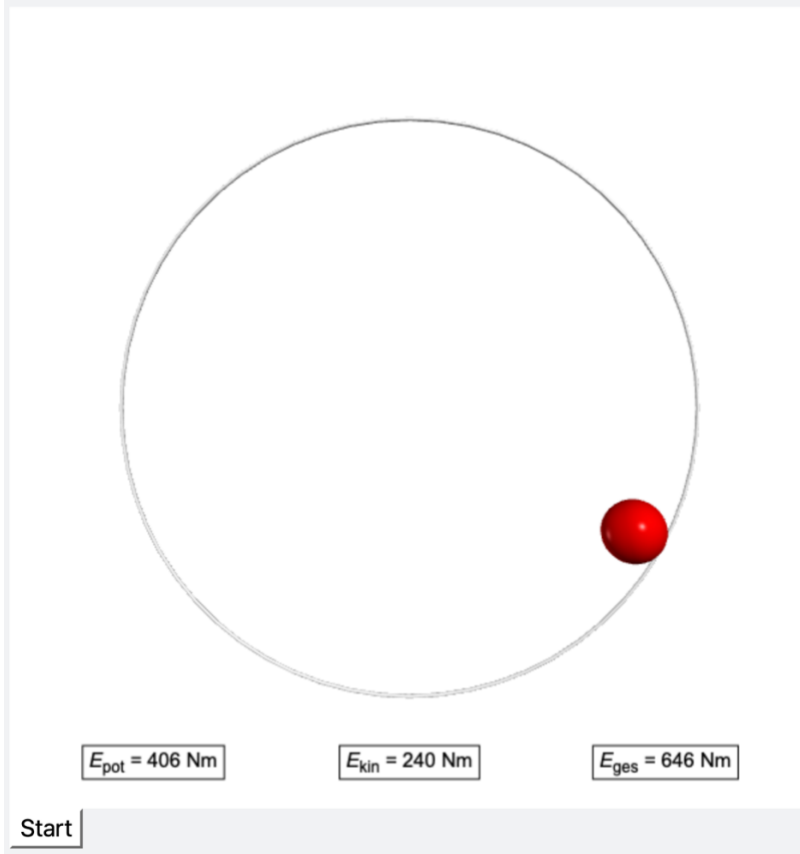
$$\frac{d\omega}{dt} = -\omega_0^2 \sin \varphi$$

Mit

$$\omega_0^2 = \frac{g}{l}$$

Die Variable l soll den Radius der Kugelschale repräsentieren. Je größer dieser Radius ist, desto kleiner ist die Winkelgeschwindigkeit der Kugel.

Kugelpendel



Sie können die Animation jederzeit anhalten. Die Summe aus potentieller und kinetischer Energie muss zu jedem Zeitpunkt konstant sein. Abweichungen entstehen durch Rundungsfehler.

```
#vp_kugelpendel.py
from vpython import *
m=10          #kg
g=9.81        #Erdbeschleunigung
breite=5.     #Breite der Zeichenfläche
l=8.          #Länge des Pendels = Radius der Kreisschale
phi=80.       #Auslenkung
r=1           #Radius der Kugel
run = True
# Animation anhalten
def startStop(b):
    global run
    run = not run
    if run: b.text = "Stop"
    else: b.text = "Start"
#Grafikbereich
scene.title="<h3>Kugelpendel</h3>"
scene.width=600
scene.height=600
scene.range=2.5*breite
scene.background = color.white
scene.userzoom = False
orbit=ring(pos=vec(0,0,0),axis=vec(0,0,1),thickness=0.05)
orbit.radius = l + r
masse = sphere(radius=r,color=color.red)
# Start-Stop-Button
button(bind=startStop,text="Stop",pos=scene.caption_anchor )
#Label-Objekte
```

```

lblEpot=label(pos=vec(-8,-11,0),text='')
lblEkin=label(pos=vec(0,-11,0),text='')
lblEges=label(pos=vec(8,-11,0),text='')
#Quadrat der Kreisfrequenz
w02=g/l
phi=radians(phi)
w=0.      #Anfangswinkelgeschwindigkeit
dt=1e-2   #Zeitschrittweite
while True:
    rate(100)
    if run:
        phi=phi+w*dt          #1
        w=w-w02*sin(phi)*dt  #-0.001*w    #2
        x= l*sin(phi)
        y=-l*cos(phi)
        v=w*l
        h=l+y
        Epot=m*g*h            #potentielle Energie
        Ekin=0.5*m*v**2       #kinetische Energie
        Eges=Epot + Ekin
        masse.pos = vector(x,y,0)
        lblEpot.text = '<i>E</i><sub>pot</sub> = %2.0f Nm' %Epot  #3
        lblEkin.text = '<i>E</i><sub>kin</sub> = %2.0f Nm' %Ekin  #4
        lblEges.text = '<i>E</i><sub>ges</sub> = %2.0f Nm' %Eges  #5

```

Das DGL-System wird mit dem Euler-Verfahren gelöst (#1 und #2). Wenn Sie den Kommentar in #2 entfernen werden die Schwingungen gedämpft. Die Ausgaben der Energien erfolgen in #3, #4 und #5.

7 Der Impuls

Gerader elastischer Stoß

Nach dem Stoß gilt für die Geschwindigkeiten des geraden elastischen Stoßes:

$$u_1 = \frac{(m_1 - m_2)v_1 + 2m_2 \cdot v_2}{m_1 + m_2}$$

$$u_2 = \frac{(m_2 - m_1)v_2 + 2m_1 \cdot v_1}{m_1 + m_2}$$

Es sollen drei Fälle betrachtet werden. Für die Massen gilt: $m_1 = m_2 = 1 \text{ kg}$

Erster Fall: $v_1 = 4 \text{ m/s}$, $v_2 = 0 \text{ m/s}$

$$u_1 = \frac{0 \cdot 4 \text{ m/s} + 2\text{kg} \cdot 0}{2 \text{ kg}} = 0$$

$$u_2 = \frac{0 \cdot 4 \text{ m/s} + 2\text{kg} \cdot 4 \text{ m/s}}{2 \text{ kg}} = 4 \text{ m/s}$$

Zweiter Fall: $v_1 = 4 \text{ m/s}$, $v_2 = -4 \text{ m/s}$

$$u_1 = \frac{0 \cdot 4 \text{ m/s} + 2\text{kg} \cdot -4 \text{ m/s}}{2 \text{ kg}} = -4 \text{ m/s}$$

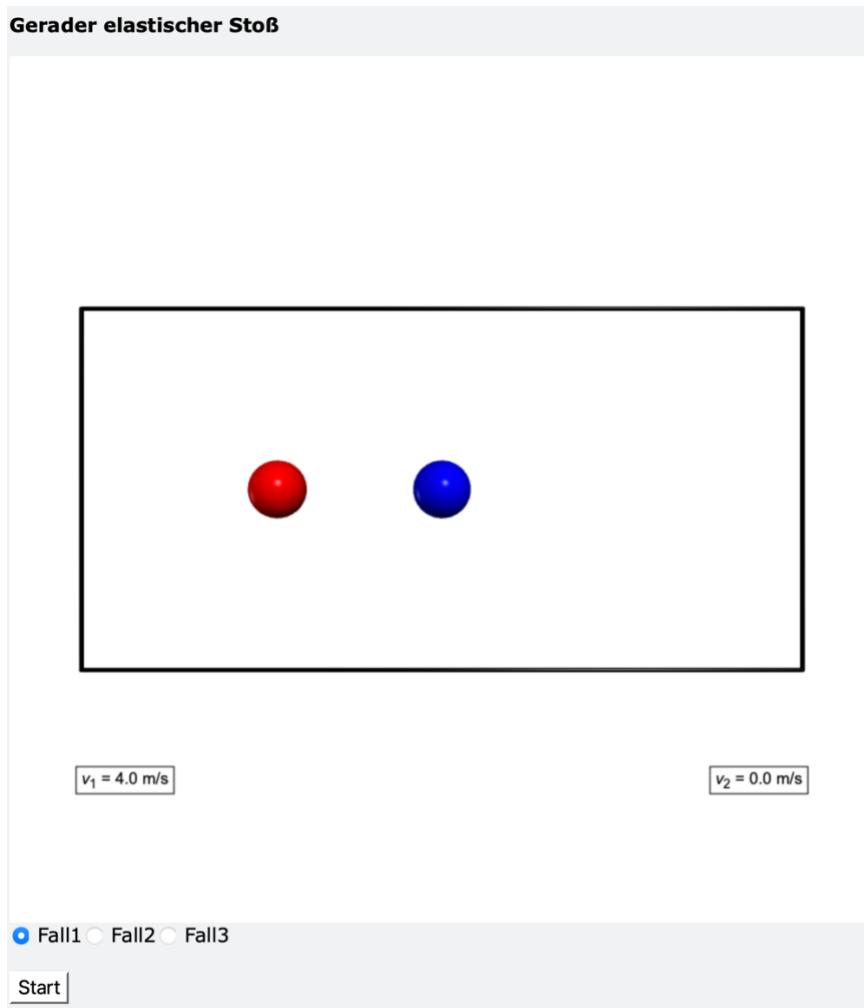
$$u_2 = \frac{0 \cdot -4 \text{ m/s} + 2\text{kg} \cdot 4 \text{ m/s}}{2 \text{ kg}} = 4 \text{ m/s}$$

Dritter Fall: $v_1 = 4 \text{ m/s}$, $v_2 = 1 \text{ m/s}$

$$u_1 = \frac{0 \cdot 4 \text{ m/s} + 2 \text{ kg} \cdot 1 \text{ m/s}}{2 \text{ kg}} = 1 \text{ m/s}$$

$$u_2 = \frac{0 \cdot 1 \text{ m/s} + 2 \text{ kg} \cdot 4 \text{ m/s}}{2 \text{ kg}} = 4 \text{ m/s}$$

Mit dem VPython-Programm `vp_gerader_stoss.py` kann der gerade elastische Stoß animiert werden. Der Screenshot des Programms zeigt die Draufsicht. Es wird angenommen, dass sich die Kugeln reibungsfrei auf der Unterlage bewegen.



Fall1: Nach dem Programmstart bewegt sich die rote Kugel von links nach rechts. Die blaue Kugel befindet sich in der Ruhelage. Nach dem Stoß bewegt sich die blaue Kugel nach rechts und die rote Kugel bleibt stehen.

Fall2: Die rote Kugel bewegt sich nach rechts und die blaue bewegt sich nach links. Nach dem Stoß bewegt sich die rote Kugel nach links und die blaue Kugel bewegt sich nach rechts.

Fall3: Die rote Kugel bewegt sich mit einer Geschwindigkeit von $v_1 = 4 \text{ m/s}$ und die blaue Kugel bewegt sich mit einer Geschwindigkeit von $v_2 = 1 \text{ m/s}$ nach rechts.

```

#vp_geraeder_stoss.py
from vpython import *
#Daten
v1= 4    #Geschwindigkeit v1
v2= 0    #Geschwindigkeit v2
x1=-1    #Anfangsposition für Kugel1: -1,0,1: links, Mitte, rechts
x2= 0    #Anfangsposition für Kugel2: -1,0,1: links, Mitte, rechts
R=0.4    #Radius der Kugeln
b = 5    #Breite = 2*b
h=b/2    #Höhe
RAND = b - R
f=0.75   #Faktor für Eindringtiefe
dt=0.01  #Zeitschrittweite
#Boolesche Variable
run = False
#Entscheidung für Stop oder Start
def startStop(b):
    global run
    run = not run
    if run:
        b.text = "Stop"
    else:
        b.text = "Start"
#Fallunterscheidung
def fall(evt):
    if evt.text=='Fall1':
        optFall12.checked=optFall13.checked=False
        kugel1.pos=RAND*vec(-1,0,0)
        kugel1.v=vec(4,0,0)
        kugel2.pos=RAND*vec(0,0,0) #Mitte
        kugel2.v=v1*vec(0,0,0)
    elif evt.text=='Fall2':
        optFall11.checked=optFall13.checked=False
        kugel1.pos=RAND*vec(-1,0,0)
        kugel1.v=vec(4,0,0)
        kugel2.pos=RAND*vec(1,0,0)
        kugel2.v=vec(-4,0,0)
    elif evt.text=='Fall3':
        optFall11.checked=optFall12.checked=False
        kugel1.pos=RAND*vec(-1,0,0)
        kugel1.v=vec(4,0,0)
        kugel2.pos=RAND*vec(0,0,0)
        kugel2.v=vec(1,0,0)
#Graphikbereich
scene.title="<h3>Gerader elastischer Stoß</h3>"
scene.height=800
scene.width= 800
scene.background = color.white
scene.range=1.2*b
#Kugel1
kugel1=sphere(radius=R,color=color.red)
kugel1.pos=RAND*vec(x1,0,0) #Position links
kugel1.v=v1*vec(1,0,0)      #Geschwindigkeitsvektor
#Kugel2
kugel2=sphere(radius=R,color=color.blue)
kugel2.pos=RAND*vec(x2,0,0) #Position rechts
kugel2.v=v2*vec(1,0,0)      #Geschwindigkeitsvektor
#Rahmen zeichnen
rahmen=[vec(-b,h,0),vec(b,h,0),vec(b,-h,0),vec(-b,-h,0),vec(-b,h,0)]
curve(pos=rahmen,color=color.black)
#Fallunterscheidungen
optFall11 = radio(bind=fall, text='Fall1', checked=True)
optFall12 = radio(bind=fall, text='Fall2')

```

```

optFall13 = radio(bind=fall1, text='Fall13')
#Befehlsschaltfläche
scene.append_to_caption('\n\n')
button(bind=startStop, text="Start", pos=scene.caption_anchor)
#Label für Ausgabe
lblV1 = label(pos=vec(-4.4,-4,0))
lblV2 = label(pos=vec( 4.4,-4,0))
lblV1.text = '<i>v</i><sub>1</sub> = %2.1f m/s' %0
lblV2.text = '<i>v</i><sub>2</sub> = %2.1f m/s' %0
# Animation
while True:
    rate(100)
    if run:
        #Kollisionsprüfung
        abstand = mag(kugel1.pos - kugel2.pos) #1
        if abstand < (f*kugel1.radius + f*kugel2.radius): #2
            d=norm(kugel2.pos - kugel1.pos) #3
            u1=dot(kugel1.v,d) #4
            u2=dot(kugel2.v,d) #5
            richtung=u2-u1 #6
            #wenn richtung positiv nach rechts bewegen
            #wenn richtung negativ nach links bewegen
            kugel1.v=kugel1.v + richtung*d
            #wenn richtung negativ nach rechts bewegen
            #wenn richtung positiv nach links bewegen
            kugel2.v=kugel2.v - richtung*d
        # Kugeln bewegen sich
        kugel1.pos = kugel1.pos + kugel1.v*dt
        kugel2.pos = kugel2.pos + kugel2.v*dt
        # Richtungsumkehr Kugel1
        if not(-RAND <= kugel1.pos.x <= RAND):
            kugel1.v.x = -kugel1.v.x
        # Richtungsumkehr Kugel2
        if not(-RAND <= kugel2.pos.x <= RAND):
            kugel2.v.x = -kugel2.v.x
        #Ausgaben für Geschwindigkeiten
        lblV1.text = '<i>v</i><sub>1</sub> = %2.1f m/s' %mag(kugel1.v)
        lblV2.text = '<i>v</i><sub>2</sub> = %2.1f m/s' %mag(kugel2.v)

```

Die VPython-Funktion `mag()` berechnet den Betrag eines Vektors (#1).

Beispiel: `a=vec(3,4,0); mag(a)` #Ausgabe 5

In #2 wird der Abstand überprüft. Wenn er einen bestimmten Wert unterschreitet, wird der nachfolgende Block ausgeführt.

Die Variablen `d`, `kugel1.v` und `kugel2.v` sind Vektoren (#3). Damit das Vorzeichen für die Bewegungsrichtung `richtung = v2 - v1` (#6) ermittelt werden kann, müssen die Skalarprodukte `v1=dot(kugel1.v, d)` und `v2=dot(kugel2.v, d)` berechnet werden (#4, #5).

Wenn Sie für die Geschwindigkeit `v2=-4` und für die Position `x2=1` festlegen, bewegen sich die Kugeln aufeinander zu. Nach dem Stoß bewegt sich die rote Kugel nach links und die blaue Kugel nach rechts.

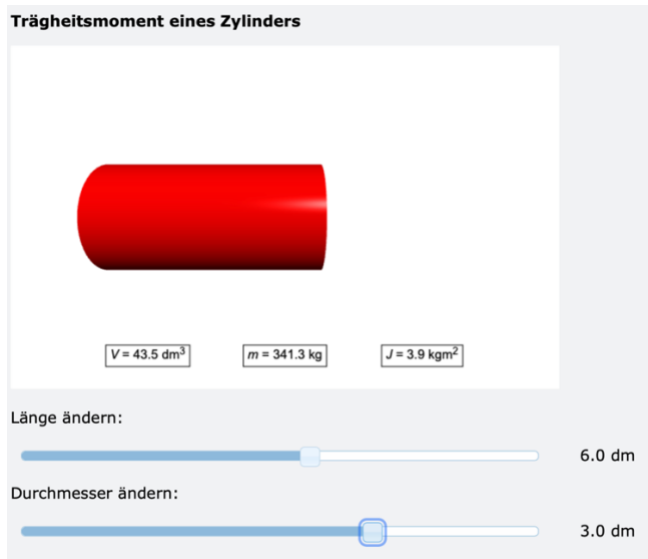
8 Rotation starrer Körper

Das Trägheitsmoment J eines Zylinders wird mit

$$J = \frac{1}{2} m \cdot r^2$$

berechnet.

Mit dem VPython-Programm *vp_slider_zylinder.py* können Sie das Volumen, die Masse und das Trägheitsmoment eines Zylinders berechnen. Die beiden Slider-Steuer-elemente ändern die Länge und den Durchmesser des Zylinders.



Wenn Sie einen der beiden Slider-Knöpfe verschieben, werden die Werte für das Volumen, die Masse und das Trägheitsmoment des Zylinders aktualisiert.

```
#vp_slider_zylinder.py
from vpython import *
#Daten
l0=3
d0=2
rho=7.85 #Dichte von Stahl in kg/dm^3
#Berechnungen durchführen
def calc():
    l=sldL.value #Länge in dm
    d=sldD.value #Durchmesser in dm
    r=d/2.0
    zylinder.length = l
    zylinder.radius = r
    V=pi*r**2*l #Volumen
    m=rho*V #Masse
    J=m*r**2/2*1e-2 #1
    txtL.text=text='\t %2.1f dm' %l
    txtD.text=text='\t %2.1f dm' %d
    lblV.text = '<i>V</i> = %2.1f dm<sup>3</sup>' %V
    lblM.text = '<i>m</i> = %2.1f kg' %m
    lblJ.text = '<i>J</i> = %2.1f kgm<sup>2</sup>' %J
#Grafikbereich
scene.title="<h3>Trägheitsmoment eines Zylinders</h3>"
scene.background = color.white
scene.range=5
#Zylinderobjekt erzeugen
zylinder = cylinder(pos=vec(-5,0,0),color=color.red)
zylinder.axis=vec(10,0,0)
zylinder.radius=d0/2
#Slider: Länge ändern
scene.caption = "\nLänge ändern: \n\n"
sldL = slider(bind=calc,min=1,max=10,value=10,length=650) #2
txtL = wtext(text='\t %2.1f dm' %sldL.value)
```

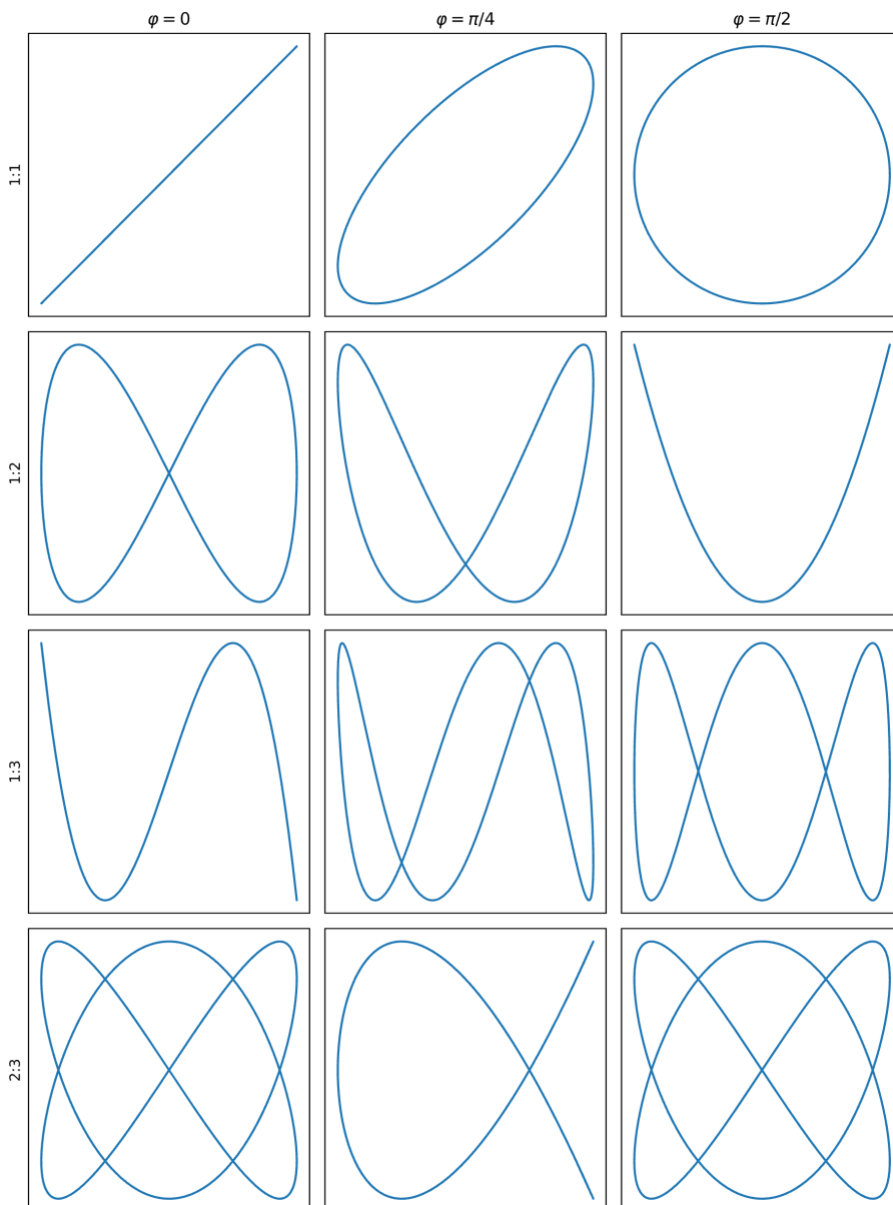
```
#Slider: Durchmesser ändern
scene.append_to_caption("\n\nDurchmesser ändern: \n\n")
sldD = slider(bind=calc,min=1,max=4,value=d0,length=650) #3
txtD = wtext(text='\t %2.0f dm' %sldD.value)
scene.append_to_caption("\n\n")
#Ergebnisse ausgeben
lblV = label(pos=vec(-4,-4,0),text='<i>V</i> = %2.1f dm<sup>3</sup>' %0)
lblM = label(pos=vec(0,-4,0),text='<i>m</i> = %2.1f kg' %0)
lblJ = label(pos=vec(4,-4,0),text='<i>J</i> = %2.1f kgm<sup>2</sup>' %0)
```

Das Trägheitsmoment J wird mit der Einheit $\text{kg}\cdot\text{m}^2$ angegeben. Es muss deshalb von $\text{kg}\cdot\text{dm}^2$ in $\text{kg}\cdot\text{m}^2$ durch Multiplikation mit dem Faktor $0,1^2$ umgerechnet werden (#1). In #2 und #3 wird die Funktion *calc* aufgerufen.

9 Mechanische Schwingungen

Lissajous Figuren

Mit dem Matplotlib-Programm *plot_lissajours.py* werden die 12 Lissajous Figuren erzeugt.



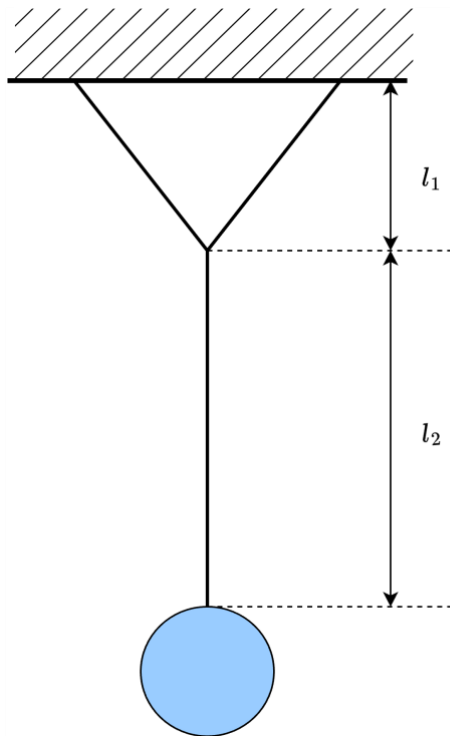
Lissajous Figuren entstehen, wenn man zwei sinusförmige Schwingungen, die **senkrecht** aufeinander stehen, überlagert:

$$x(t) = \sin(\omega t)$$

$$y(t) = \sin(\omega t - \varphi)$$

Die Kreisfrequenzen ω stehen in einem bestimmten Zahlenverhältnis $n:m$. Dieses Verhältnis und die Phasenverschiebung φ bestimmen die Gestalt der Lissajous Figuren.

Erzeugen von Lissajours-Figuren



Wenn das Pendel parallel zur Zeichenebene hin und her schwingt, wirkt nur die Pendellänge l_2 . Für die Winkelgeschwindigkeit gilt:

$$\omega_1 = \sqrt{\frac{g}{l_2}}$$

Wenn das Pendel senkrecht zur Zeichenebene schwingt, wirkt die Pendellänge $l_1 + l_2$. Für die Winkelgeschwindigkeit gilt dann:

$$\omega_2 = \sqrt{\frac{g}{l_1 + l_2}}$$

Die Kugel ist mit feinkörnigem Sand gefüllt. Am unteren Ende Kugel befindet sich ein kleines Loch. Wenn die Kugel mit einem bestimmten Phasenwinkel ausgelenkt wird, zeichnet sie Lissajous Figuren auf einer Bodenplatte.

Auf YouTube finden Sie zahlreiche Videos die dieses Phänomen veranschaulichen.

Mit der Anweisung

```
plot(x(t,n),y(t,m,phi))
```

wird eine Lissajous Figur gezeichnet.

```
#plot_lissajours.py
import numpy as np
import matplotlib.pyplot as plt
#x-Komponente
def x(t,w):
    return np.sin(w*t)
#y-Komponente
def y(t,w,phi):
    return np.sin(w*t - phi)
#Grafikbereich
fig, ax=plt.subplots(4,3,figsize=(9,12),label="Lissajours'sche Figuren")
t=np.linspace(0,2*np.pi,500)
liste=[(0,0,1,1,0),(0,1,1,1,np.pi/4),(0,2,1,1,np.pi/2), #1
        (1,0,1,2,0),(1,1,1,2,np.pi/4),(1,2,1,2,np.pi/2),
        (2,0,1,3,0),(2,1,1,3,np.pi/4),(2,2,1,3,np.pi/2),
        (3,0,2,3,0),(3,1,2,3,np.pi/4),(3,2,2,3,np.pi/2)]
for i,j,n,m,phi in liste: #2
```

```

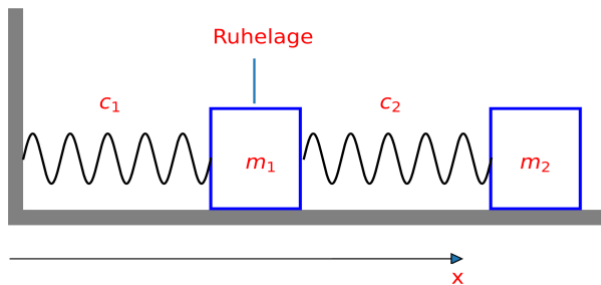
ax[i,j].plot(x(t,n),y(t,m,phi))
ax[i,j].set_xticks([])
ax[i,j].set_yticks([])
#Beschriftungen
ax[0,0].set_title(r'\varphi=0$')
ax[0,1].set_title(r'\varphi=\pi/4$')
ax[0,2].set_title(r'\varphi=\pi/2$')
ax[0,0].set_ylabel('1:1')
ax[1,0].set_ylabel('1:2')
ax[2,0].set_ylabel('1:3')
ax[3,0].set_ylabel('2:3')
fig.tight_layout()
plt.show()

```

Die Liste in #1 enthält die verschiedenen Frequenzverhältnisse $n:m$ und die Phasenverschiebungen. Die `for`-Schleife iteriert über die Zeilen- und Spaltenindizes sowie über die Frequenzverhältnisse und Phasenverschiebungen (#2).

Gekoppeltes Federpendel

Das Feder-Masse-System besteht aus zwei Massen und zwei Schraubenfedern.

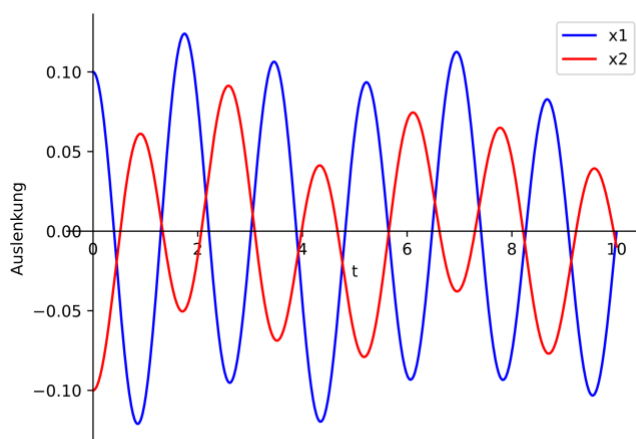


Das Schwingverhalten wird durch das folgende lineare DGL-System beschrieben:

$$m_1 \cdot \ddot{x}_1 + d_1 \cdot \dot{x}_1 + c_1 x_1 - c_2 (x_2 - x_1) = 0$$

$$m_2 \cdot \ddot{x}_2 + d_2 \cdot \dot{x}_2 + c_2 (x_2 - x_1) = 0$$

Das nachfolgende Bild zeigt den Funktionsplot der Lösung für die Anfangsbedingungen $x_1=0.1$ und $x_2=-0.1$.



Das DGL-System kann direkt in den SymPy-Quelltext übernommen werden (#1, #2).

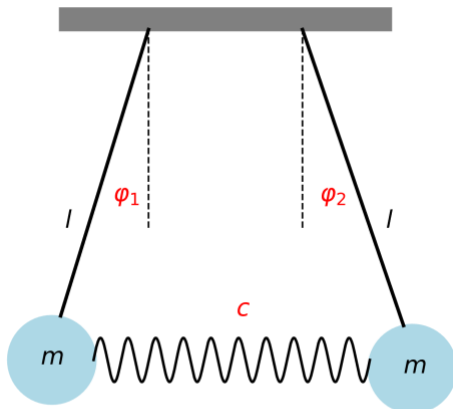
```
#sym_gekoppeltes_federpendel.py
from sympy import symbols, Eq, Function, plot, N
from sympy.solvers.ode.systems import dsolve_system
t = symbols("t")
x1 = Function("x1")(t)
x2 = Function("x2")(t)
m1, m2 = 2, 2          #Massen
c1, c2 = 10, 10        #Federkonstanten
d1, d2 = 0.1, 0.1      #Dämpfungen
#DGL-System
dgl1 = Eq(m1*x1.diff(t, 2) + d1*x1.diff(t, 1) + c1*x1 - c2*(x2-x1), 0) #1
dgl2 = Eq(m2*x2.diff(t, 2) + d2*x2.diff(t, 1) + c2*(x2-x1), 0)      #2
#Anfangswerte
aw = {
    x1.subs(t, 0): 0.1,
    x2.subs(t, 0): -0.1,
    x1.diff(t, 1).subs(t, 0): 0,
    x2.diff(t, 1).subs(t, 0): 0
}
#Lösung des DGL-Systems
gleichungen = [dgl1, dgl2]
aL = dsolve_system(gleichungen) #3
sL = dsolve_system(gleichungen, ics=aw) #4
gX1 = sL[0][0].rhs #5
gX2 = sL[0][1].rhs #6
#Ausgaben
# print("allgemeine Lösung\n", aL)
# print("spezielle Lösung")
# print("x1(t) =", N(gX1, 3))
# print("x2(t) =", N(gX2, 3))
p = plot(gX1, gX2, (t, 0, 10), show=False, legend=True)
p.xlabel = 't'
p.ylabel = 'Auslenkung'
p[0].line_color = 'blue'
p[0].label = 'x1'
p[1].line_color = 'red'
p[1].label = 'x2'
p.show()
```

In #1 und #2 stehen die Differenzialgleichungen. In #3 löst die SymPy-Methode `dsolve_system()` das DGL-System allgemein. Durch Angabe des Parameters `ics=aw` wird die spezielle Lösung berechnet (#4). In #5 und #6 werden die rechten Seiten der speziellen Lösungsgleichungen für die x_1 - und x_2 -Auslenkungen ermittelt.

Wenn Sie die Kommentare vor den `print`-Funktionen entfernen, können Sie sich die allgemeine und die spezielle Lösung ausgeben lassen.

Gekoppeltes Fadenpendel

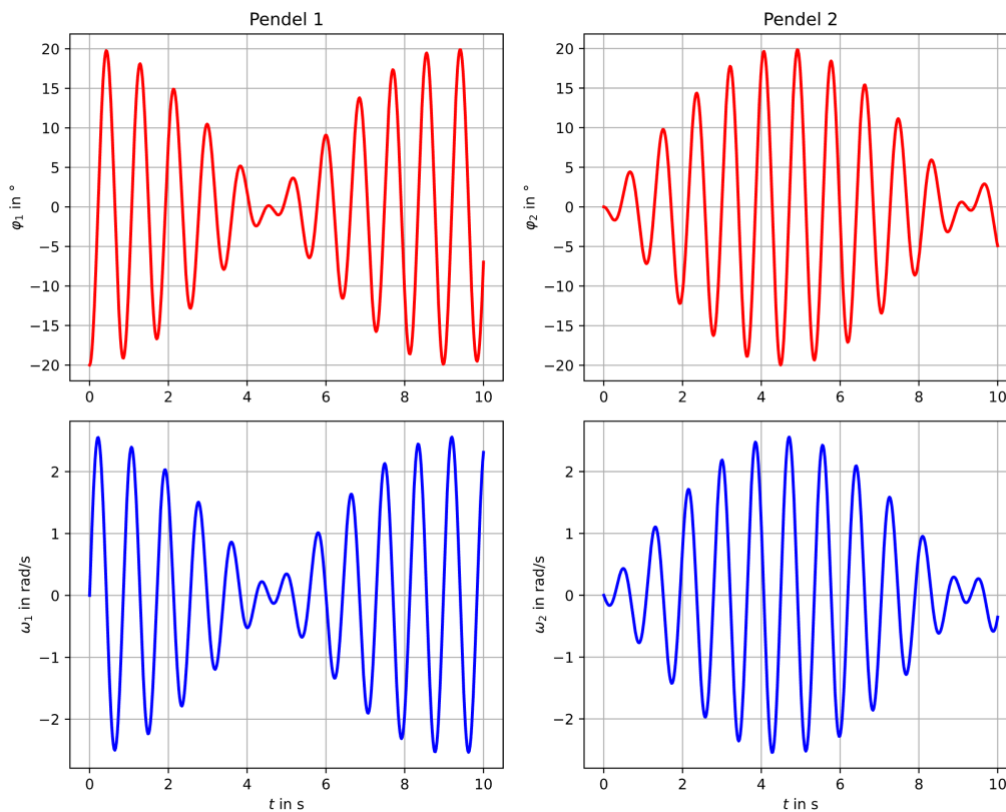
Für das gekoppelte Fadenpendel soll die Auslenkung φ und die Winkelgeschwindigkeit ω der Pendelbewegung simuliert werden.



Die Auslenkungen φ_1 und φ_2 sowie die Winkelgeschwindigkeiten ω_1 und ω_2 werden für kleine Auslenkungen durch das folgende DGL-System beschrieben:

$$\begin{aligned}\frac{d\varphi_1}{dt} &= \omega_1 \\ \frac{d\omega_1}{dt} &= -\frac{g}{l}\varphi_1 - \frac{c}{m}(\varphi_1 - \varphi_2) \\ \frac{d\varphi_2}{dt} &= \omega_2 \\ \frac{d\omega_2}{dt} &= -\frac{g}{l}\varphi_2 + \frac{c}{m}(\varphi_1 - \varphi_2)\end{aligned}$$

Die nachfolgenden Bilder zeigen die Funktionsplots der Lösungen des DGL-Systems.



Mit dem SciPy-Programm `sci_gekoppeltes_fadenpendel.py` werden die obigen Bilder erzeugt.

```
#sci_gekoppeltes_fadenpendel.py
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
#Pendeldataen
auslenkung1 = -20 #Auslenkung Pendel 1 in °
auslenkung2 = 0   #Auslenkung Pendel 2 in °
l = 0.2           #Pendellaengen in m
m = 0.2           #Pendelmassen in kg
c = 1             #Federkonstante N/m
g = 9.81          #Erdbeschleunigung in m/s^2
tmax=10
```

```

phi01,phi02=np.radians(auslenkung1),np.radians(auslenkung2)
#DGL-System, der Unterstrich bedeutet 1. Ableitung nach der Zeit
def dgl(t,phi0,l,m,c): #1
    phi1,w1,phi2,w2 = phi0
    phi1_dt=w1
    w1_dt=-g/l*phi1-c/m*(phi1-phi2)
    phi2_dt=w2
    w2_dt=-g/l*phi2+c/m*(phi1-phi2)
    return phi1_dt, w1_dt, phi2_dt, w2_dt
#Anfangswerte
omega01 = omega02 = 0
phi0 =[phi01,omega01,phi02,omega02]
#Loesung des DGL-Systems
t = np.linspace(0,tmax,500)
z=solve_ivp(dgl,[0,tmax],phi0,args=(l,m,c),dense_output=True) #2
phi1,w1,phi2,w2=z.sol(t) #3
#Grafikbereich
fig,axes=plt.subplots(2,2,figsize=(10,8))
#Auslenkung Pendel 1
axes[0,0].plot(t,np.degrees(phi1),"r-",lw=2)
#Auslenkung Pendel 2
axes[0,1].plot(t,np.degrees(phi2),"r-",lw=2)
#Winkelgeschwindigkeit Pendel 1
axes[1,0].plot(t,w1,'b-',lw=2)
#Winkelgeschwindigkeit Pendel 2
axes[1,1].plot(t,w2,"b-",lw=2)
#Beschriftungen
axes[0,0].set_title("Pendel 1")
axes[1,0].set_xlabel("$t$ in s")
axes[1,0].set_ylabel(r"$\omega_{1}$ in rad/s")
axes[0,0].set_ylabel(r"$\varphi_{1}$ in °",)
axes[0,1].set_title("Pendel 2")
axes[0,1].set_ylabel(r"$\varphi_{2}$ in °")
axes[1,1].set_ylabel(r"$\omega_{2}$ in rad/s")
axes[1,1].set_xlabel("$t$ in s")
for i,j in [[0,0],[0,1],[1,0],[1,1]]:axes[i,j].grid(True)
fig.tight_layout()
plt.show()

```

In #1 wird das DGL-System definiert, in #2 wird es gelöst und in #3 werden die Lösungen getrennt.

10 Mechanische Wellen

Für die Wellenzahl gilt:

$$k = \frac{2\pi}{\lambda}$$

Eine mechanische Welle bewegt sich mit der Geschwindigkeit

$$v = \frac{\omega}{k}$$

Für die Wellengleichung gilt:

$$y = A \cdot \sin(k \cdot x - \omega \cdot t)$$

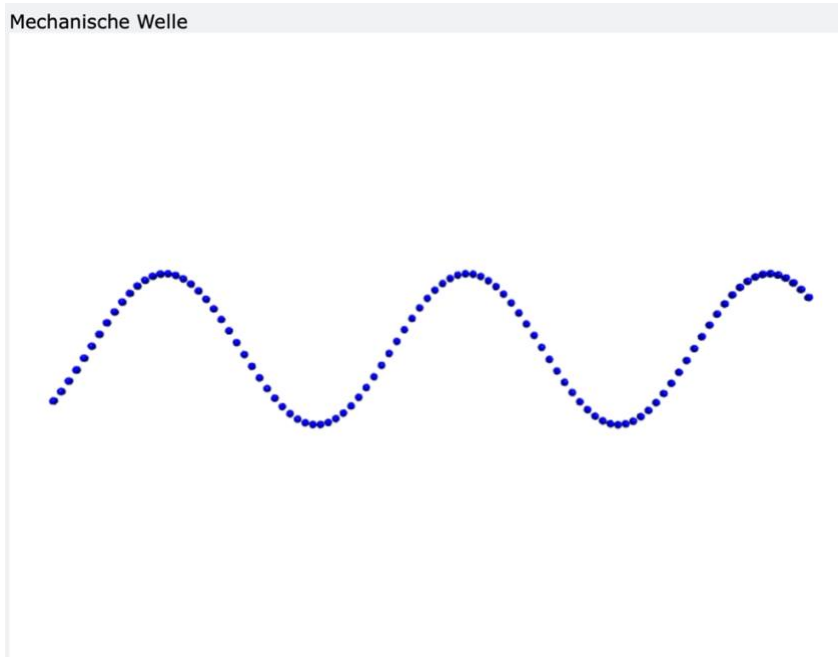
Das VPython-Programm *vp_welle_ki.py* wurde am 15.06.2025 mit **ChatGPT Deutsch** erzeugt. In die Eingabeaufforderung wurde der Text: "Schreibe ein VPython Programm, das eine mechanische Welle animiert" eingegeben. Es wurde nur die Hintergrundfarbe geändert. Alle Kommentare stammen von der KI.

```

#vp_welle_ki.py
from vpython import *
# Szene einrichten
scene = canvas(title='Mechanische Welle', width=800, height=600,
center=vector(0,0,0))
scene.background = color.white #hinzugefügt
# Parameter der Welle
amplitude = 1
wavelength = 4
k = 2*pi/wavelength
omega = 2*pi/2 # Periodendauer von 2 Sekunden
speed = omega/k
# Anzahl der Punkte
num_points = 100
x_start = -5
x_end = 5
dx = (x_end - x_start)/(num_points - 1)
# Erstelle die Punkte der Welle
points = [] #1
for i in range(num_points): #2
    x = x_start + i*dx #3
    y = amplitude*sin(k*x) #4
    point = sphere(pos=vector(x,y,0), radius=0.05, color=color.blue) #5
    points.append(point) #6
# Animationsschleife
t = 0
dt = 0.01
while True: #7
    rate(100)
    t += dt
    for i, point in enumerate(points): #8
        x = point.pos.x #9
        y = amplitude*sin(k*x - omega*t) #10
        point.pos.y = y #11

```

Das Programm erzeugt eine mechanische Wellenbewegung aus einzelnen Punkten. Die Welle bewegt sich von links nach rechts im Browserfenster. Der Screenshot zeigt eine Momentaufnahme der Wellenbewegung.



Analyse

Die Formeln der Wellengleichung wurden von der KI richtig generiert.

#1: Es wird die leere Liste `points` erzeugt.

#2: Die Laufvariable `i` nimmt Werte zwischen 0 und 99 an.

#3: Die Anweisung erzeugt die Werte für die x-Koordinatendaten. Es werden Werte zwischen -5 und +5 erzeugt. Die Schrittweite dx beträgt $(5 - (-5))/99 = 10/99 = 0,101010101010101$.

#4: Es werden die Werte für die y-Koordinatendaten der Welle berechnet: $y = \text{amplitude} * \sin(k * x)$.

#5: Die x-, y-Koordinatendaten werden dem Vektor `vector(x, y, 0)` übergeben und in das Objekt `points` gespeichert.

#6: Die einzelnen Punkte werden an das Listenende der Liste `points` angehängt. Es entsteht eine statische Sinusschwingung. Das kann man überprüfen, indem man die Animationsschleife auskommentiert und das Programm neu startet.

#7: Die Animation der Wellenbewegung wird innerhalb von zwei verschachtelten Schleifen durchgeführt. In der äußeren `while`-Schleife wird bei jedem Schleifendurchlauf die Zeit `t` um die Zeitschrittweite `dt` erhöht.

#8: In der inneren `for`-Schleife) durchlaufen die Laufvariablen `i` und `point` diejenigen Punktepaaire (`x|y`), die in #6 gespeichert wurden. Die eingebaute Python-Funktion `enumerate(points)` generiert aus der Liste `points` die x- und y- Werte der Wellengleichung.

Beispiel für `enumerate()`

```
liste=[0,1,4,9,16,25]
for x,y in enumerate(liste):
    print(x,y)
```

Ausgabe

```
0 0
1 1
2 4
3 9
4 16
5 25
```

#9: Die Punkte der Ausbreitung in x-Richtung werden dargestellt.

#10: Es werden die y-Koordinatendaten für die Auslenkungen berechnet. Das erste Argument der Sinusfunktion beschreibt die Verschiebung der Welle auf der x-Achse. Das zweite Argument bestimmt die Auslenkung zum Zeitpunkt `t`.

#11: Die Punkte der Auslenkung in y-Richtung werden dargestellt.