Einführung in die Programmierung für Studierende der Physik

JOHANN WOLFGANG GOETHE UNIVERSITÄT
23.06.2022

MATTHIAS HANAUSKE

FRANKFURT INSTITUTE FOR ADVANCED STUDIES
JOHANN WOLFGANG GOETHE UNIVERSITÄT
INSTITUT FÜR THEORETISCHE PHYSIK
ARBEITSGRUPPE RELATIVISTISCHE ASTROPHYSIK
D-60438 FRANKFURT AM MAIN
GERMANY

10. Vorlesung

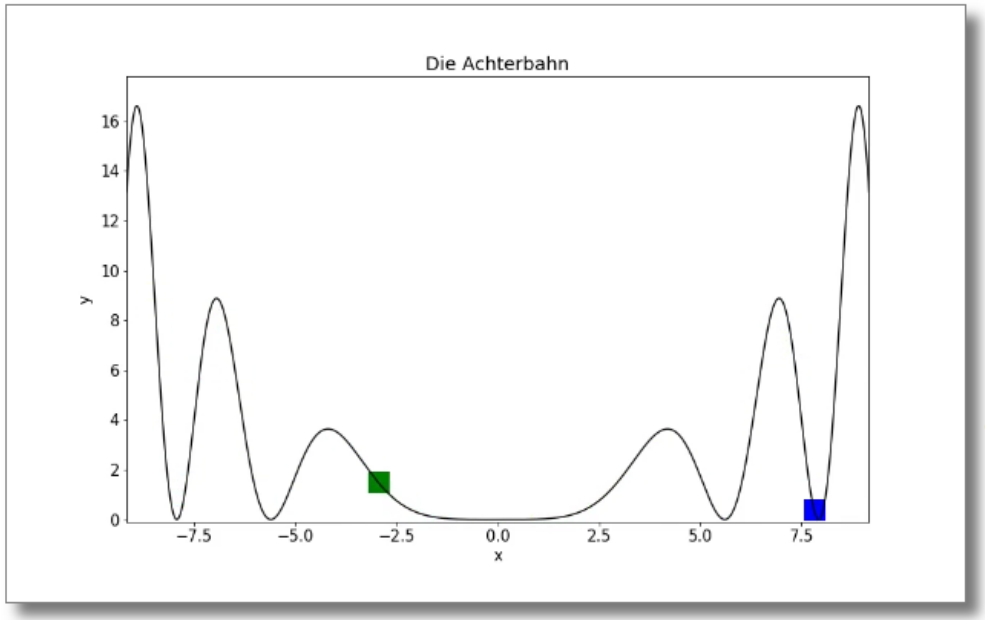
Plan für die heutige Vorlesung

- Beispiel Projekt: Die Achterbahn
- Abgeleitete Klassen, Vererbung von Klassenmerkmalen und Klassenhierarchien

Vorlesung 10

In dieser Vorlesung werden wir zunächst die ersten Teilaufgaben eines weiteren Projektes (das *Projekt Achterbahn*) exemplarisch besprechen. Es wird gezeigt, wie man die Bewegungsgleichungen des betrachteten Systems unter Verwendung der Euler-Lagrange Gleichungen mittels eines Jupyter Notebooks generiert. Das Umschreiben der DGL zweiter Ordnung in ein System von zwei DGLs erster Ordnung und die Implementierung und Lösung der Gleichungen mittels eines C++ Programmes wird ebenfalls beispielhaft vorgeführt. Die weiteren Teilaufgaben des Projektes wurden hingegen noch nicht bearbeitet, sodass Studierende auch dieses Projekt weiter bearbeiten können. Im zweiten Unterpunkt der Vorlesung werden wir auf die Vererbung von Klassenmerkmalen bei abgeleiteten Klassen und auf Klassenhierarchien eingehen.

Beispiel Projekt: Die Achterbahn



Das Projekt *Achterbahn* ist ein Anwendungsfall aus der klassischen Mechanik. Das Problem ist eine Verallgemeinerung des Problems "Bewegung eines Massenpunktes auf einer Zykloidenbahn" (siehe Aufgabe 15.7: Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel V15. Seite 271]). Das System besteht aus zwei Achterbahn-Wagen unterschiedlicher Massen $m_i\,,\ i \in [0,1]$, die reibungsfrei auf einer Metallschiene gleiten können. Die Achterbahn hat eine Länge von $l\,[m]$ und besitzt keine Kurven, sodass man den Verlauf der Metallschiene der Achterbahn als eine eindimensionale Funktion $h(x)\,,\ x \in [-l/2, l/2]$ beschreiben kann, wobei $h(x)$ die Höhe der Metallschiene der Achterbahn an der Position x in der Einheit Metern beschreibt. Die Herleitung der Bewegungsgleichungen erfolgt am elegantesten mittels der Euler-Lagrange Gleichungen, bzw. mittels der Hamilton Theorie. Die Bewegung eines Wagens wird durch eine Differentialgleichung zweiter Ordnung bestimmt und diese hängt nicht von dem Wert der Masse des Wagens ab. Um diese DGL zweiter Ordnung numerisch lösen zu können, schreibt man sie in ein System von zwei DGLs erster Ordnung um. Die nebenstehende Animation visualisiert die C++ Simulation zweier Wagen, wobei der

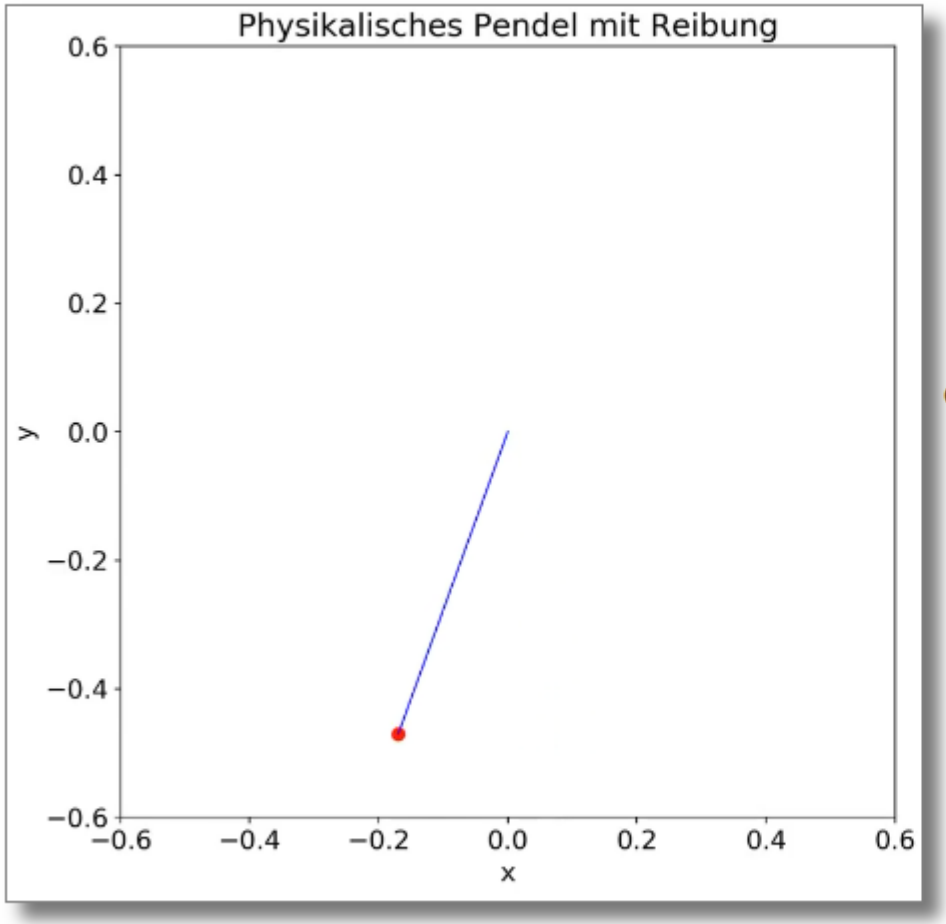
Vorlesung 10

Einige der im vorigen Kapitel vorgestellten studentischen Projekte benutzen die Programmiersprache Python zunächst zum Aufstellen der zugrundeliegenden Bewegungsgleichungen des betrachteten Systems. Die Herleitung der Bewegungsgleichungen eines physikalischen Systems mittels der Euler-Lagrange Gleichungen, ist ein eleganter mathematischer Trick, um auch komplizierte Systeme analytisch beschreiben zu können. Mittels eines Jupyter Notebooks unter Verwendung der SymPy Bibliothek ist eine solche Herleitung auch auf dem Computer gut zu implementieren. Im ersten Teil dieser Vorlesung wird ein zusätzliches studentisches Projekt besprochen (siehe Projekt: Die Achterbahn), das eine solche analytische Herleitung der Bewegungsgleichungen exemplarisch, beispielhaft zeigt. Die mittels des Python Jupyter Notebooks 'Achterbahn_2.ipynb' (View Notebook, Download Notebook) gefundene Bewegungsgleichung wird dann in ein System von erster Ordnung Differentialgleichungen umgeschrieben und mittels eines C++ Programms numerisch gelöst. Die simulierten Daten werden dann mittels Python in einer Animation visualisiert (siehe Video im linken Panel dieser Vorlesung). Es wird für die ersten Teilaufgaben des Projektes eine Musterlösung: Projekt *Achterbahn* vorgestellt und die weiteren Teilaufgaben können als studentisches Projekt bearbeitet werden.

Im zweiten Teil dieser Vorlesung betrachten wir weitere wichtige Sprachkonstrukte der Programmiersprache C++. C++ Klassen stehen häufig in Beziehung zueinander und die hierarchische Einteilung in Basisklasse (auch Oberklasse oder Superklasse) und abgeleitete Klasse (auch Subklasse) und das Konzept der Vererbung von Merkmalen der Basisklasse an die Subklasse, ist ein oft verwendetes und nützliches Konstrukt bei der Erstellung von umfangreichen C++ Programmen. Die Vererbung von Klassenmerkmalen kann in Form einer *Schnittstellenvererbung* oder/und *Implementierungsvererbung* geschehen. Die Konzepte der Vererbung werden zunächst an einfachen Beispielen erläutert und danach wird der Vererbungsmechanismus am Beispiel des Pendels erläutert.

Abgeleitete Klassen, Vererbung von Klassenmerkmalen und Klassenhierarchien

Abgeleitete Klassen, Vererbung von Klassenmerkmalen und Klassenhierarchien



Im zweiten Unterpunkt der Vorlesung werden wir auf die Vererbung von Klassenmerkmalen bei abgeleiteten Klassen eingehen. C++ Klassen stehen häufig in Beziehung zueinander. Man hat beispielsweise eine Oberklasse (z.B. Pendel), und aus dieser leitet sich eine andere Klasse (z.B. mathematisches Pendel, oder physikalisches Pendel) ab. Diese abgeleitete Klasse (auch Subklasse) erbt dann bestimmte Eigenschaften der Daten-Member und Member-Funktionen der Oberklasse 'Pendel' (auch Basisklasse oder Superklasse). Die Superklasse hat dabei oft einen übergeordneten, abstrakten Charakter und es können in ihr virtuelle Funktionen deklariert werden, die in den jeweiligen abgeleiteten Klassen dann definiert werden. In umfangreichen C++ Programmen baut sich somit eine Art von Klassenhierarchie auf und oft werden dabei auch sogenannte *Templates* eingesetzt, auf die wir erst in der nächsten Vorlesung eingehen werden. Die nebenstehende Abbildung zeigt die Simulation eines physikalischen Pendels mit Stokesschem Reibungsterm. Das zugrundeliegende C++ besitzt eine Superklasse 'Ding', die als eine Schnittstelle der kartesischen Orte und Geschwindigkeiten des Pendels fungiert. Von ihr wurde die Klasse 'Pendel' des *physikalischen Pendels* abgeleitet (Schnittstellenvererbung) und von dieser die Sub-Subklasse 'Pendel_math' des *mathematischen Pendels* (harmonischer Oszillator mit Reibung) abgeleitet (Implementierungsvererbung). Näheres siehe Abgeleitete Klassen, Vererbung von Klassenmerkmalen und Klassenhierarchien.

Beispiel Projekt: Die Achterbahn

Das Projekt *Achterbahn* ist ein Anwendungsfall aus der klassischen Mechanik. Das System besteht aus zwei Achterbahn-Wagen unterschiedlicher Massen $m_i\,,\ i \in [0,1]$, die reibungsfrei auf einer Metallschiene gleiten können. Die Achterbahn hat eine Länge von $l\,[m]$ und besitzt keine Kurven, sodass man den Verlauf der Metallschiene der Achterbahn als eine eindimensionale Funktion $h(x)\,,\ x \in [-l/2, l/2]$ beschreiben kann, wobei $h(x)$ die Höhe der Metallschiene der Achterbahn an der Position x in der Einheit Metern beschreibt. Der Einstiegspunkt in die Wagen befindet sich bei $x = 0$, $y = 0$ und die Achterbahn ist so konstruiert, dass sie die zwei Wagen zu unterschiedlichen Anfangszeitpunkten τ_i in die Achterbahnschiene katapultiert, wobei die Anfangsgeschwindigkeiten $\vec{v}_i(\tau_i)$ der Wagen (gemessen mit einem Sensor bei $x = 10\,[cm]$) sich dabei unterscheiden. Jeder der Achterbahn-Wagen besitzt am vorderen und hinterem Bereich eine starke Metallfeder, die zum Einsatz kommt, wenn sich zwei der Wagen treffen. Es wird dabei angenommen, dass es sich bei einem Zusammenprall der Wagen um einen elastischen Stoß zweier Körper handelt.
Betrachten wir uns zunächst den Fall mit nur einem Wagen (Koordinaten: $\vec{r} = (x(t),\, y(t)) = (x(t),\, h(x(t)))$, Masse m). Die Herleitung der Bewegungsgleichungen erfolgt am elegantesten mittels der Euler-Lagrange Gleichungen, bzw. mittels der Hamilton Theorie. Die Lagrangefunktion des Systems lautet:

$$L = T - V \text{mit:}$$

$$T = \frac{1}{2} m \left[\dot{x}(t)^2 + \dot{y}(t)^2\right] \quad , \quad V(t) = m\,g\,y(t)\ ,$$

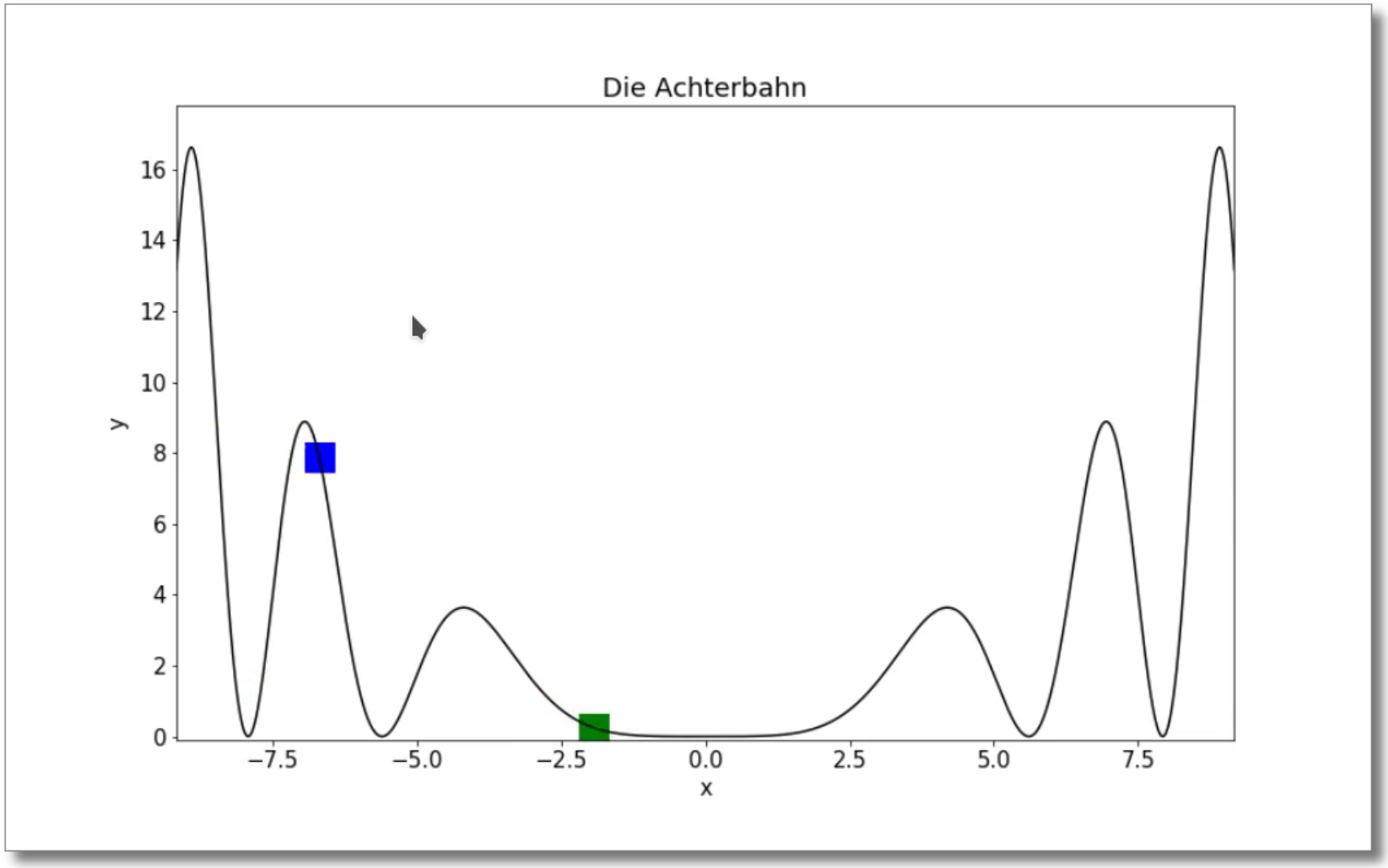
$$\text{mit:} \quad \dot{y}(t) = \frac{dy}{dt} = \frac{dh(x(t))}{dt} = \frac{dx(t)}{dt} \cdot \frac{dh(x)}{dx} = \dot{x}(t) \cdot \frac{dh(x)}{dx} \quad ,$$

wobei g der Wert der Erdbeschleunigung ist. Nun kann man für eine beliebige Form des Verlaufs der Achterbahn (beschrieben durch die Funktion $h(x)$) die Bewegungsgleichung des Wagens mittels der Lagrange-Gleichungen

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0 \quad \text{mit:}$$

$$L = \frac{1}{2} m \left[\dot{x}(t)^2 + \left(\frac{dh(x(t))}{dt}\right)^2\right] - m\,g\,h(t) = \frac{1}{2} m \left[\dot{x}^2 + \left(\dot{x} \cdot \frac{dh(x)}{dx}\right)^2\right] - m\,g\,h(x) = \frac{1}{2} m\,\dot{x}^2 \cdot \left[1 + \left(\frac{dh(x)}{dx}\right)^2\right] - m\,g\,h(x)$$

herleiten.



Mögliche Teilaufgaben des Projektes

Berechnen Sie zunächst auf analytischem Weg, mittels eines Jupyter Notebooks unter Verwendung der SymPy Bibliothek, die Euler-Lagrange Gleichungen der Bewegung eines Wagens auf der Achterbahn, wobei der Verlauf der Metallschiene der Achterbahn durch die Funktion $h(x) = e^{\frac{\sqrt{x^2}}{10}} \cdot \left(\sin\left(\frac{x^2}{10}\right)\right)^2$ bestimmt ist. Schreiben Sie dann die Bewegungsgleichung in ein System von erster Ordnung Differentialgleichungen um.

Der Wagen ($m = 100$ [Kg]) werde zur Zeit $t = 0$ [s], mit einer Anfangsgeschwindigkeit von $\dot{x}(0) = 30.42$ [km/h] bei $x = 0.01$ [m] in die Achterbahn katapultiert. Lösen Sie das System von Differentialgleichungen erster Ordnung zunächst mittels Python und visualisieren Sie die Simulation in einem Jupyter Notebook. Erhöhen Sie nun die Anfangsgeschwindigkeit auf $\dot{x}(0) = 60$ [km/h] und stellen beide numerischen Lösungen in einer Animation dar.

Lösen Sie nun das System von Differentialgleichungen erster Ordnung mittels eines C++ Programms und benutzen Sie das Runge-Kutta Ordnung vier Verfahren.

Vergleichen Sie beide Simulationen quantitativ.

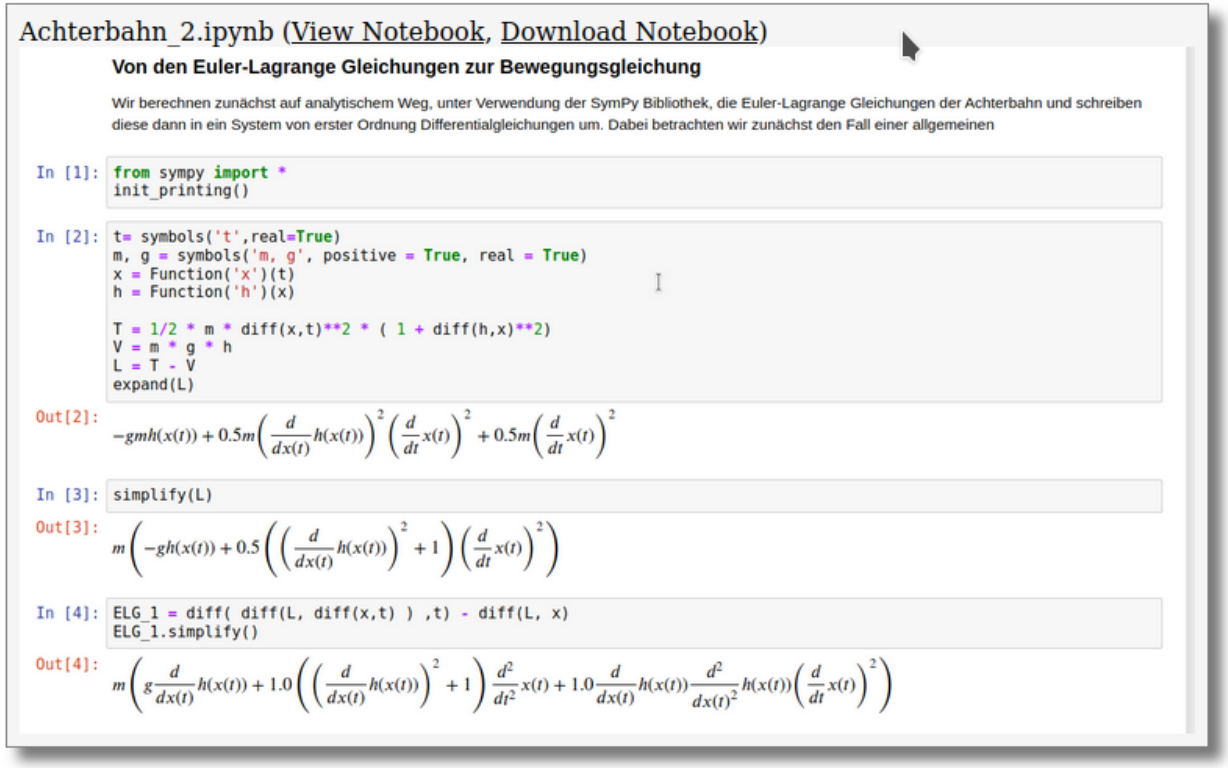
Verallgemeinern Sie das C++ Programm so, dass mehrere Wagen auf der Achterbahn fahren können. Simulieren Sie ein System bestehend aus zwei wechselwirkungsfreien Wagen (siehe obige Animation) und benutzen Sie dabei für den ersten Wagen $i = 0$, $\tau_0 = 0$ und $\dot{x}_0(0) = 16.45\,[m/s]$ (blauer Wagen) und für den zweiten Wagen den ersten Wagen $i = 1$, $\tau_1 = 2\,[s]$ und $\dot{x}_1(2) = 5.5\,[m/s]$ (grüner Wagen).

Um die Richtigkeit der berechneten Daten des C++ Programmes zu verifizieren, führen Sie (nur in dieser Teilaufgabe) eine separate Simulation mit abgeänderter Achterbahnform durch, bei der es eine analytische Lösung gibt. Berechnen Sie zunächst auf analytischem Weg, mittels eines Jupyter Notebooks unter Verwendung der SymPy Bibliothek, die Euler-Lagrange Gleichungen der Bewegung eines Massenpunktes der Masse m auf einer Zykloidenbahn (siehe Aufgabe 15.7: Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel V15, Seite 271]). Der Massepunkt gleitet reibungsfrei auf einer Zykloide, die durch

Musterlösung: Projekt Achterbahn

1. Teilaufgabe

Berechnen Sie zunächst auf analytischem Weg, mittels eines Jupyter Notebooks unter Verwendung der SymPy Bibliothek, die Euler-Lagrange Gleichungen der Bewegung eines Wagens auf der Achterbahn, wobei der Verlauf der Metallschiene der Achterbahn durch die Funktion $h(x) = e^{\frac{\sqrt{x^2}}{10}} \cdot \left(\sin\left(\frac{x^2}{10}\right)\right)^2$ bestimmt ist. Schreiben Sie dann die Bewegungsgleichung in ein System von erster Ordnung Differentialgleichungen um.



Achterbahn_2.ipynb (View Notebook, Download Notebook)

Von den Euler-Lagrange Gleichungen zur Bewegungsgleichung

Wir berechnen zunächst auf analytischem Weg, unter Verwendung der SymPy Bibliothek, die Euler-Lagrange Gleichungen der Achterbahn und schreiben diese dann in ein System von erster Ordnung Differentialgleichungen um. Dabei betrachten wir zunächst den Fall einer allgemeinen

In [1]:
```
from sympy import *
init_printing()
```

In [2]:
```
t= symbols('t',real=True)
m, g = symbols('m, g', positive = True, real = True)
x = Function('x')(t)
h = Function('h')(x)

T = 1/2 * m * diff(x,t)**2 * ( 1 + diff(h,x)**2)
V = m * g * h
L = T - V
expand(L)
```

Out[2]:
$$-gmh(x(t)) + 0.5m\left(\frac{d}{dx(t)}h(x(t))\right)^2\left(\frac{d}{dt}x(t)\right)^2 + 0.5m\left(\frac{d}{dt}x(t)\right)^2$$

In [3]:
```
simplify(L)
```

Out[3]:
$$m\left(-gh(x(t)) + 0.5\left(\left(\frac{d}{dx(t)}h(x(t))\right)^2 + 1\right)\left(\frac{d}{dt}x(t)\right)^2\right)$$

In [4]:
```
ELG_1 = diff( diff(L, diff(x,t) ) ,t) - diff(L, x)
ELG_1.simplify()
```

Out[4]:
$$m\left(g\frac{d}{dx(t)}h(x(t)) + 1.0\left(\left(\frac{d}{dx(t)}h(x(t))\right)^2 + 1\right)\frac{d^2}{dt^2}x(t) + 1.0\frac{d}{dx(t)}h(x(t))\frac{d^2}{dx(t)^2}h(x(t))\left(\frac{d}{dt}x(t)\right)^2\right)$$

Das nebenstehende Python Jupyter Notebook 'Achterbahn_2.ipynb' (View Notebook, Download Notebook) stellt eine Musterlösung der ersten Teilaufgaben des Projektes Die Achterbahn dar.

Wir berechnen zunächst auf analytischem Weg, unter Verwendung der SymPy Bibliothek, die Euler-Lagrange Gleichungen der Achterbahn und schreiben diese dann in ein System von erster Ordnung Differentialgleichungen um. Dabei betrachten wir zunächst den Fall einer allgemeinen Funktion $h(x)$, damit eine, in späteren Teilaufgaben möglicherweise abgeänderte Form der Achterbahn, einfach in den Algorithmus eingebaut werden kann. Wir definieren zunächst die nötigen SymPy-Symbole und SymPy-Funktionen und implementieren die Lagrangefunktion $L = T - V$ in der folgenden Form:

$$T = \frac{1}{2}m\left[\dot{x}(t)^2 + \dot{y}(t)^2\right] = \frac{1}{2}m\,\dot{x}^2 \cdot \left[1 + \left(\frac{dh(x)}{dx}\right)^2\right]$$

$$V(t) = m\,g \cdot y(t) = m\,g \cdot h(x(t))$$

Mittels der Euler-Lagrange Gleichungen

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0$$

$$m\left(g\frac{d}{dx(t)}h(x(t))+\left(\left(\frac{d}{dx(t)}h(x(t))\right)^2+1\right)\frac{d^2}{dt^2}x(t)+\frac{d}{dx(t)}h(x(t))\frac{d^2}{dx(t)^2}h(x(t))\left(\frac{d}{dt}x(t)\right)^2\right)=0 \quad .$$

Die Bewegung des Wagens wird somit durch eine Differentialgleichung zweiter Ordnung bestimmt und diese hängt nicht von dem Wert der Masse des Wagens ab. Wir formen die DGL in die übliche Form einer Differentialgleichung zweiter Ordnung um:

$$\ddot{x}(t)=\frac{d^2}{dt^2}x(t)=-\frac{\left(g+\frac{d^2}{dx^2}h(x)\left(\frac{d}{dt}x(t)\right)^2\right)\frac{d}{dx}h(x)}{\left(\frac{d}{dx}h(x)\right)^2+1}=f(t,x,\dot{x})$$

Setzt man nun die formbestimmende Funktion $h(x)=e^{\frac{\sqrt{x^2}}{10}}\cdot\left(\sin\left(\frac{x^2}{10}\right)\right)^2$ ein, so erhält man mittels Sympy den in der folgenden Abbildung dargestellte Ausdruck für die DGL bestimmende Funktion $f(t,x,\dot{x})$.

In [15]:
```
f=solve(ELG, diff(x,t,t))[0]
f
```

Out[15]:

$$\frac{0.2\left(-158.113883008419g\sqrt{x^2(t)}e^{0.316227766016838\sqrt{x^2(t)}}\sin\left(0.1x^2(t)\right)-200.0gx^2(t)e^{0.316227766016838\sqrt{x^2(t)}}\cos\left(0.1x^2(t)\right)\right.}{}$$

$$+75.8946638440411\sqrt{x^2(t)}x^2(t)e^{0.632455532033676\sqrt{x^2(t)}}\sin^3\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2-63.2455532033676\sqrt{x^2(t)}x^2(t)e^{0.632455532033676\sqrt{x^2(t)}}\sin$$

$$\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2-15.8113883008419\sqrt{x^2(t)}e^{0.632455532033676\sqrt{x^2(t)}}\sin^3\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2$$

$$+63.2455532033676\sqrt{x^2(t)}e^{0.632455532033676\sqrt{x^2(t)}}\cos^3\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2-63.2455532033676\sqrt{x^2(t)}e^{0.632455532033676\sqrt{x^2(t)}}\cos$$

$$\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2-32.0x^4(t)e^{0.632455532033676\sqrt{x^2(t)}}\cos^3\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2+16.0x^4(t)e^{0.632455532033676\sqrt{x^2(t)}}\cos\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2$$

$$+80.0x^2(t)e^{0.632455532033676\sqrt{x^2(t)}}\sin^3\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2-80.0x^2(t)e^{0.632455532033676\sqrt{x^2(t)}}\sin\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2$$

$$\left.+60.0x^2(t)e^{0.632455532033676\sqrt{x^2(t)}}\cos^3\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2-60.0x^2(t)e^{0.632455532033676\sqrt{x^2(t)}}\cos\left(0.1x^2(t)\right)\left(\frac{d}{dt}x(t)\right)^2\right)x(t)\sin\left(0.1x^2(t)\right)$$

$$\overline{\left(3.16227766016838\left(x^2(t)\right)^{0.5}\sin\left(0.1x^2(t)\right)+4.0x^2(t)\cos\left(0.1x^2(t)\right)\right)^2e^{0.632455532033676\sqrt{x^2(t)}}\sin^2\left(0.1x^2(t)\right)+100.0x^2(t)}$$

Numerisches Lösen des Systems von Differentialgleichungen

Um die entstandene DGL zweiter Ordnung numerisch lösen zu können, schreiben wir sie in ein System von zwei DGLs erster Ordnung um. Wir betrachten nun die numerische Lösung und machen die folgenden Variablenumbenennung: $u_1(t) = x(t)$, und $u_2(t) = \dot{x}(t)$ und definieren das System von DGLs wie folgt:

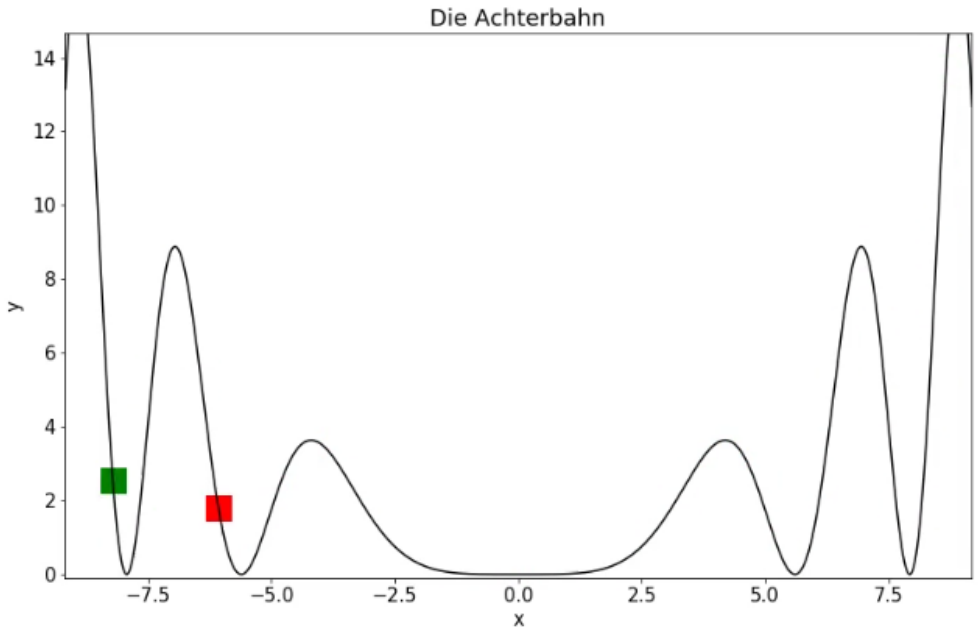
$$\dot{u_1}(t) = \frac{du_1}{dt} = \frac{dx}{dt} = u_2(t)$$

$$\dot{u_2}(t) = \frac{du_2}{dt} = \frac{d\dot{x}}{dt} = \ddot{x}(t) = f(u_1, u_2) = f(t, x, \dot{x})$$

2. Teilaufgabe

Der Wagen ($m = 100$ [Kg]) werde zur Zeit $t = 0$ [s], mit einer Anfangsgeschwindigkeit von $\dot{x}(0) = 30.42$ [km/h] bei $x = 0.01$ [m] in die Achterbahn katapultiert. Lösen Sie das System von Differentialgleichungen erster Ordnung zunächst mittels Python und visualisieren Sie die Simulation in einem Jupyter Notebook. Erhöhen Sie nun die Anfangsgeschwindigkeit auf $\dot{x}(0) = 60$ [km/h] und stellen beide numerischen Lösungen in einer Animation dar.

Achterbahn_2.ipynb (View Notebook, Download Notebook)



Die nebenstehende Animation stellt die Bewegung der beiden Wagen dar und wurde mit dem Python Jupyter Notebook 'Achterbahn_2.ipynb' (View Notebook, Download Notebook) erstellt. Der rote Wagen wurde mit $\dot{x}(0) = 30.42$ [km/h] in die Bahn katapultiert und der grüne Wagen mit $\dot{x}(0) = 60$ [km/h].

3. Teilaufgabe

Lösen Sie nun das System von Differentialgleichungen erster Ordnung mittels eines C++ Programms und benutzen Sie das Runge-Kutta Ordnung vier Verfahren.

Als Vorlage verwenden wir das in der vorigen Vorlesung vorgestellte C++ Programm DGL_2_a.cpp, das eine Differentialgleichung zweiter Ordnung mittels der Euler- und Runge-Kutta Ordnung vier Methode löst (siehe unteres Frame). Die für unser Problem keine analytische Lösung existiert, benötigen wir die Funktionen 'double u_1_analytisch(...)' und 'double u_2_analytisch(...)' nicht mehr. Die DGL bestimmende Funktion $f(t, x, \dot{x})$ exportieren wir uns als C++ Quelltext vom Jupyter Notebook und fügen den Ausdruck in der Funktion 'double f_2(...)'

Benutzte Vorlage

Vorlage: DGL_2_a.cpp

```
/* Berechnung der Lösung einer Differentialgleichung zweiter Ordnung ( y''= 2*y' - 2*y + exp(2*t)*sin(t) )
 * welches in ein System von zwei Differentialgleichung umgeschrieben wurde
 * Form des DGL Systems: u_1'=y'=f_1(t,u_1,u_2)=u_2 , u_2'=y''=f_2(t,u_1,u_2)
 * Loesung mittels der Euler Methode und Runge-Kutta Ordnung vier Methode
 * Zeitentwicklung von u_1(t), u_2(t) fuer unterschiedliche t-Werte in [a,b]
 *  Ausgabe zum Plotten mittels Python Jupyter Notebook DGL_2.ipynb: "./a.out >> DGL_2_a.dat
 */
#include <stdio.h>            // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>              // Bibliothek für mathematisches (e-Funktion, Betrag, ...)

double f_1(double t, double u_1, double u_2){                          // Deklaration und Definition der Funktion f_1(t,u_1,u_2)
    double wert;
    wert = u_2;                                                        // Eigentliche Definition der Funktion
    return wert;                                                       // Rueckgabewert der Funktion f_1
}                                                                      // Ende der Funktion f_1

double f_2(double t, double u_1, double u_2){                          // Deklaration und Definition der Funktion f_2(t,u_1,u_2)
    double wert;
    wert = 2*u_2 - 2*u_1 + exp(2*t)*sin(t);                            // Eigentliche Definition der Funktion
    return wert;                                                       // Rueckgabewert der Funktion f_2
}                                                                      // Ende der Funktion f_2

double u_1_analytisch(double t, double alpha_1, double alpha_2){       // Analytische Loesung y(t)=u_1(t)
    double wert;                                                       // bei gegebenem Anfangswert y(a)=1, y'(a)=1
    wert = exp(t)*sin(t)*(alpha_2 + 1.0/5.0 - alpha_1) + exp(t)*cos(t)*(alpha_1 + 2.0/5.0) + ((sin(t) - 2*cos(t))*exp(2*t))/5.0;
    return wert;                                                       // Rueckgabewert
}                                                                      // Ende der Definition

double u_2_analytisch(double t, double alpha_1, double alpha_2){       // Analytische Loesung y'(t)=u_2(t)
    double wert;                                                       // bei gegebenem Anfangswert y(a)=1, y'(a)=1
    wert = ((-3*cos(t) + 4*sin(t))*exp(2*t))/5.0 - 2*((alpha_1 - alpha_2/2.0 + 1.0/10.0)*sin(t) - cos(t)*(alpha_2 + 3.0/5.0)/2.0)*exp(t);
    return wert;                                                       // Rueckgabewert
}                                                                      // Ende der Definition

int main(){                                                            // Hauptfunktion
    double a = 0;                                                      // Untergrenze des Zeit-Intervalls [a,b] in dem die Loesung berechnet werden soll
```

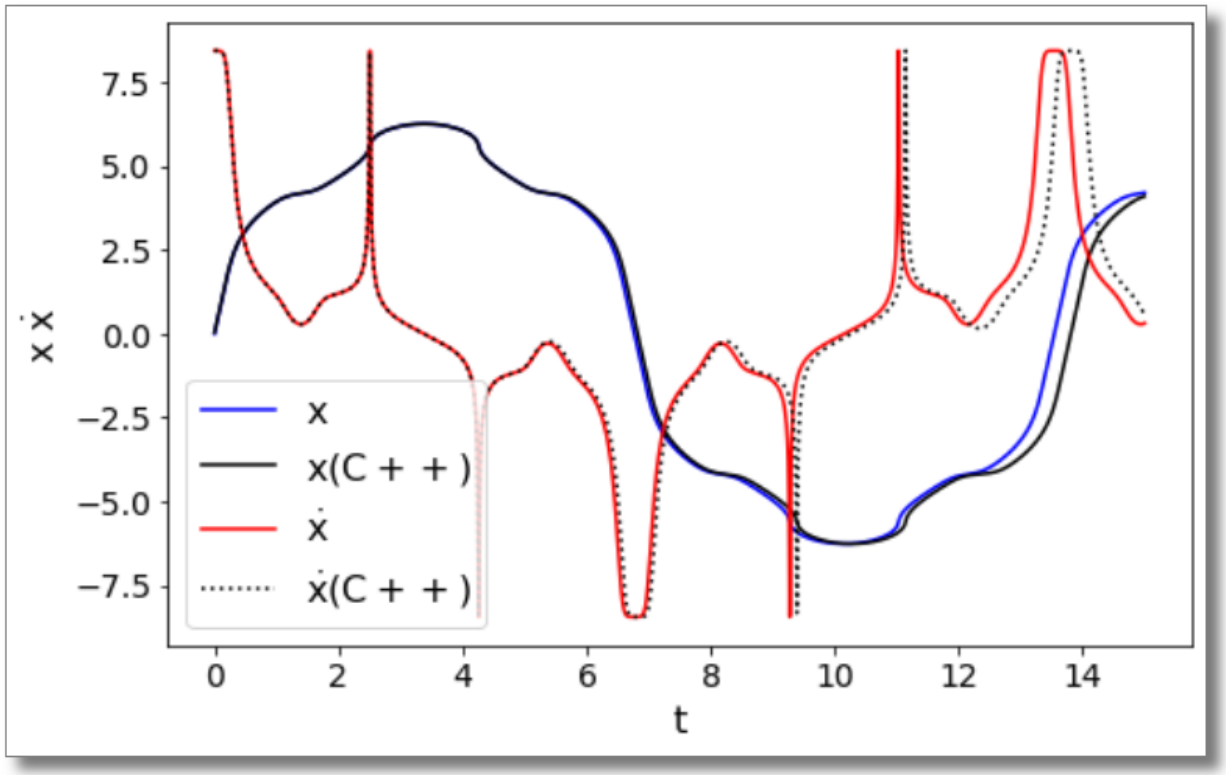
Achterbahn_1.hpp

```
/* Klasse zur Berechnung der Loesung einer Differentialgleichung der Form y''=f_2(t,y)
 * mittels Runge-Kutta Ordnung vier Verfahren
 * Verfahren zur Loesung der DGL ist in eine Klasse ausgelagert
 * Zeitentwicklung der fuer unterschiedliche t-Werte in [a,b]
 * Kostruktor: Achterbahn_1(Anfangszeit a, Endzeit b, Anzahl der Punkte N, Anfangswert1 alpha_1=y(a)=u_1(a), Anfangswert2 alpha_2=y'(a)=u_2(a),)
 */
#include <stdio.h>              // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>                // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
#include <vector>               // Vector-Container der Standardbibliothek
using namespace std;            // Benutze den Namensraum std

class Achterbahn_1{             //Definition der Klasse 'Achterbahn_1'
    double a = 0;               // Untergrenze des Zeit-Intervalls [a,b] in dem die Loesung berechnet werden soll
    double b = 1;               // Obergrenze des Intervalls [a,b]
    const unsigned int N = 5000; // Anzahl der Punkte in die das t-Intervall aufgeteilt wird
    double h = (b - a)/N;       // Abstand dt zwischen den aequidistanten Punkten des t-Intervalls (h=dt)
    double alpha_1 = 0.0;       // 1.Anfangswert bei t=a: u_1(a)=alpha_1
    double alpha_2 = 0.0;       // 2.Anfangswert bei t=a: u_2(a)=alpha_2
    double t = a;               // Aktueller Zeitwert
    double k1_1,k2_1,k3_1,k4_1; // Deklaration der vier Runge-Kutta Parameter fuer u_1
    double k1_2,k2_2,k3_2,k4_2; // Deklaration der vier Runge-Kutta Parameter fuer u_2

    vector<double> Zeit;        // Deklaration eines double Vektors zum Speichern der Zeit-Werte
    vector<double> u_RungeK_4_1; // Deklaration eines double Vektors zum Speichern der Loesung fuer u_1
    vector<double> u_RungeK_4_2; // Deklaration eines double Vektors zum Speichern der Loesung fuer u_2


    public:
        // Konstruktor mit fuenf Argumenten (Initialisierung der Parameter, Berechnung der Loesung der DGL)
        Achterbahn_1(double a_, double b_, const unsigned int N_, double alpha_1_, double alpha_2_) : a(a_),b(b_),N(N_),alpha_1(alpha_1_),alpha_2(alpha_2_) {
            Zeit.resize( N + 1 );                           // Die Anzahl der Eintraege im Vektor wird auf N+1 erhoeht
            u_RungeK_4_1.resize( N + 1 );                   // Die Anzahl der Eintraege im Vektor wird auf N+1 erhoeht
            u_RungeK_4_2.resize( N + 1 );                   // Die Anzahl der Eintraege im Vektor wird auf N+1 erhoeht

            Zeit[ N ] = b;                                  // Zum Zeit-Vektor die Endzeit eintragen
            u_RungeK_4_1[0] = alpha_1;                      // Zum y-Vektor den Anfangswert alpha_1=y(a) eintragen
            u_RungeK_4_2[0] = alpha_2;                      // Zum y'-Vektor den Anfangswert alpha_2=y'(a) eintragen

            for(int i=0; i < N; ++i){                       // for-Schleife ueber die einzelnen Punkte des t-Intervalls
                t = a + i*h;                                // Zeit-Parameter wird um h erhoeht
                Zeit[i] = t;                                // Zum Zeit-Vektor die neue Zeit eintragen

                k1_1 = h*f_1(t,u_RungeK_4_1[i],u_RungeK_4_2[i]);                    // Runge-Kutta Parameter k1 fuer u_1
                k1_2 = h*f_2(t,u_RungeK_4_1[i],u_RungeK_4_2[i]);                    // Runge-Kutta Parameter k1 fuer u_2
                k2_1 = h*f_1(t+h/2,u_RungeK_4_1[i]+k1_1/2,u_RungeK_4_2[i]+k1_2/2);  // Runge-Kutta Parameter k2 fuer u_1
                k2_2 = h*f_2(t+h/2,u_RungeK_4_1[i]+k1_1/2,u_RungeK_4_2[i]+k1_2/2);  // Runge-Kutta Parameter k2 fuer u_2
                k3_1 = h*f_1(t+h/2,u_RungeK_4_1[i]+k2_1/2,u_RungeK_4_2[i]+k2_2/2);  // Runge-Kutta Parameter k3 fuer u_1
                k3_2 = h*f_2(t+h/2,u_RungeK_4_1[i]+k2_1/2,u_RungeK_4_2[i]+k2_2/2);  // Runge-Kutta Parameter k3 fuer u_2
                k4_1 = h*f_1(t+h,u_RungeK_4_1[i]+k3_1,u_RungeK_4_2[i]+k3_2);        // Runge-Kutta Parameter k4 fuer u_1
                k4_2 = h*f_2(t+h,u_RungeK_4_1[i]+k3_1,u_RungeK_4_2[i]+k3_2);        // Runge-Kutta Parameter k4 fuer u_2

                u_RungeK_4_1[i+1] = u_RungeK_4_1[i] + (k1_1 + 2*k2_1 + 2*k3_1 + k4_1)/6; // Zum y-Vektor den neuen Wert eintragen
                u_RungeK_4_2[i+1] = u_RungeK_4_2[i] + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6; // Zum y'-Vektor den neuen Wert eintragen
            }                                                                   // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls
        };                                                                      // Ende des Konstruktors

        double f_1(double t, double u_1, double u_2);                  // Deklaration der Member-Funktion f_1(t,y) (Definition findet ausserhalb der Klasse statt)
        double f_2(double t, double u_1, double u_2);                  // Deklaration der Member-Funktion f_2(t,y) (Definition findet ausserhalb der Klasse statt)

        const vector<double>& get_u_1() const { return u_RungeK_4_1; } // Definition der konstanten Member-Funktion get_u_1, Rueckgabewert vector der Loesung y(t)=u_1(t) der DGL
        const vector<double>& get_u_2() const { return u_RungeK_4_2; } // Definition der konstanten Member-Funktion get_u_2, Rueckgabewert vector der Loesung y'(t)=u_2(t) der DGL
        const vector<double>& get_zeit() const { return Zeit; }        // Definition der konstanten Member-Funktion get_zeit(), Rueckgabewert vector der zeit-Punkte
};                                                                     // Ende der Klasse
```

Achterbahn_1.cpp

```cpp
// Musterlösung Projekt Achterbahn (Aufgabe 1)
/* Berechnung der Lösung einer Differentialgleichung der Form x'=f(t,x)
 * mittels Runge-Kutta Ordnung vier Verfahren
 * Verfahren zur Loesung der DGL ist in eine Klasse ausgelagert
 * Zeitentwicklung der fuer unterschiedliche t-Werte in [a,b]
 */

#include "Achterbahn_1.hpp"    // Einbinden der Klasse die das System der DGLs mittels Runge-Kutta Ordnung vier loest
#include <stdio.h>             // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>               // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
#include <vector>              // Vector-Container der Standardbibliothek
using namespace std;           // Benutze den Namensraum std

double Achterbahn_1::f_1(double t, double u_1, double u_2){          // Deklaration und Definition der Funktion f_1(t,u_1,u_2)
    double wert;
    wert = u_2;                                                      // Eigentliche Definition der Funktion
    return wert;                                                     // Rueckgabewert der Funktion f_1
}                                                                    // Ende der Funktion f_1

double Achterbahn_1::f_2(double t, double u_1, double u_2){          // Deklaration und Definition der Funktion f_2(t,u_1,u_2), berechnet mittels des Jupyter Notebooks Achterbahn.ipynb
    double wert;
    wert = 0.20000000000000001*u_1*(-32.0*pow(u_1, 4)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(cos(0.10000000000000001*pow(u_1, 2)), 3) +
    16.0*pow(u_1, 4)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2)) +
    75.894663844041105*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 3)*fabs(u_1) +
    80.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 3) -
    63.245553203367592*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2))*fabs(u_1) -
    80.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2)) +
    60.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(cos(0.10000000000000001*pow(u_1, 2)), 3) -
    60.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2)) -
    1962.0*pow(u_1, 2)*exp(0.316227766016838*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2)) -
    15.811388300841898*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 3)*fabs(u_1) +
    63.245553203367592*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(cos(0.10000000000000001*pow(u_1, 2)), 3)*fabs(u_1) -
    63.245553203367592*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2))*fabs(u_1) -
    1551.0971923125903*exp(0.316227766016838*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2))*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2))/(100.0*pow(u_1, 2) +
    pow(4.0*pow(u_1, 2)*cos(0.10000000000000001*pow(u_1, 2)) +
    3.1622776601683795*sin(0.10000000000000001*pow(u_1, 2))*pow(fabs(u_1), 1.0), 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 2));
    return wert;                                                     // Rueckgabewert der Funktion f_2
}                                                                    // Ende der Funktion f_2

int main(){                                                          // Hauptfunktion
    Achterbahn_1 Wagen_0 { 0, 15, 5000, 0.01, 30.42*1000/(60*60) };  // Konstruktor erzeugt die zeitliche Entwicklung eines Wagens auf der Achterbahn
    vector<double> Zeit = Wagen_0.get_zeit();                        // Aufrufen der Member-Funktion get_zeit() und kopieren der Zeitwerte
    vector<double> x = Wagen_0.get_u_1();                            // Aufrufen der Member-Funktion et_u_1() und kopieren der Loesungswerte
    vector<double> v_x = Wagen_0.get_u_2();                          // Aufrufen der Member-Funktion et_u_2() und kopieren der Loesungswerte

    printf("# 0: Index i \n# 1: t-Wert \n# 2:x(t) \n");              // Beschreibung der ausgegebenen Groessen
    printf("# 3: v_x(t) \n");                                        // (i, t, x, v_x)

    for(int i=0; i  <Zeit.size(); ++i){                              // for-Schleife zur Terminalausgabe der Loesung
        printf("%4d %19.15f %19.15f %19.15f \n",i, Zeit[i], x[i], v_x[i]);
    }
}                                                                    // Ende der Hauptfunktion
```

Numerische Lösung mit Python



In einer veralteten Version der Musterlösung dieser Teilaufgabe (siehe C++ Programm Achterbahn_1_old.cpp und hpp-Datei dsolve_Sys_1.hpp) wurde die ausgelagerte Klasse noch 'dsolve_Sys_1' genannt und die Lösungs-Standardvektoren wurden mittels '.push_back(...)' mit Werten aufgefüllt. Der hier gewählte Weg ist für den Programmablauf vorteilhafter, da die im dynamischen Speicher bereitgestellte Größe der Lösungsvektoren sich während der Simulation sich nicht ständig anpassen muss.

4. Teilaufgabe

Vergleichen Sie beide Simulationen quantitativ.

Am Ende werden die berechneten Daten im Terminal ausgegeben. Leitet man die Terminalausgabe in eine Datei 'Achterbahn_1.dat' um und liest sie in das Jupyter Notebook 'Achterbahn_2.ipynb' (View Notebook, Download Notebook) ein, so kann man die numerische Lösung des C++ Programms mit den Python Simulationen vergleichen. Die nebenstehende Abbildung wurde mittels dieses Notebooks generiert und sie stellt die numerische Python Lösung ($N = 100000$, blaue Kurve $\vec{x}(t)$ und rote Kurve $\dot{\vec{x}}(t)$) im Vergleich zu den Ergebnissen des C++ Programmes ($N = 5000$, , schwarze durchgehende Kurve $\vec{x}(t)$ und schwarze gepunktete Kurve $\dot{\vec{x}}(t)$) dar. Man erkennt ein leichtes Abweichen der C++ Resultate bei späten Zeiten, wobei man durch Erhöhung von N zu den Python Resultaten konvergiert.

5. Teilaufgabe

Verallgemeinern Sie das C++ Programm so, dass mehrere Wagen auf der Achterbahn fahren können. Simulieren Sie ein System bestehend aus zwei wechselwirkungsfreien Wagen (siehe obige Animation) und benutzen Sie dabei für den ersten Wagen $i = 0$, $\tau_0 = 0$ und $\dot{x}_0(0) = 16.45\,[m/s]$ (blauer Wagen) und für den zweiten Wagen den ersten Wagen $i = 1$, $\tau_1 = 2\,[s]$ und $\dot{x}_1(2) = 5.5\,[m/s]$ (grüner Wagen).

Die in der vorigen Teilaufgabe konstruierte Klasse hatte die Eigenheit, dass sie im Anweisungsblock des Konstruktors die gesamte Berechnung der Wagenbewegung direkt bei Instanzbildung ausführte. Im Hinblick auf die weiteren Teilaufgaben (speziell das spätere Implementieren des elastischen Stoßes der beiden Wagen) ist es vorteilhaft, die Berechnung der numerischen Zeitschritte nach und nach zu erledigen und erst nach der Instanzbildung der Objekte durchzuführen. Auch hatten wir in der vorigen Teilaufgabe gesehen, dass es für eine genaue Berechnung eine größere Anzahl als 5000 Gitterpunkte benötigt, man jedoch auch nicht zu viele Datenwerte im Terminal ausgeben möchte.

Die Erweiterung des Programms auf mehrere Wagen wurde möglichst allgemein gehalten, damit z.B. eine weitere Einfügung eines dritten oder vierten Wagens einfach möglich ist. Die folgenden C++ Quelltexte (Achterbahn_2.cpp und Achterbahn_2.hpp) stellen eine Verallgemeinerung, der in der vorigen Teilaufgabe beschriebenen Programme, auf mehrere Wagen dar und wenden diese auf die oben angegebenen Anfangsbedingungen an.

Der Konstruktor der neuen Klasse 'Achterbahn_2' enthält nun fünf Argumente (Achterbahn_2(double t_, double alpha_1_, double alpha_2_, int N_sim_) : ... { ...}), die Anfangszeit, den Anfangsort und die Anfangsgeschwindigkeit des Wagens und die Anzahl der ausgegebenen Elemente der Lösungsvektoren. Im Gegensatz zu dem vorigen Konstruktor werden bei dessen Aufruf lediglich die Anfangswerte in die jeweiligen privaten Daten-Member initialisiert. Die eigentliche Berechnung und das Auffüllen der Lösungs-vector-Container erfolgt über eine öffentlich aufrufbare Memberfunktion 'Gehe_Zeitschritt(..)', die mittels einer Runge-Kutta Ordnung vier Methode einen gewissen Zeitbereich vorausberechnet und den letzten Punkt dieser Berechnung in die Lösungs-vector-Container als neues Element einfügt. Zusätzlich zu dieser Methode wurde eine weitere Funktion ' Ruhe_Zeitschritt()' implementiert (um dem verspäteten "in die Bahn katapultieren' des zweiten Wagens gerecht zu werden. Die DGL bestimmende Funktion wurde in der Klasse als eine virtuelle Funktion deklariert (näheres über virtuelle Funktionen und abstrakte Klassen finden Sie im zweiten Unterpunkt dieser Vorlesung), wobei die Definition dieser Funktion sich in der Datei Achterbahn_2.cpp befindet, sodass der Benutzer sie gegebenenfalls noch abändern kann, ohne die Klasse 'Achterbahn' verändern zu müssen.

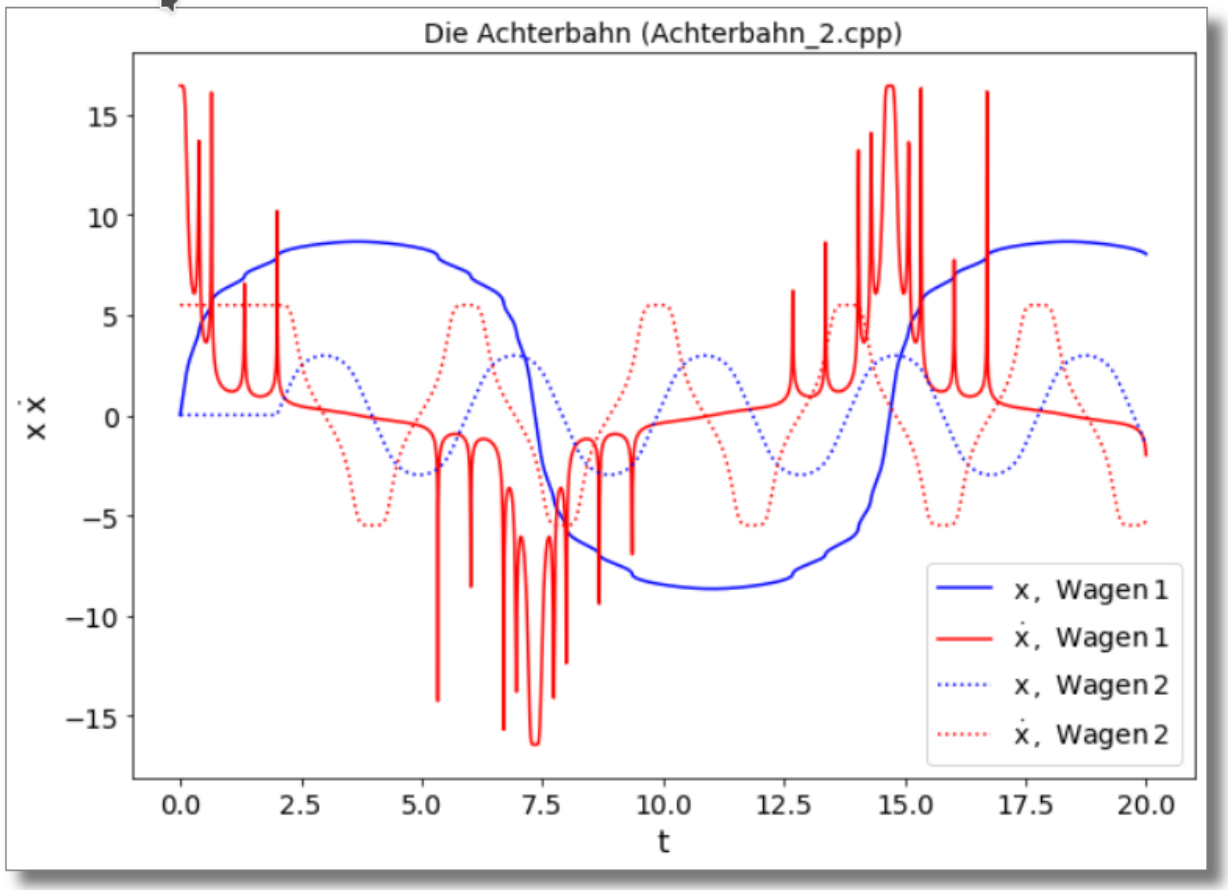
Achterbahn_2.hpp

```
/* Klasse zur Berechnung der Loesung einer Differentialgleichung der Form y''=f_2(t,y)
 * mittels Runge-Kutta Ordnung vier Verfahren
 * Verfahren zur Loesung der DGL ist in eine Klasse ausgelagert
 * Zeitentwicklung mittels einer Methode Gehe_Zeitschritt(...)
 * Konstruktor: Achterbahn_2(Anfangszeit a, Anfangswert1 alpha_1=y(a)=u_1(a), Anfangswert2 alpha_2=y'(a)=u_2(a), Anzahl der Zeitgitterpunkte der Simulation)
 */
#include <stdio.h>              // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>                // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
#include <vector>               // Vector-Container der Standardbibliothek

class Achterbahn_2{             //Definition der Klasse 'Achterbahn_2'
    double t = 0;
    double alpha_1 = 0.0;       // 1.Anfangswert bei t=a: u_1(a)=alpha_1
    double alpha_2 = 0.0;       // 2.Anfangswert bei t=a: u_2(a)=alpha_2
    double k1_1,k2_1,k3_1,k4_1; // Deklaration der vier Runge-Kutta Parameter fuer u_1
    double k1_2,k2_2,k3_2,k4_2; // Deklaration der vier Runge-Kutta Parameter fuer u_2
    int index = 0;
    int N_sim = 2500;           // Anzahl der Gitter-Zeitpunkte der Simulation (Dimension der folgenden Loesungsvektoren)

    std::vector<double> u_RungeK_4_1; // Deklaration eines double Vektors zum Speichern der Loesung fuer u_1
    std::vector<double> u_RungeK_4_2; // Deklaration eines double Vektors zum Speichern der Loesung fuer u_2
    std::vector<double> Zeit;         // Deklaration eines double Vektors zum Speichern der Zeit-Werte

    public:
        // Konstruktor mit fuenf Argumenten (Initialisierung der Parameter, Berechnung der Loesung der DGL)
        Achterbahn_2(double t_, double alpha_1_, double alpha_2_, int N_sim_) : t(t_),alpha_1(alpha_1_),alpha_2(alpha_2_),N_sim(N_sim_) {
            Zeit.resize( N_sim + 2 );               // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht
            u_RungeK_4_1.resize( N_sim + 2 );       // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht
            u_RungeK_4_2.resize( N_sim + 2 );       // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht

            Zeit[0] = t_;                           // Zum Zeit-Vektor die Endzeit eintragen
            u_RungeK_4_1[0] = alpha_1;              // Zum y-Vektor den Anfangswert alpha_1=y(a) eintragen
            u_RungeK_4_2[0] = alpha_2;              // Zum y'-Vektor den Anfangswert alpha_2=y'(a) eintragen
        };                                          // Ende des Konstruktors

        double f_1(double t, double u_1, double u_2){      // Deklaration und Definition der Funktion f_1(t,u_1,u_2)
            double wert;
            wert = u_2;                                     // Eigentliche Definition der Funktion
            return wert;                                    // Rueckgabewert der Funktion f_1
        }                                                   // Ende der Funktion f_1

        virtual double f_2(double t, double u_1, double u_2); // Deklaration der virtuellen Member-Funktion f_1(t,y) (Definition findet ausserhalb der Klasse statt)

        double get_zeit(int element) const { return Zeit[element]; }       // Definition der konstanten Member-Funktion get_zeit(i), Rueckgabewert i-ter Zeitpunkt
        double get_u1(int element) const { return u_RungeK_4_1[element]; } // Definition der konstanten Member-Funktion get_u1(i), Rueckgabewert i-ter Wert y(t_i)=u_1(t_i) der DGL
        double get_u2(int element) const { return u_RungeK_4_2[element]; } // Definition der konstanten Member-Funktion get_u2(i), Rueckgabewert i-ter Wert y'(t_i)=u_2(t_i) der DGL

        //Oeffentliche Methode Ruhe_Zeitschritt( ...)
        void Ruhe_Zeitschritt(double h_){
            Zeit[index] = Zeit[index] + h_;                         // Zum Zeit-Vektor die neue Zeit eintragen
            u_RungeK_4_1[index+1] = u_RungeK_4_1[index];            // Zum y-Vektor den neuen Wert eintragen
            u_RungeK_4_2[index+1] = u_RungeK_4_2[index];            // Zum y'-Vektor den neuen Wert eintragen
            index++;                                                // index um eins erhoehen
        }

        //Oeffentliche Methode Gehe_Zeitschritt( ...)
        void Gehe_Zeitschritt(double a, double b, int N){
            double h = (b - a)/N;           // Abstand dt zwischen den aequidistanten Punkten des t-Intervalls (h=dt)
            double u_1 = u_RungeK_4_1[index]; // Definition der lokalen Variable u_1=y und Initialisierung mit dem letzten berechneten Wert
            double u_2 = u_RungeK_4_2[index]; // Definition der lokalen Variable u_2=y' und Initialisierung mit dem letzten berechneten Wert

            Zeit[index] = a;                                // Zum Zeit-Vektor den aktuellen Zeitwert eintragen
            for(int i=0; i < N; ++i){                       // for-Schleife ueber die einzelnen Punkte des t-Intervalls
                t = a + i*h;                                // Zeit-Parameter wird um h erhoeht

                k1_1 = h*f_1(t,u_1,u_2);                    // Runge-Kutta Parameter k1 fuer u_1
                k1_2 = h*f_2(t,u_1,u_2);                    // Runge-Kutta Parameter k1 fuer u_2
                k2_1 = h*f_1(t+h/2,u_1+k1_1/2,u_2+k1_2/2);  // Runge-Kutta Parameter k2 fuer u_1
                k2_2 = h*f_2(t+h/2,u_1+k1_1/2,u_2+k1_2/2);  // Runge-Kutta Parameter k2 fuer u_2
                k3_1 = h*f_1(t+h/2,u_1+k2_1/2,u_2+k2_2/2);  // Runge-Kutta Parameter k3 fuer u_1
                k3_2 = h*f_2(t+h/2,u_1+k2_1/2,u_2+k2_2/2);  // Runge-Kutta Parameter k3 fuer u_2
                k4_1 = h*f_1(t+h,u_1+k3_1,u_2+k3_2);        // Runge-Kutta Parameter k4 fuer u_1
                k4_2 = h*f_2(t+h,u_1+k3_1,u_2+k3_2);        // Runge-Kutta Parameter k4 fuer u_2
                u_1 = u_1 + (k1_1 + 2*k2_1 + 2*k3_1 + k4_1)/6;
                u_2 = u_2 + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6;
            }                                               // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls

            u_RungeK_4_1[index+1] = u_1;                    // Zum y-Vektor den neuen Wert eintragen
            u_RungeK_4_2[index+1] = u_2;                    // Zum y'-Vektor den neuen Wert eintragen
            index++;                                        // index um eins erhoehen
        }                                                   // Ende der Funktion Gehe_Zeitschritt( ...)
};                                                          // Ende der Klasse
```

Achterbahn_2.cpp

```cpp
// Musterlösung Projekt Achterbahn (Aufgabe 1)
/* Berechnung der Lösung einer Differentialgleichung der Form x'=f(t,x)
 * mittels Runge-Kutta Ordnung vier Verfahren
 * Verfahren zur Loesung der DGL ist in eine Klasse ausgelagert
 * Zeitentwicklung der fuer unterschiedliche t-Werte in [a,b]
 * Zwei Wagen werden als Instanzen der Klasse Achterbahn_2 in einem vector-Container definiert
 */

#include "Achterbahn_2.hpp"    // Einbinden der Klasse die das System der DGLs mittels Runge-Kutta Ordnung vier loest
#include <stdio.h>             // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>               // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
#include <vector>              // Vector-Container der Standardbibliothek
using namespace std;           // Benutze den Namensraum std

double Achterbahn_2::f_2(double t, double u_1, double u_2){          // Deklaration und Definition der Funktion f_2(t,u_1,u_2), berechnet mittels des Jupyter Notebooks Achterbahn.ipynb
    double wert;
    wert = 0.20000000000000001*u_1*(-32.0*pow(u_1, 4)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(cos(0.10000000000000001*pow(u_1, 2)), 3) +
    16.0*pow(u_1, 4)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2)) +
    75.894663844041105*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 3)*fabs(u_1) +
    80.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 3) -
    63.245553203367592*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2))*fabs(u_1) -
    80.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2)) +
    60.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(cos(0.10000000000000001*pow(u_1, 2)), 3) -
    60.0*pow(u_1, 2)*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2)) -
    1962.0*pow(u_1, 2)*exp(0.316227766016838*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2)) -
    15.811388300841898*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 3)*fabs(u_1) +
    63.245553203367592*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*pow(cos(0.10000000000000001*pow(u_1, 2)), 3)*fabs(u_1) -
    63.245553203367592*pow(u_2, 2)*exp(0.63245553203367599*fabs(u_1))*cos(0.10000000000000001*pow(u_1, 2))*fabs(u_1) -
    1551.0971923125903*exp(0.316227766016838*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2))*fabs(u_1))*sin(0.10000000000000001*pow(u_1, 2))/(100.0*pow(u_1, 2) +
    pow(4.0*pow(u_1, 2)*cos(0.10000000000000001*pow(u_1, 2)) +
    3.1622776601683795*sin(0.10000000000000001*pow(u_1, 2))*pow(fabs(u_1), 1.0), 2)*exp(0.63245553203367599*fabs(u_1))*pow(sin(0.10000000000000001*pow(u_1, 2)), 2));
    return wert;                                                      // Rueckgabewert der Funktion f_2
}                                                                     // Ende der Funktion f_2

int main(){                         // Hauptfunktion
    double a = 0;                   // Untergrenze des Zeit-Intervalls [a,b] in dem die Loesung berechnet werden soll
    double b = 20;                  // Obergrenze des Intervalls [a,b]
    int N_RK = 100;                 // Anzahl der Gitter-Zeitpunkte des Runge-Kutta Ordnung vier Verfahrens
    int N_sim = 2500;               // Anzahl der Gitter-Zeitpunkte der Simulation (Anzahl der ausgegebenen Punkte)
    double h_sim = (b - a)/N_sim; // Abstand dt zwischen den aequidistanten Punkten des Sim-Intervalls (h=dt)
    double t;                       // Aktueller Zeitwert

    vector<double> tau = {0.0, 2.0 };   // Deklaration eines double Vektors zum Speichern der Anfangszeiten der sechs Wagen
    vector<double> v_0 = {16.45, 5.5 }; // Deklaration eines double Vektors zum Speichern der Anfangszeiten der sechs Wagen

    vector<Achterbahn_2> Wagen;         // Deklaration eines vector-Containers der Wagen

    for (unsigned int n = 0; n < tau.size(); ++n){                // for-Schleife zum Auffuellen des Containers mit Wagen-Elementen
        Wagen.push_back( Achterbahn_2 {a, 0.01, v_0[n], N_sim } ); // Initialisierung der Wagen und einfuegen in den Container
    }                                                             // Ende for-Schleife

    for(int i = 0; i <= N_sim; ++i){                              // for-Schleife ueber die Simulations-Zeitpunkte
        t = a + i * h_sim;                                        // Zeit-Parameter wird um h_sim erhoeht
        for(unsigned int n = 0; n < tau.size(); ++n){             // for-Schleife ueber die einzelnen Wagen
            if( t >= tau[n] ){                                    // if-Anweisung: Ist Wagen schon auf der Achterbahn?
                Wagen[n].Gehe_Zeitschritt(t, t+h_sim, N_RK);      // Aufruf der Runge-Kutta-Methode 'Gehe_Zeitschritt(...)' in der Klasse Achterbahn_2
            } else {                                              // else-Anweisung: Ist Wagen schon auf der Achterbahn?
                Wagen[n].Ruhe_Zeitschritt(h_sim);                 // Aufruf der Runge-Kutta-Methode 'Ruhe_Zeitschritt(...)' in der Klasse Achterbahn_2
            }                                                     // Ende else-Anweisung
        }                                                         // Ende for-Schleife ueber die einzelnen Wagen
    }                                                             // Ende for-Schleife ueber die Simulations-Zeitpunkte

    printf("# 0: Index i \n# 1: t-Wert \n"); // Beschreibung der ausgegebenen Groessen
    printf("# 2: x(t), Wagen 1 \n# 3: v_x(t), Wagen 1 \n");
    printf("# 4: x(t), Wagen 2 \n# 5: v_x(t), Wagen 2 \n");

    for(int i=0; i  <= N_sim; ++i){                               // for-Schleife zur Terminalausgabe der Loesung (Zeitgitterpunkte)
        printf("%5d %19.15f ",i, Wagen[0].get_zeit(i));           // Ausgabe der Simulierten Zeitwerte mittels der Methode get_zeit()
        for(auto& n : Wagen){                                     // for-Schleife zur Terminalausgabe der Loesung (Wagen)
            printf("%19.15f %19.15f ", n.get_u1(i), n.get_u2(i)); // Ausgabe der Simulierten Orts- und Geschwindigkeitswerte mittels der Methoden n.get_u_1() und n.get_u_2()
        }                                                         // Ende for-Schleife (Wagen)
        printf("\n");
    }                                                             // Ende for-Schleife (Zeitgitterpunkte)
}                                                                 // Ende der Hauptfunktion
```

Mehrere Wagen auf der Achterbahn



In dem Hauptprogramm Achterbahn_2.cpp wurde zwei Gitterpunktsvariablen definiert. N_{RK} 'int N_RK = 100;' beschreibt hierbei die Anzahl der Gitter-Zeitpunkte des Runge-Kutta Ordnung vier Verfahrens und N_{sim} 'int N_sim = 2500; ' die Anzahl der Gitter-Zeitpunkte der Simulation (Anzahl der im Terminal ausgegebenen Punkte). Effektiv besitzt die durchgeführte Berechnung somit $N = N_{sim}$·N_{RK}= 250000 Zeitgitterpunkte, wobei im Terminal lediglich 2500 Datenpunkte ausgegeben werden.

```
for (unsigned int n = 0; n < tau.size(); ++n){
    Wagen.push_back( Achterbahn_2 {a, 0.01, v_0[n], N_sim } );
}
```

Zusätzlich wird im Hauptprogramm ein eigener vector-Container bestehend aus Elementen des Typs unsrer Klasse 'dsolve_Sys_2' erzeugt. Da die Elemente dieses Containers die einzelnen Wagen repräsentieren werden, wurde als Name des Containers 'Wagen gewählt (vector<dsolve_Sys_2> Wagen;). Die Anfangszeiten τ_i und Anfangsgeschwindigkeiten $\dot{x}_i(\tau_i)$ der Wagen wurden im Hinblick auf mehrere Wagen als vector-Container definiert. Das Auffüllen des Wagen-Containers und die Instanzbildung der Wagen erfolgt der Allgemeinheit halber innerhalb der, oben rechts abgebildeten for-Schleife.

6. Teilaufgabe

Um die Richtigkeit der berechneten Daten des C++ Programmes zu verifizieren, führen Sie (nur in dieser Teilaufgabe) eine separate Simulation mit abgeänderter Achterbahnform durch, bei der es eine analytische Lösung gibt. Berechnen Sie zunächst auf analytischem Weg, mittels eines Jupyter Notebooks unter Verwendung der SymPy Bibliothek, die Euler-Lagrange Gleichungen der Bewegung eines Massenpunktes der Masse m auf einer Zykloidenbahn (siehe Aufgabe 15.7: Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel V15. Seite 271]). Der Massepunkt gleitet reibungsfrei auf einer Zykloide , die durch

$$x(\theta) = a\,(\theta - \sin(\theta)) \;\;,\;\; y(\theta) = a\,(1 + \cos(\theta)) \;\;,\quad 0 \leq \theta(t) \leq 2\pi$$

gegeben ist. Vergleichen Sie Ihre Werte mit der analytischen Lösung in Aufgabe 15.7 (Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel V15. Seite 271]). Vergleichen Sie auch die Resultate einer entsprechenden Simulation mittels Ihres C++ Programmes.

Gerne können Sie auch dieses Projekt weiter bearbeiten und einige der noch nicht bearbeiteten Teilaufgaben lösen ...

7. Teilaufgabe

Implementieren Sie nun einen elastischen Stoß der zwei Wagen in Ihr C++ Programm und führen Sie die obere Animation nochmals durch.

.....

Abgeleitete Klassen, Vererbung von Klassenmerkmalen und Klassenhierarchien

Wir hatten in der Vorlesung 7 in das Konzept der *objektorientierten Programmierung* eingeführt und gesehen, dass man mittels des Konzeptes einer C++ Klasse selbstdefinierte Objekte im Programm realisieren kann. Eine Klasse stellt eine formale Beschreibung dar (ein Bauplan), wie das Objekt (bzw. die in Programmcode formulierte Idee) beschaffen ist, und definiert, welche Merkmale (Instanzvariablen bzw. Daten-Member der Klasse) und Verhaltensweisen (Methoden der Klasse bzw. Member-Funktionen) das zu beschreibende Objekt hat. Bei der Konstruktion einer Klassenstruktur eines Programmes kann es nun geschehen, dass man das Programm in mehrere Teilideen/Teilkonzepte unterteilen kann und die so entworfenen Klassen stehen dann häufig in Beziehung zueinander. So können z.B. einige Klassen des Programms auch die Daten-Member und Verhaltensweisen von anderen Klassen verwenden (Vererbung von Merkmalen) und es stellt sich nun die Frage, wie man diese Beziehungen in der Programmiersprache C++ direkt ausdrücken kann. Das Konzept der *abgeleiteten Klasse* und die damit verbundenen C++ Sprachmechanismen dienen dazu, hierarchische Beziehungen, d.h. Gemeinsamkeiten zwischen den Klassen, auszudrücken.

- **Implementierungsvererbung**:
  Indem eine abgeleitete Klasse die Instrumente der Basisklasse erbt, spart sich der Programmierer Schreibaufwand und die übergeordnete Struktur des Programms wird übersichtlicher.
- **Schnittstellenvererbung**:
  In vielen größeren Programmen ist es nötig eine Art von übergeordnete Schnittstelle in einer Basisklasse bereitzustellen, die es erlaubt verschiedene abgeleitete Klassen über sie zu verwenden.

In diesem Unterpunkt werden wir in den Themenberech der *abgeleiteten Klassen* und *Vererbung* einführen, den man grob in die neben abgebildeten zwei Varianten einordnen kann. Bei der *Implementierungsvererbung* spart sich der Programmierer oft viel Schreibarbeit und oft wird durch die Vererbungsstruktur seiner Klassen das Programmkonzept übersichtlicher und ist leichter zu verstehen. Bei der *Schnittstellenvererbung* spricht man auch von *Polymorphie zur Laufzeit*, bzw. von *dynamischer Polymorphie*. Die Basisklasse stellt hierbei einen Schnittstelle dar, die ähnlich einer switch-Anweisung, aus unterschiedlichen Programmsträngen auswählt und die einzelnen, möglichen case-Marken sind durch die jeweiligen abgeleiteten Klassen im Programm implementiert. Eine solche polymorphe Struktur kann man, alternativ auch elegant mittels des *Template-Konzeptes* formulieren, wobei man in einem solchen Fall von *Polymorphie zur Übersetzungszeit*, bzw. von *statischer Polymorphie* (näheres siehe nächste Vorlesung).

Abgeleitete Klassen und die Vererbung von Klassenmerkmalen

Abgeleitete Klassen und die Vererbung von Klassenmerkmalen

Es wird nun die formale Struktur einer abgeleiteten Klasse (auch Subklasse) vorgestellt, die von einer Basisklasse (auch Oberklasse oder Superklasse) abgeleitet wird und mittels des Sprachkonzeptes der Vererbung Merkmale übernimmt.

class Base { 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };

class Sub : public Base{ 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };

Base_SubClass_0.cpp

```cpp
/* Beispiel einer einfachen abgeleiteten Klasse
*/
#include <iostream>             // Ein- und Ausgabebibliothek
using namespace std;            // Benutze den Namensraum std

//Definition der Basisklasse 'Base'
class Base{
    // Private Instanzvariablen (Daten-Member) der Klasse
    unsigned int n;
    double x;

    // Oeffentlicher Bereich der Klasse
    public:
        // Konstruktor mit zwei Argumenten
        Base(unsigned int set_n, double set_x) : n{set_n}, x{set_x} {
            printf("Konstruktor der Klasse Base \n");
        }

        // Member-Funktionen der Klasse
        // als const deklariert, da sie die privaten Instanzvariablen nicht veraendern
        unsigned int get_Nummer() const {return n;}
        double get_Ort() const {return x;}
};

//Definition der abgeleiteten Klasse 'Sub'
class Sub : public Base{
    // Private Instanzvariablen (Daten-Member) der Klasse
    double m = 1;

    // Oeffentlicher Bereich der Klasse
    public:
        // Konstruktor mit drei Argumenten
        Sub(unsigned int set_n, double set_x, double set_m) : Base(set_n, set_x), m{set_m} {
            printf("Konstruktor der Klasse Sub \n");
        }

        // Member-Funktionen der Klasse
        // als const deklariert, da sie die privaten Instanzvariablen nicht veraendern
        double get_m() const {return m;}
};

int main(){                         // Hauptfunktion
    Base Ding_0 = Base(0, 2.5);     // Konstruktor bilded eine Instanz der Klasse Base
    Sub Ding_1 = Sub(1, 3.5, 10.2); // Konstruktor bilded eine Instanz der abgeleiteten Klasse Sub

    // Terminalausgabe
    printf("Das Ding %2d befindet sich am Ort x=%5.2f \n", Ding_0.get_Nummer(), Ding_0.get_Ort());
    printf("Das Ding %2d befindet sich am Ort x=%5.2f und hat die Masse m=%5.2f \n", Ding_1.get_Nummer(), Ding_1.get_Ort(), Ding_1.get_m());
}
```

Base_SubClass_1.cpp

```cpp
/* Beispiel einer einfachen abgeleiteten Klasse (alle Merkmale öffentlich)
*/
#include <iostream>             // Ein- und Ausgabebibliothek
using namespace std;            // Benutze den Namensraum std

//Definition der Basisklasse 'Base'
class Base{
    // Oeffentlicher Bereich der Klasse
    public:
        // Instanzvariablen (Daten-Member)
        unsigned int n;
        double x;

        // Konstruktor mit zwei Argumenten
        Base(unsigned int set_n, double set_x) : n{set_n}, x{set_x} {
            printf("Konstruktor der Klasse Base \n");
        }
};

//Definition der abgeleiteten Klasse 'Sub'
class Sub : public Base{
    // Oeffentlicher Bereich der Klasse
    public:
        // Instanzvariablen (Daten-Member) der Klasse
        double m = 1;

        // Konstruktor mit drei Argumenten
        Sub(unsigned int set_n, double set_x, double set_m) : Base(set_n, set_x), m{set_m} {
            printf("Konstruktor der Klasse Sub \n");
        }
};

int main(){                         // Hauptfunktion
    Base Ding_0 = Base(0, 2.5);     // Konstruktor bilded eine Instanz der Klasse Base
    Sub Ding_1 = Sub(1, 3.5, 10.2); // Konstruktor bilded eine Instanz der abgeleiteten Klasse Sub

    // Terminalausgabe
    printf("Das Ding %2d befindet sich am Ort x=%5.2f \n", Ding_0.n, Ding_0.x);
    printf("Das Ding %2d befindet sich am Ort x=%5.2f und hat die Masse m=%5.2f \n", Ding_1.n, Ding_1.x, Ding_1.m);
}
```

Vererbung_0.cpp

```cpp
/* Beispiel einer einfachen Klasse
 * Zwei private Vektoren (Ort,Geschwindigkeit) als Instanzvariablen,
 * Konstruktor fuer das Initialisieren
 * Drei oeffentliche const Member-Funktionen
*/
#include <iostream>             // Ein- und Ausgabebibliothek
#include <vector>               // Vector-Container der Standardbibliothek
using namespace std;            // Benutze den Namensraum std

//Definition der Klasse 'Ding'
class Ding{
    // Private Instanzvariablen (Daten-Member) der Klasse
    double t = 0.0;                         // Aktueller Zeitpunkt
    vector<double> r = { 0.0, 0.0, 0.0 }; // Ortsvektor zum Ort des Dings ++
    vector<double> v = { 0.0, 0.0, 0.0 }; // Geschwindigkeit des Dings

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Konstruktor mit drei Argumenten zum Initialisieren
        Ding(double t_, vector<double> r_, vector<double> v_) : t{t_}, r{r_}, v{v_} { }

        // Member-Funktionen der Klasse
        // Drei als const deklariert, da sie die privaten Instanzvariablen nicht veraendern
        double get_zeit() const {return t;}
        vector<double> get_r() const {return r;}
        vector<double> get_v() const {return v;}
};

int main(){                                                     // Hauptfunktion
    Ding Ding_A = Ding(0.0, {1.0, 2.0, 3.0 }, {10.0, 0.0, 0.0 }); // Konstruktor bilded eine Instanz der Klasse Ding

    // Terminalausgabe von Ort und Geschwindigkeit
    printf("Das Ding A befindet sich am Ort (%5.2f,%5.2f,%5.2f )", Ding_A.get_r()[0], Ding_A.get_r()[1], Ding_A.get_r()[2]);
    printf(" und hat die Geschwindigkeit (%5.2f,%5.2f,%5.2f ) \n", Ding_A.get_v()[0], Ding_A.get_v()[1], Ding_A.get_v()[2]);
}
```

Vererbung_1.cpp

```cpp
/* Beispiel einer von der Basisklasse 'Ding' abgeleitete Klasse 'Pendel'
 * Zwei zusaetzliche private Instanzvariablen (l und beta) und die Winkelspezifischen Anfangswerte (phi und v_phi)
 * wurden der abgeleiteten Pendel-Klasse hinzugefuegt
 * Der Konstruktor der Klasse Pendel erzeugt ein Objekt Ding und initialisiert seine eigenen Daten-Member
*/
#include <iostream>             // Ein- und Ausgabebibliothek
#include <cmath>                // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
#include <vector>               // Vector-Container der Standardbibliothek
using namespace std;            // Benutze den Namensraum std

//Definition der Klasse 'Ding'
class Ding{
    // Private Instanzvariablen (Daten-Member) der Klasse
    double t = 0.0;                         // Aktueller Zeitpunkt
    vector<double> r = { 0.0, 0.0, 0.0 }; // Ortsvektor zum Ort des Dings ++
    vector<double> v = { 0.0, 0.0, 0.0 }; // Geschwindigkeit des Dings

    // Oeffentlicher Bereich
    public:
        // Konstruktor mit drei Argumenten zum Initialisieren
        Ding(double t_, vector<double> r_, vector<double> v_) : t{t_}, r{r_}, v{v_} { }

        vector<double> get_r() const { return r; } // Definition der konstanten Member-Funktion get_r, Rueckgabewert vector des Ortes
        vector<double> get_v() const { return v; } // Definition der konstanten Member-Funktion get_v, Rueckgabewert vector der Geschwindigkeit
        double get_zeit() const { return t; }      // Definition der konstanten Member-Funktion get_zeit(), Rueckgabewert des aktuellen Zeitpunktes
};

//Definition der Klasse 'Pendel' (hier nur eindimensionales (x,y), beschrieben durch Winkel phi, Aufhaengepunkt bei (0,0,0))
class Pendel : public Ding {
    // Private Instanzvariablen (Daten-Member) der Klasse
    double l = 1.0;           // Laenge des Pendels
    double beta = 0.0;        // Reibungskoeffizient
    double phi = 0.0;         // Anfangswinkel des Pendels
    double v_phi = 0.0;       // Anfangswinkelgeschwindigkeit des Pendels

    public:
        // Konstruktor erzeugt ein Objekt Ding und initialisiert die Laenge l des Pendels und den Reibungskoeffizienten beta
        Pendel(double t_, double phi_, double v_phi_, double l_, double beta_) :
        Ding( t_,  { l_ * sin(phi), - l_ * cos(phi), 0 },  { l_ * cos(phi) * v_phi_, l_ * sin(phi) * v_phi_, 0 }), phi{phi_}, v_phi{v_phi_} , l{l_}, beta{beta_} { }
};

int main(){                                             // Hauptfunktion
    Pendel Pendel_A = Pendel(0.0, 0.0, 10.0, 1.0, 0.0);           // Konstruktor bilded eine Instanz der Klasse Pendel

    // Terminalausgabe von Ort und Geschwindigkeit
    printf("Das Pendel A befindet sich am Ort (%5.2f,%5.2f,%5.2f )", Pendel_A.get_r()[0], Pendel_A.get_r()[1], Pendel_A.get_r()[2]);
    printf(" und hat die Geschwindigkeit (%5.2f,%5.2f,%5.2f ) \n", Pendel_A.get_v()[0], Pendel_A.get_v()[1], Pendel_A.get_v()[2]);
}
```

DGL des physikalischen Pendels mit Reibung

Wir möchten nun die Bewegung des Pendels für einen gewissen Zeitraum simulativ berechnen. Betrachten wir z.B. das einfache physikalische Pendel aus dem Themenbereich der Mechanik (siehe z.B. W.Greiner, Klassische Mechanik I). Die zugrundeliegende Differentialgleichung (DGL) des Problems lautet

$$\frac{d^2\phi(t)}{dt^2} = -\frac{g}{l} \cdot \sin\left(\phi(t)\right) - \beta \cdot \frac{d\phi(t)}{dt} \quad ,$$

wobei g die Erdbeschleunigung, l die Länge des Pendels, β der Reibungsparameter (Stokesscher Ansatz) und $\phi(t)$ die zeitliche Entwicklung des Pendelwinkels beschreibt. Wir schreiben diese DGL (2.Ordnung) in ein System bestehend aus zwei DGLs erster Ordnung um, und benutzen zum numerischen Lösen des Systems, die in der Musterlösung des Übungsblattes Nr. 9 vorgestellte Funktion des Runge-Kutta Ordung vier Verfahrens. Dieses Verfahren implementieren wir

Vererbung_3.cpp

```cpp
/* Beispiel einer von der Basisklasse 'Ding' abgeleitete Klasse 'Pendel'
 * Zwei zusaetzliche private Instanzvariablen (l und beta) und die Winkelspezifischen Anfangswerte (phi und v_phi)
 * wurden der abgeleiteten Pendel-Klasse hinzugefuegt
 * Der Konstruktor der Klasse Pendel erzeugt ein Objekt Ding und initialisiert seine eigenen Daten-Member
*/
#include <iostream>             // Ein- und Ausgabebibliothek
#include <cmath>                // Bibliothek für mathematisches (e-Funktion, Betrag, ...)
#include <vector>               // Vector-Container der Standardbibliothek
using namespace std;            // Benutze den Namensraum std

//Definition der Klasse 'Ding'
class Ding{
    public:
        // Oeffentliche Instanzvariablen (Daten-Member) der Klasse
        double t = 0.0;                         // Aktueller Zeitpunkt
        vector<double> r = { 0.0, 0.0, 0.0 }; // Ortsvektor zum Ort des Dings ++
        vector<double> v = { 0.0, 0.0, 0.0 }; // Geschwindigkeit des Dings

        // Konstruktor mit drei Argumenten zum Initialisieren
        Ding(double t_, vector<double> r_, vector<double> v_) : t{t_}, r{r_}, v{v_} { }
};

//Definition der Klasse 'Pendel' (hier nur eindimensionales (x,y), beschrieben durch Winkel phi, Aufhaengepunkt bei (0,0,0))
class Pendel : public Ding {
    // Private Instanzvariablen (Daten-Member) der Klasse
    double l = 1.0;       // Laenge des Pendels
    double beta = 0.0;    // Reibungsparameter

    vector<double> phi;   // Deklaration eines double Vektors zum Speichern der Loesung fuer u_1
    vector<double> v_phi; // Deklaration eines double Vektors zum Speichern der Loesung fuer u_2
    vector<double> Zeit;  // Deklaration eines double Vektors zum Speichern der Zeit-Werte
    unsigned N_sim = 2500;            // Anzahl der Gitter-Zeitpunkte der Simulation (Dimension der folgenden Loesungsvektoren)
    unsigned index = 0;

    double k1_1,k2_1,k3_1,k4_1; // Deklaration der vier Runge-Kutta Parameter fuer u_1
    double k1_2,k2_2,k3_2,k4_2; // Deklaration der vier Runge-Kutta Parameter fuer u_2

    public:
        // Konstruktor erzeugt ein Objekt Ding und initialisiert die Laenge l des Pendels und den Reibungskoeffizienten beta
        Pendel(double t_, double phi_, double v_phi_, double l_, double beta_, unsigned N_sim_) :
        Ding( t_,  { l_ * sin(phi_), - l_ * cos(phi_), 0 },  { l_ * cos(phi_) * v_phi_, l_ * sin(phi_) * v_phi_, 0 }), l{l_}, beta{beta_}, N_sim{N_sim_}{
            Zeit.resize( N_sim + 2 );         // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht
            phi.resize( N_sim + 2 );          // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht
            v_phi.resize( N_sim + 2 );        // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht

            Zeit[0] = t_;                     // Zum Zeit-Vektor die Endzeit eintragen
            phi[0] = phi_;                    // Zum y-Vektor den Anfangswert alpha_1=y(a) eintragen
            v_phi[0] = v_phi_;                // Zum y'-Vektor den Anfangswert alpha_2=y'(a) eintragen
        }
```

```cpp
//Definition der Klasse 'Pendel' (hier nur eindimensionales (x,y), beschrieben durch Winkel phi, Aufhaengepunkt bei (0,0,0))
class Pendel : public Ding {
    // Private Instanzvariablen (Daten-Member) der Klasse
    double l = 1.0;       // Laenge des Pendels
    double beta = 0.0;    // Reibungsparameter

    vector<double> phi;   // Deklaration eines double Vektors zum Speichern der Loesung fuer u_1
    vector<double> v_phi; // Deklaration eines double Vektors zum Speichern der Loesung fuer u_2
    vector<double> Zeit;  // Deklaration eines double Vektors zum Speichern der Zeit-Werte
    unsigned N_sim = 2500;            // Anzahl der Gitter-Zeitpunkte der Simulation (Dimension der folgenden Loesungsvektoren)
    unsigned index = 0;

    double k1_1,k2_1,k3_1,k4_1; // Deklaration der vier Runge-Kutta Parameter fuer u_1
    double k1_2,k2_2,k3_2,k4_2; // Deklaration der vier Runge-Kutta Parameter fuer u_2

    public:
        // Konstruktor erzeugt ein Objekt Ding und initialisiert die Laenge l des Pendels und den Reibungskoeffizienten beta
        Pendel(double t_, double phi_, double v_phi_, double l_, double beta_, unsigned N_sim_) :
        Ding( t_,  { l_ * sin(phi_), - l_ * cos(phi_), 0 },  { l_ * cos(phi_) * v_phi_, l_ * sin(phi_) * v_phi_, 0 }), l{l_}, beta{beta_}, N_sim{N_sim_}{
            Zeit.resize( N_sim + 2 );        // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht
            phi.resize( N_sim + 2 );         // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht
            v_phi.resize( N_sim + 2 );       // Die Anzahl der Eintraege im Vektor wird auf N+2 erhoeht

            Zeit[0] = t_;                    // Zum Zeit-Vektor die Endzeit eintragen
            phi[0] = phi_;                   // Zum y-Vektor den Anfangswert alpha_1=y(a) eintragen
            v_phi[0] = v_phi_;               // Zum y'-Vektor den Anfangswert alpha_2=y'(a) eintragen
        }

        double f_1(double t, double u_1, double u_2){      // Deklaration und Definition der Funktion f_1(t,u_1,u_2)
            double wert;
            wert = u_2;                                     // Eigentliche Definition der Funktion
            return wert;                                    // Rueckgabewert der Funktion f_1
        }                                                   // Ende der Funktion f_1

        double f_2(double t, double u_1, double u_2){      // Deklaration und Definition der Funktion f_2(t,u_1,u_2)
            double wert;
            wert = - 9.81 / l * sin(u_1) - beta * u_2;      // DGl des physikalischen Pendels mit Reibung (Stokesscher Ansatz)
            return wert;                                    // Rueckgabewert der Funktion f_2
        }                                                   // Ende der Funktion f_2

        void Gehe_Zeitschritt(double dt, int N){            // Einen Zeitschritt um dt weiter mittels Runge-Kutta
            double t_;
            double a = Zeit[index];    // Letzter berechneter Zeitpunkt
            double h = dt / N;         // Abstand dt zwischen den aequidistanten Punkten des t-Intervalls (h=dt)
            double u_1 = phi[index];   // Definition der lokalen Variable u_1=y und Initialisierung mit dem letzten berechneten Wert
            double u_2 = v_phi[index]; // Definition der lokalen Variable u_2=y' und Initialisierung mit dem letzten berechneten Wert

            for(int i=0; i <= N; ++i){                      // for-Schleife ueber die einzelnen Punkte des t-Intervalls
                t_ = a + i*h;                               // Zeit-Parameter wird um h erhoeht

                k1_1 = h*f_1(t_,u_1,u_2);                   // Runge-Kutta Parameter k1 fuer u_1
                k1_2 = h*f_2(t_,u_1,u_2);                   // Runge-Kutta Parameter k1 fuer u_2
                k2_1 = h*f_1(t_+h/2,u_1+k1_1/2,u_2+k1_2/2); // Runge-Kutta Parameter k2 fuer u_1
                k2_2 = h*f_2(t_+h/2,u_1+k1_1/2,u_2+k1_2/2); // Runge-Kutta Parameter k2 fuer u_2
                k3_1 = h*f_1(t_+h/2,u_1+k2_1/2,u_2+k2_2/2); // Runge-Kutta Parameter k3 fuer u_1
                k3_2 = h*f_2(t_+h/2,u_1+k2_1/2,u_2+k2_2/2); // Runge-Kutta Parameter k3 fuer u_2
                k4_1 = h*f_1(t_+h,u_1+k3_1,u_2+k3_2);       // Runge-Kutta Parameter k4 fuer u_1
                k4_2 = h*f_2(t_+h,u_1+k3_1,u_2+k3_2);       // Runge-Kutta Parameter k4 fuer u_2
                u_1 = u_1 + (k1_1 + 2*k2_1 + 2*k3_1 + k4_1)/6;
                u_2 = u_2 + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6;
            }                                               // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls

            Zeit[index+1] = t_;                             // Zum Zeit-Vektor den neuen Wert eintragen
            phi[index+1] = u_1;                             // Zum y-Vektor den neuen Wert eintragen
            v_phi[index+1] = u_2;                           // Zum y'-Vektor den neuen Wert eintragen
            index++;                                        // index um eins erhoehen

            t = t_;                                         // Neuen Zeit-Wert des Pendels in der Klasse Ding aktualisieren
            r = { l * sin(u_1), - l * cos(u_1), 0 };        // Neuen Ort Wert des Pendels in der Klasse Ding aktualisieren
            v = { l * cos(u_1) * u_2, l * sin(u_1) * u_2, 0 }; // Neuen Geschwindigkeitswert des Pendels in der Klasse Ding aktualisieren
        }                                                   // Ende der Gehe_Zeitschritt Funktion
};                                                          // Ende der Klasse 'Pendel'
```

```cpp
int main(){                         // Hauptfunktion
    double a = 0.0;                 // Untergrenze des Zeit-Intervalls
    double b = 0.95;                // Untergrenze des Zeit-Intervalls
    int N_RK = 100;                 // Anzahl der Gitter-Zeitpunkte des Runge-Kutta Ordnung vier Verfahrens
    int N_sim = 1000;               // Anzahl der Gitter-Zeitpunkte der Simulation (Anzahl der ausgegebenen Punkte)
    double dt = (b - a)/N_sim;      // Abstand dt zwischen den aequidistanten Punkten des Sim-Intervalls (h=dt)
    double t;                       // Aktueller Zeitwert

    Pendel Pendel_A = Pendel(a, 0.0, 40.0, 0.5, 0.0, N_sim); // Konstruktor bilded eine Instanz der Klasse Pendel und erzeugt implizit ein Objekt Ding

    printf("# 0: Index i \n# 1: t-Wert \n"); // Beschreibung der ausgegebenen Groessen
    printf("# 2: x(t) \n# 3: y(t) \n");
    printf("# 4: v_x(t) \n# 5: v_y(t) \n");
    printf("%5d %19.15f %19.15f %19.15f %19.15f %19.15f \n",0, Pendel_A.t, Pendel_A.r[0], Pendel_A.r[1], Pendel_A.v[0], Pendel_A.v[1]);

    // Terminalausgabe von Ort und Geschwindigkeit des Pendels
    for(int i=1; i  <= N_sim; ++i){                          // for-Schleife zur Terminalausgabe der Loesung (Zeitgitterpunkte)
        Pendel_A.Gehe_Zeitschritt(dt, N_RK);                 // Aufruf der Methode Gehe_Zeitschritt(dt, N) der Klasse Pendel
        printf("%5d %19.15f %19.15f %19.15f %19.15f %19.15f \n",i, Pendel_A.t, Pendel_A.r[0], Pendel_A.r[1], Pendel_A.v[0], Pendel_A.v[1]);
    }                                                        // Ende for-Schleife (Zeitgitterpunkte)
}                                                            // Ende main()-Funktion
```

Pendel.hpp

```cpp
/* Beispiel einer von der Basisklasse 'Ding' abgeleitete Klasse 'Pendel' und einer davon abgeleiteten Sub-Subklasse Pendel_math
 * Zwei zusaetzliche private Instanzvariablen (l und beta) und die Winkelspezifischen Anfangswerte (phi und v_phi)
 * wurden der abgeleiteten Pendel-Klasse hinzugefuegt
 * Der Konstruktor der Klasse Pendel erzeugt ein Objekt Ding und initialisiert seine eigenen Daten-Member
 * Der Konstruktor der, von der Klasse Pendel abgeleiteten Subklasse Pendel_math, erzeugt ein Objekt Pendel (und implizit somit auch ein Objekt Ding`)
*/
#include <iostream>             // Ein- und Ausgabebibliothek
#include <cmath>                // Bibliothek fuer mathematisches (e-Funktion, Betrag, ...)
#include <vector>               // Vector-Container der Standardbibliothek
using namespace std;            // Benutze den Namensraum std

// Definition der Klasse 'Ding'
class Ding{
    public:
        // Oeffentliche Instanzvariablen (Daten-Member) der Klasse
        double t = 0.0;                         // Aktueller Zeitpunkt
        vector<double> r = { 0.0, 0.0, 0.0 }; // Ortsvektor zum Ort des Dings ++
        vector<double> v = { 0.0, 0.0, 0.0 }; // Geschwindigkeit des Dings

        // Konstruktor mit drei Argumenten zum Initialisieren
        Ding(double t_, vector<double> r_, vector<double> v_) : t{t_}, r{r_}, v{v_} { }
};

// Definition der Klasse 'Pendel' (hier nur eindimensionales (x,y), beschrieben durch Winkel phi, Aufhaengepunkt bei (0,0,0))
class Pendel : public Ding {
    // Private Instanzvariablen (Daten-Member) der Klasse
    vector<double> phi;    // Deklaration eines double Vektors zum Speichern der Loesung fuer u_1
    vector<double> v_phi;  // Deklaration eines double Vektors zum Speichern der Loesung fuer u_2
    vector<double> Zeit;   // Deklaration eines double Vektors zum Speichern der Zeit-Werte
    unsigned N_sim = 2500; // Anzahl der Gitter-Zeitpunkte der Simulation (Dimension der folgenden Loesungsvektoren)
    unsigned index = 0;    // Anzahl der schon berechneten Loesungswerte

    double k1_1,k2_1,k3_1,k4_1; // Deklaration der vier Runge-Kutta Parameter fuer u_1
    double k1_2,k2_2,k3_2,k4_2; // Deklaration der vier Runge-Kutta Parameter fuer u_2

    public:
        // Oeffentliche Instanzvariablen (Daten-Member) der Klasse
        double l = 1.0;       // Laenge des Pendels
        double beta = 0.0;    // Reibungsparameter

        // Konstruktor erzeugt ein Objekt Ding und initialisiert die Laenge l des Pendels und den Reibungskoeffizienten beta
        Pendel(double t_, double phi_, double v_phi_, double l_, double beta_, unsigned N_sim_) :
        Ding( t_,  { l_ * sin(phi_), - l_ * cos(phi_), 0 },  { l_ * cos(phi_) * v_phi_, l_ * sin(phi_) * v_phi_, 0 }), l{l_}, beta{beta_}, N_sim{N_sim_}{
            Zeit.resize( N_sim + 2 );         // Die Anzahl der Eintraege im Vektor Zeit wird auf N+2 erhoeht
            phi.resize( N_sim + 2 );          // Die Anzahl der Eintraege im Vektor phi wird auf N+2 erhoeht
            v_phi.resize( N_sim + 2 );        // Die Anzahl der Eintraege im Vektor v_phi wird auf N+2 erhoeht

            Zeit[0] = t_;                     // Zum Zeit-Vektor die Endzeit eintragen
            phi[0] = phi_;                    // Zum y-Vektor den Anfangswert alpha_1=phi(a) eintragen
            v_phi[0] = v_phi_;                // Zum y'-Vektor den Anfangswert alpha_2=v_phi(a) eintragen
        }
```

```cpp
// Definition der Klasse 'Pendel' (hier nur eindimensionales (x,y), beschrieben durch Winkel phi, Aufhaengepunkt bei (0,0,0))
class Pendel : public Ding {
    // Private Instanzvariablen (Daten-Member) der Klasse
    vector<double> phi;    // Deklaration eines double Vektors zum Speichern der Loesung fuer u_1
    vector<double> v_phi;  // Deklaration eines double Vektors zum Speichern der Loesung fuer u_2
    vector<double> Zeit;   // Deklaration eines double Vektors zum Speichern der Zeit-Werte
    unsigned N_sim = 2500; // Anzahl der Gitter-Zeitpunkte der Simulation (Dimension der folgenden Loesungsvektoren)
    unsigned index = 0;    // Anzahl der schon berechneten Loesungswerte

    double k1_1,k2_1,k3_1,k4_1; // Deklaration der vier Runge-Kutta Parameter fuer u_1
    double k1_2,k2_2,k3_2,k4_2; // Deklaration der vier Runge-Kutta Parameter fuer u_2

    public:
        // Oeffentliche Instanzvariablen (Daten-Member) der Klasse
        double l = 1.0;       // Laenge des Pendels
        double beta = 0.0;    // Reibungsparameter

        // Konstruktor erzeugt ein Objekt Ding und initialisiert die Laenge l des Pendels und den Reibungskoeffizienten beta
        Pendel(double t_, double phi_, double v_phi_, double l_, double beta_, unsigned N_sim_) :
        Ding( t_,  { l_ * sin(phi_), - l_ * cos(phi_), 0 },  { l_ * cos(phi_) * v_phi_, l_ * sin(phi_) * v_phi_, 0 }), l{l_}, beta{beta_}, N_sim{N_sim_}{
            Zeit.resize( N_sim + 2 );        // Die Anzahl der Eintraege im Vektor Zeit wird auf N+2 erhoeht
            phi.resize( N_sim + 2 );         // Die Anzahl der Eintraege im Vektor phi wird auf N+2 erhoeht
            v_phi.resize( N_sim + 2 );       // Die Anzahl der Eintraege im Vektor v_phi wird auf N+2 erhoeht

            Zeit[0] = t_;                    // Zum Zeit-Vektor die Endzeit eintragen
            phi[0] = phi_;                   // Zum y-Vektor den Anfangswert alpha_1=phi(a) eintragen
            v_phi[0] = v_phi_;               // Zum y'-Vektor den Anfangswert alpha_2=v_phi(a) eintragen
        }

        double f_1(double t, double u_1, double u_2){      // Deklaration und Definition der Funktion f_1(t,u_1,u_2)
            double wert;
            wert = u_2;                                     // Eigentliche Definition der Funktion
            return wert;                                    // Rueckgabewert der Funktion f_1
        }                                                   // Ende der Funktion f_1

        virtual double f_2(double t, double u_1, double u_2){ // Deklaration und Definition der Funktion f_2(t,u_1,u_2)
            double wert;                                    // virtuell, da diese Funktion in weiteren abgeleiteten Klassen ueberschieben werden wird
            wert = - 9.81 / l * sin(u_1) - beta * u_2;      // DGl des physikalischen Pendels mit Reibung (Stokesscher Ansatz)
            return wert;                                    // Rueckgabewert der Funktion f_2
        }                                                   // Ende der Funktion f_2

        void Gehe_Zeitschritt(double dt, int N){            // Einen Zeitschritt um dt weiter mittels Runge-Kutta
            double t_;
            double a = Zeit[index];    // Letzter berechneter Zeitpunkt
            double h = dt / N;         // Abstand dt zwischen den aequidistanten Punkten des t-Intervalls (h=dt)
            double u_1 = phi[index];   // Definition der lokalen Variable u_1=y und Initialisierung mit dem letzten berechneten Wert
            double u_2 = v_phi[index]; // Definition der lokalen Variable u_2=y' und Initialisierung mit dem letzten berechneten Wert

            for(int i=0; i <= N; ++i){                      // for-Schleife ueber die einzelnen Punkte des t-Intervalls
                t_ = a + i*h;                               // Zeit-Parameter wird um h erhoeht

                k1_1 = h*f_1(t_,u_1,u_2);                   // Runge-Kutta Parameter k1 fuer u_1
                k1_2 = h*f_2(t_,u_1,u_2);                   // Runge-Kutta Parameter k1 fuer u_2
                k2_1 = h*f_1(t_+h/2,u_1+k1_1/2,u_2+k1_2/2); // Runge-Kutta Parameter k2 fuer u_1
                k2_2 = h*f_2(t_+h/2,u_1+k1_1/2,u_2+k1_2/2); // Runge-Kutta Parameter k2 fuer u_2
                k3_1 = h*f_1(t_+h/2,u_1+k2_1/2,u_2+k2_2/2); // Runge-Kutta Parameter k3 fuer u_1
                k3_2 = h*f_2(t_+h/2,u_1+k2_1/2,u_2+k2_2/2); // Runge-Kutta Parameter k3 fuer u_2
                k4_1 = h*f_1(t_+h,u_1+k3_1,u_2+k3_2);       // Runge-Kutta Parameter k4 fuer u_1
                k4_2 = h*f_2(t_+h,u_1+k3_1,u_2+k3_2);       // Runge-Kutta Parameter k4 fuer u_2
                u_1 = u_1 + (k1_1 + 2*k2_1 + 2*k3_1 + k4_1)/6;
                u_2 = u_2 + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6;
            }                                               // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls

            Zeit[index+1] = t_;                             // Zum Zeit-Vektor den neuen Wert eintragen
            phi[index+1] = u_1;                             // Zum y-Vektor den neuen Wert eintragen
            v_phi[index+1] = u_2;                           // Zum y'-Vektor den neuen Wert eintragen
            index++;                                        // index um eins erhoehen

            t = t_;                                         // Neuen Zeit-Wert des Pendels in der Klasse Ding aktualisieren
            r = { l * sin(u_1), - l * cos(u_1), 0 };        // Neuen Ort Wert des Pendels in der Klasse Ding aktualisieren
            v = { l * cos(u_1) * u_2, l * sin(u_1) * u_2, 0 }; // Neuen Geschwindigkeitswert des Pendels in der Klasse Ding aktualisieren
        }                                                   // Ende der Gehe_Zeitschritt Funktion
};                                                          // Ende der Klasse 'Pendel'
```

Die Vererbung von Klassenmerkmalen kann in Form einer *Schnittstellenvererbung* oder/und *Implementierungsvererbung* geschehen.

Die Sub-Subklasse ‚Pendel_math' stellt hierbei eine typische *Implementierungsvererbung* dar.

```
//Definition der Klasse 'Pendel_math' (mathematische Pendel, harmonischer Oszillator) als abgeleitete Klasse der Klasse Pendel
class Pendel_math : public Pendel {
    public:
        // Konstruktor erzeugt ein Objekt 'Pendel' und implizit auch ein Objekt Ding`
        Pendel_math(double t_, double phi_, double v_phi_, double l_, double beta_, unsigned N_sim_) : Pendel(t_, phi_, v_phi_, l_, beta_, N_sim_){}

            double f_2(double t, double u_1, double u_2){ // Ueberschreiben der virtuellen Funktion aus der Klasse 'Pendel'
                double wert;
                wert = - 9.81 / l * u_1 - beta * u_2;     // DGl des harmonischen Oszillators (omega_0**2 = g/l) mit Reibung (Stokesscher Ansatz)
                return wert;                              // Rueckgabewert der Funktion f_2
            }                                             // Ende der Funktion f_2
};                                                        // Ende der Klasse 'Pendel_math'
```

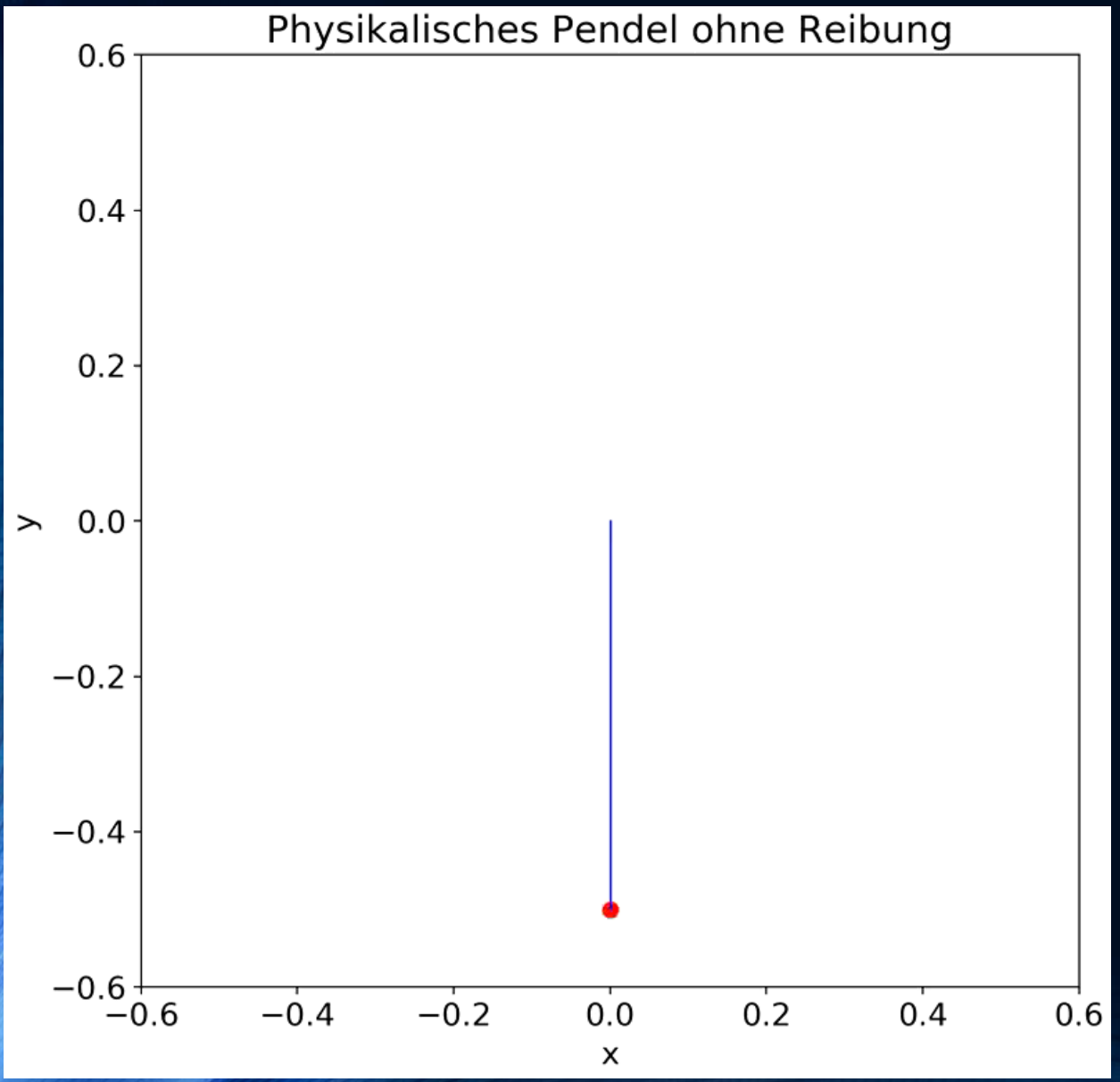
Pendel.cpp

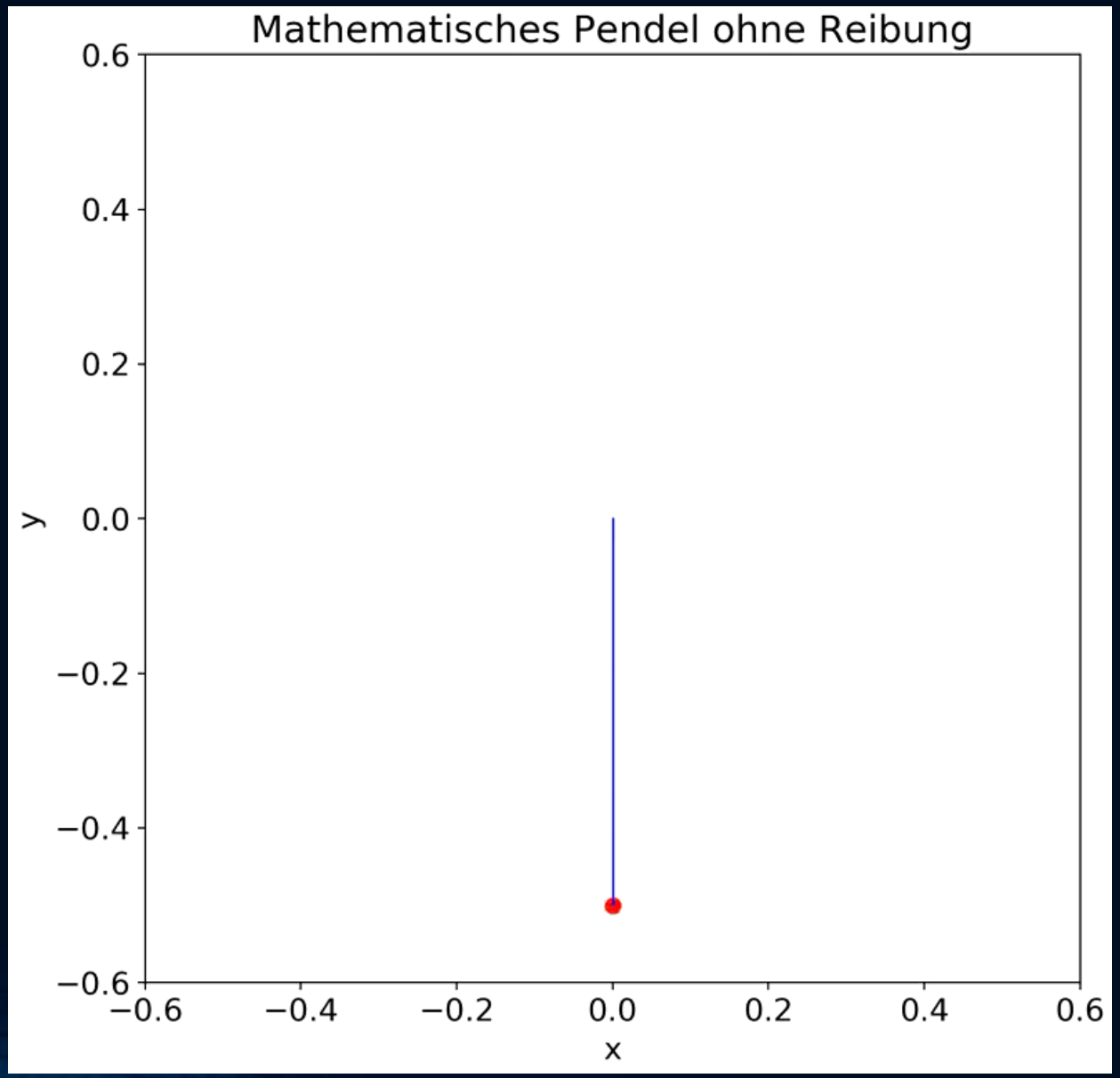
```cpp
/* Beispiel einer, von der Basisklasse 'Ding' abgeleitete Klasse 'Pendel'
 * Zwei zusaetzliche private Instanzvariablen (l und beta) und die winkelspezifischen Anfangswerte (phi und v_phi)
 * wurden der abgeleiteten Pendel-Klasse hinzugefuegt
 * Der Konstruktor der Klasse Pendel erzeugt ein Objekt Ding und initialisiert seine eigenen Daten-Member
 * Von der Klasse Pendel wurde eine weitere Subklasse 'Pendel_math' abgeleitet, welche die lineare Näherung der Pendel-DGL benutzt (gedaempfter harmonischer Ozillator
*/
#include "Pendel.hpp"          // Pendel und Ding Klassen
#include <iostream>            // Ein- und Ausgabebibliothek

int main(){                    // Hauptfunktion
    double a = 0.0;            // Untergrenze des Zeit-Intervalls
    double b = 5.0;           // Untergrenze des Zeit-Intervalls
    int N_RK = 500;            // Anzahl der Gitter-Zeitpunkte des Runge-Kutta Ordnung vier Verfahrens
    int N_sim = 1000;          // Anzahl der Gitter-Zeitpunkte der Simulation (Anzahl der ausgegebenen Punkte)
    double dt = (b - a)/N_sim; // Abstand dt zwischen den aequidistanten Punkten des Sim-Intervalls (h=dt)

    Pendel Pendel_A = Pendel(a, 0.0, 8.2, 0.5, 0.0, N_sim);          // Konstruktor bilded eine Instanz der Subklasse Pendel und erzeugt implizit ein Objekt Ding
    Pendel_math Pendel_B = Pendel_math(a, 0.0, 8.2, 0.5, 0.0, N_sim); // Konstruktor bilded eine Instanz der Sub-Subklasse Pendel_math

    printf("# 0: Index i \n# 1: t-Wert \n");                          // Beschreibung der ausgegebenen Groessen
    printf("# 2: x(t) , physikalisches Pendel \n# 3: y(t) , physikalisches Pendel \n");
    printf("# 4: v_x(t) , physikalisches Pendel \n# 5: v_y(t) , physikalisches Pendel \n");
    printf("# 6: x(t) , mathematisches Pendel \n# 7: y(t) , mathematisches Pendel \n");
    printf("# 8: v_x(t) , mathematisches Pendel \n# 9: v_y(t) , mathematisches Pendel \n");
    printf("%5d %19.15f %19.15f %19.15f %19.15f %19.15f ",0, Pendel_A.t, Pendel_A.r[0], Pendel_A.r[1], Pendel_A.v[0], Pendel_A.v[1]);
    printf("%19.15f %19.15f %19.15f %19.15f \n",Pendel_B.r[0], Pendel_B.r[1], Pendel_B.v[0], Pendel_B.v[1]);

    // Terminalausgabe von Ort und Geschwindigkeit des Pendels
    for(int i=1; i  <= N_sim; ++i){                                  // for-Schleife zur Terminalausgabe der Loesung (Zeitgitterpunkte)
        Pendel_A.Gehe_Zeitschritt(dt, N_RK);                         // Aufruf der Methode Gehe_Zeitschritt(dt, N) der Klasse Pendel
        Pendel_B.Gehe_Zeitschritt(dt, N_RK);                         // Aufruf der Methode Gehe_Zeitschritt(dt, N) der Klasse Pendel
        printf("%5d %19.15f %19.15f %19.15f %19.15f %19.15f ",i, Pendel_A.t, Pendel_A.r[0], Pendel_A.r[1], Pendel_A.v[0], Pendel_A.v[1]);
        printf("%19.15f %19.15f %19.15f %19.15f \n",Pendel_B.r[0], Pendel_B.r[1], Pendel_B.v[0], Pendel_B.v[1]);
    }                                                                // Ende for-Schleife (Zeitgitterpunkte)
}                                                                    // Ende main()-Funktion
```

Pendel.py

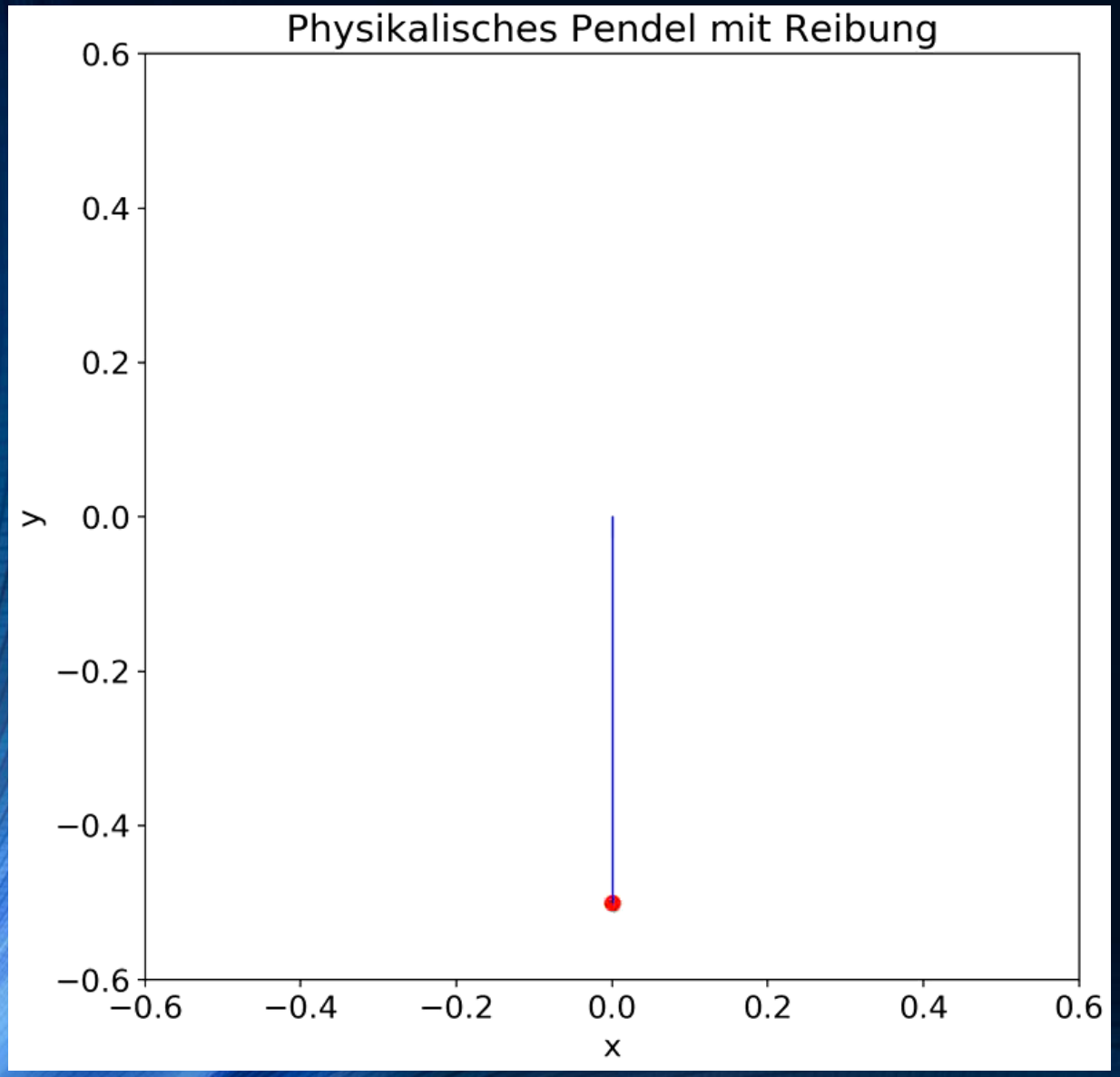
```
# Python Programm zum Visualisieren der Daten des Pendel.cpp Programms,
# Mathematisches vs. physikalisches Pendel ohne Reibung
# Es werden hier mehrere Bilder der zeitlichen Entwicklung des Systems in einem Ordner 'Bilder' gespeichert
# !!!! Sie muessen vor der Ausfuehrung des Programms den Ordner Bilder erstellen !!!!
# Die einzelnen Bilder kann mann dann mittels des folgenden Terminalbefehls zu einem Video binden:
# ffmpeg -framerate 200 -i './Pendel_%04d.jpg' -s 800x800  Pendel.mp4
# (die Framerate ergibt sich durch N_sim/b = 1000/5)

import matplotlib
import matplotlib.pyplot as plt         # Python Bibliothek zum Plotten (siehe https://matplotlib.org/ )
import numpy as np                      # Python Bibliothek fuer Mathematisches (siehe https://numpy.org/ )

import matplotlib.gridspec as gridspec
params = {
    'figure.figsize'    : [8,8],
#    'text.usetex'       : True,
    'axes.titlesize' : 18,
    'axes.labelsize' : 15,
    'xtick.labelsize' : 15 ,
    'ytick.labelsize' : 15
}
matplotlib.rcParams.update(params)

data = np.genfromtxt("./Pendel.dat")  # Einlesen der berechneten Daten von Pendel.cpp

for it in range(len(data[:,0])):      # for-Schleife fuer die zeitliche Entwicklung der Dinge in der Kiste
    print(it)                         # Terminalausgabe der Erstellung des i-ten Bildes
    plt.cla()
    plt.title(r'Mathematisches Pendel ohne Reibung') # Titel der Abbildung
#    plt.title(r'Physikalisches Pendel ohne Reibung') # Titel der Abbildung
    plt.xlabel('x')                                  # Beschriftung x-Achse
    plt.ylabel('y')                                  # Beschriftung y-Achse
    plt.xlim(-0.6, 0.6)                              # Limitierung der x-Achse
    plt.ylim(-0.6, 0.6)                              # Limitierung der y-Achse

    plt.scatter( data[it,6], data[it,7], s=50, marker='o', c="red")      # Zeichnen des Pendel-Koerpers
    plt.plot( [0 , data[it,6]] , [0 , data[it,7]] ,c="blue",linewidth=1) # Zeichnen der Pendel-Stange
#    plt.scatter( data[it,2], data[it,3], s=50, marker='o', c="red")      # Zeichnen des Pendel-Koerpers
#    plt.plot( [0 , data[it,2]] , [0 , data[it,3]] ,c="blue",linewidth=1) # Zeichnen der Pendel-Stange

    # Bild-Ausgabe mit Speicherung eines individuellen Iteration-Namens
    pic_name = "./Bilder/" + "Pendel_" + "{:0>4d}".format(it) + ".jpg"
    plt.savefig(pic_name, dpi=200,bbox_inches="tight",pad_inches=0.05,format="jpg")
    plt.close()
```

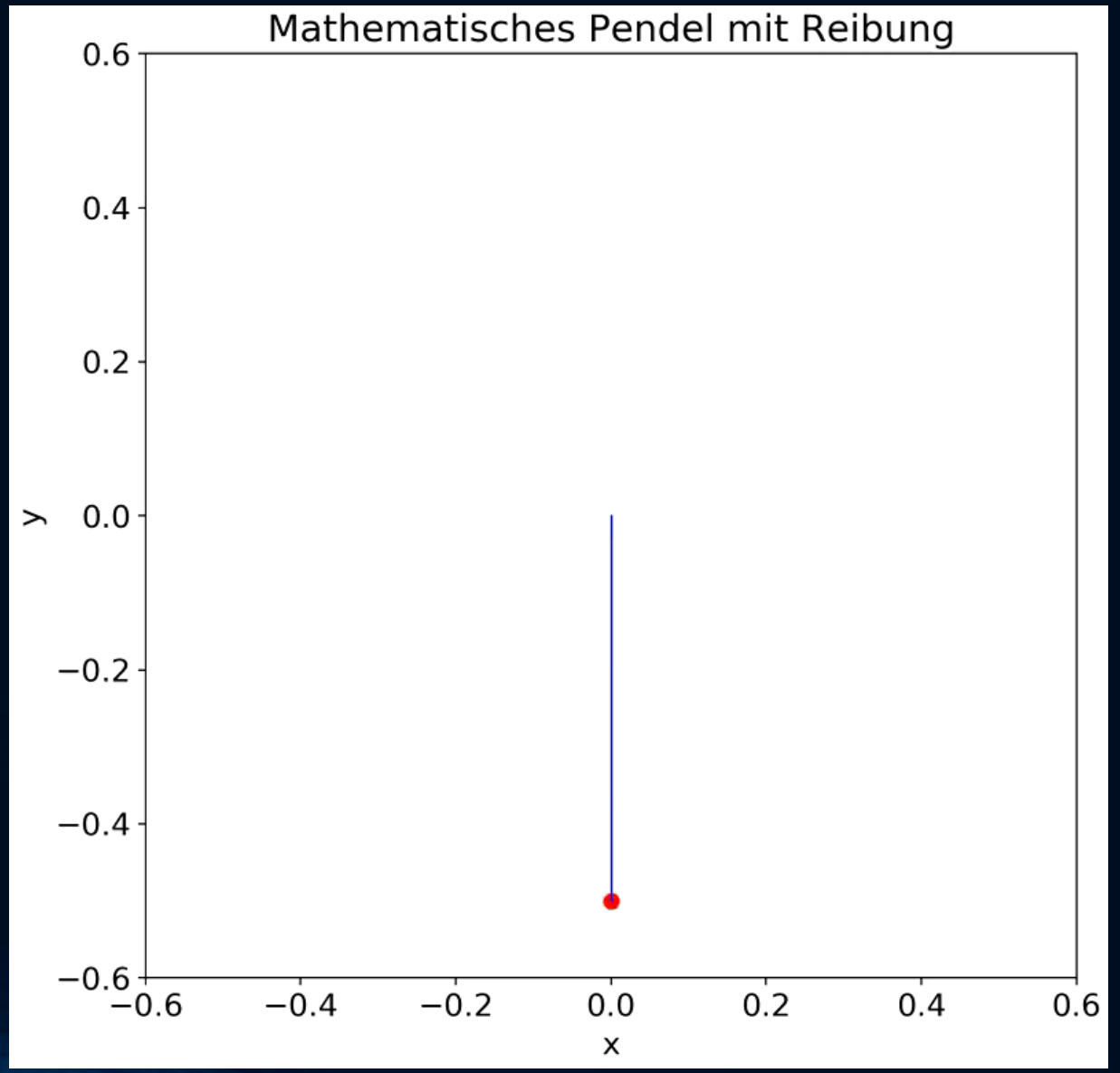
Die abgebildeten Animationen visualisieren die Bewegung des physikalischen und mathematischen Pendels ohne Reibung.





Die abgebildeten Animationen visualisieren die Bewegung des physikalischen und mathematischen Pendels mit Reibung.





Mehrfachvererbung und Klassenhierarchien

In dem vorigen Unterpunkt wurde in das C++ Konstrukt der abgeleiteten Klasse eingeführt. Besitzt eine abgeleitete Klasse gleichzeitig mehrere Oberklassen, so spricht man von einer *Mehrfachvererbung*. Die C++ Struktur einer solchen *Mehrfachvererbung* besitzt formal das folgende Aussehen:

```
class Base_1 { 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };

class Base_2 { 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };

class Sub : public Base_1, public Base_2{ 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };
```

Die abgeleitete Klasse mit dem Namen 'Sub' erbt hierbei die Klassenmerkmale von der Basisklassen 'Base_1' und 'Base_1'. In umfangreichen C++ Programmen baut sich somit eine Klassenhierarchie bestehend aus Basis-, Sub- und Sub-Subklassen auf.