Allgemeine Relativitätstheorie mit dem Computer

(General Theory of Relativity on the Computer)

Vorlesung gehalten an der

J.W.Goethe-Universität in Frankfurt am Main

(Sommersemester 2020)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 15.05.2020

Erster Vorlesungsteil: Allgemeine Relativitätstheorie mit Python

Bewegung eines Probekörpers um ein schwarzes Loch in der Ebene

Im folgenden wird die Geodätengleichung in vorgegebener Schwarzschild Raumzeit betrachtet. Die Geodätengleichung beschreibt wie sich ein Probekörper (Masse = 0) im Raum bewegt und sagt voraus, dass diese Bewegung sich stehts entlang der kürzesten Kurve, in der durch die Metrik beschriebenen gekrümmten Raumzeit, vollzieht. Zunächst wird das Python Packet "GraviPy" eingebunden, welches auf dem Packet SymPy basiert und symbolische Berechnungen in der Allgemeinen Relativitätstheorie relativ einfach möglich macht.

In [2]:

```
from gravipy.tensorial import *
import sympy as sym
import inspect
import numpy as np
import math
sym.init_printing()
```

```
Definition der Koordinaten und der kovarianten Raumzeit-Metrik der
Schwarzschildmetrik:
$$ g_{\mu\nu}=\left( \begin{array}{ccc} 1-\frac{2\,M}{r} & 0 & 0 & 0\\ 0&  -
\frac{1}{1-\frac{2\,M}{r}}&  0&0 \\ 0&  0&  -r^2& 0\\ 0&  0&  0& -r^2
\hbox{sin}^2(\theta)\end{array} \right)$$
```

In [3]:

```
# define some symbolic variables
t, r, theta, phi, M = symbols('t, r, \\theta, \phi, M')
# create a coordinate four-vector object instantiating
# the Coordinates class
x = Coordinates('x', [t, r, theta, phi])
# define a matrix of a metric tensor components
Metric = diag((1-2*M/r), -1/(1-2*M/r), -r**2, -r**2*sin(theta)**2)
#Metric = diag(A, -B, -r**2, -r**2*sin(theta)**2)
# create a metric tensor object instantiating the MetricTensor class
g = MetricTensor('g', x, Metric)
```

In [4]:

```
g(All, All)
```

Out[4]:

$$\begin{bmatrix} -\frac{2M}{r}+1 & 0 & 0 & 0 \\ 0 & -\frac{1}{-\frac{2M}{r}+1} & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2\sin^2(\theta) \end{bmatrix}$$

Kontravariante Form der Metrik ($g^{\mu\nu}$)

In [5]:

```
g(-All, -All)
```

Out[5]:

$$\begin{bmatrix} \frac{1}{-\frac{2M}{r}+1} & 0 & 0 & 0 \\ 0 & \frac{2M}{r}-1 & 0 & 0 \\ 0 & 0 & -\frac{1}{r^2} & 0 \\ 0 & 0 & 0 & -\frac{1}{r^2\sin^2(\theta)} \end{bmatrix}$$

Die Chistoffel Symbole in (kontravarianter Form):

$$\Gamma_{\rho\mu\nu} = g_{\rho\sigma}\Gamma^{\sigma}_{\mu\nu} = \frac{1}{2}(g_{\rho\mu|\nu} + g_{\rho\nu|\mu} - g_{\mu\nu|\rho})$$

Hier speziell

$$\Gamma_{222} = \Gamma_{rrr}$$

In [6]:

```
Ga = Christoffel('Ga', g)
Ga(2, 2, 2)
```

Out[6]:

$$\frac{M}{(2M-r)^2}$$

Der Riemann Tensor:

$$R_{\mu\nu\rho\sigma} = \frac{\partial\Gamma_{\mu\nu\sigma}}{\partial x^\rho} - \frac{\partial\Gamma_{\mu\nu\rho}}{\partial x^\sigma} + \Gamma^\alpha_{\nu\sigma}\Gamma_{\mu\rho\alpha} - \Gamma^\alpha_{\nu\rho}\Gamma_{\mu\sigma\alpha} - \frac{\partial g_{\mu\alpha}}{\partial x^\rho}\Gamma^\alpha_{\nu\sigma} + \frac{\partial g_{\mu\alpha}}{\partial x^\sigma}\Gamma^\alpha_{\nu\rho}$$

Hier speziell

$$R_{1313} = R_{t\theta t\theta}$$

In [7]:

```
Rm = Riemann('Rm', g)
Rm(1,3,1,3)
```

Out[7]:

$$\frac{M\,(2M - r)}{r^2}$$

Oder in gemischt kontra- kovarianter Form

$$R^1{}_{313} = R^t{}_{\theta t\theta}$$

In [8]:

```
Rm(-1,3,1,3)
```

Out[8]:

$$\frac{M\,(2M - r)}{r\,(-2M + r)}$$

Der Ricci Tensor:

$$R_{\mu\nu} = \frac{\partial\Gamma^\sigma_{\mu\nu}}{\partial x^\sigma} - \frac{\partial\Gamma^\sigma_{\mu\sigma}}{\partial x^\nu} + \Gamma^\sigma_{\mu\nu}\Gamma^\rho_{\sigma\rho} - \Gamma^\rho_{\mu\sigma}\Gamma^\sigma_{\nu\rho}$$

In [9]:

```
Ri = Ricci('Ri', g)
Ri(All, All)
```

Out[9]:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Der Ricci Tensor lässt sich auch durch folgende Kontraktion aus dem Riemann Tensor berechnen:
$R_{\mu\nu} = R^\rho_{\mu\rho\nu}$

In [10]:

```
ricci = sum([Rm(i, All, k, All)*g(-i, -k)
             for i, k in list(variations(range(1, 5), 2, True))],
            zeros(4))
ricci.simplify()
ricci
```

Out[10]:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Der Ricci Skalar ergibt sich aus der Kontraktion des Ricci Tensors: $R = R^{\mu}_{\mu} = g^{\mu\nu} R_{\mu\nu}$

In [11]:

```
Ri.scalar()
```

Out[11]:

$$0$$

Der Einstein Tensor:

$$G_{\mu\nu} = R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R$$

In [12]:

```
G = Einstein('G', Ri)
G(All, All)
```

Out[12]:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Geodätische Bewegung eines Probekörpers

Die Geodätengleichunglässt sich durch folgendes Variationsprinzip herleiten:

$$\int_A^B ds = \int_A^B \sqrt{g_{\mu\nu} dx^{\mu} dx^{\nu}} = \int_A^B \sqrt{g_{\mu\nu} \frac{dx^{\mu}}{d\lambda} \frac{dx^{\nu}}{d\lambda}} d\lambda \rightarrow \text{Extremal}$$

, wobei sich dann die Geodätengleichung mittels der Euler-Lagrange Gleichungen $L = \sqrt{g_{\mu\nu} \frac{dx^{\mu}}{d\lambda} \frac{dx^{\nu}}{d\lambda}}$, bzw. alternativ $L = g_{\mu\nu} \frac{dx^{\mu}}{d\lambda} \frac{dx^{\nu}}{d\lambda}$ ergibt:

$$\frac{d}{d\lambda} \left(\frac{\partial L}{\partial \frac{\partial x^{\mu}}{\partial \lambda}} \right) - \frac{\partial L}{\partial x^{\mu}} = 0 \quad \rightarrow \quad \frac{d^2 x^{\mu}}{d\lambda^2} + \Gamma^{\mu}_{\nu\rho} \frac{dx^{\nu}}{d\lambda} \frac{dx^{\rho}}{d\lambda} = 0$$

$\Gamma^{\mu}_{\nu\rho}$ sind die Christoffel Symbole zweiter Art und λ ein affiner Parameter (z.B. die Eigenzeit τ des Probekörpers). Die Geodätengleichung ist ein System gekoppelter Differentialgleichungen und sie lässt sich in Python mit der Funktion "Geodesic" berechnen (in vollständig kovarianter Form):

$$\frac{d^2x_\mu}{d\tau^2} + \Gamma_{\rho\sigma\mu}\frac{dx^\rho}{d\tau}\frac{dx^\sigma}{d\tau} = 0$$

In [13]:

```
tau = Symbol('\\tau')
w = Geodesic('w', g, tau)
w(All).transpose()
```

Out[13]:

$$\begin{bmatrix} \frac{2M\frac{d}{d\tau}r(\tau)\frac{d}{d\tau}t(\tau)}{r^2(\tau)} + \left(-\frac{2M}{r(\tau)}+1\right)\frac{d^2}{d\tau^2}t(\tau) \\ -\frac{M\left(\frac{d}{d\tau}t(\tau)\right)^2}{r^2(\tau)} + \frac{M\left(\frac{d}{d\tau}r(\tau)\right)^2}{\left(-\frac{2M}{r(\tau)}+1\right)^2 r^2(\tau)} + r(\tau)\sin^2(\theta(\tau))\left(\frac{d}{d\tau}\phi(\tau)\right)^2 + r(\tau)\left(\frac{d}{d\tau}\theta(\tau)\right)^2 - \\ r^2(\tau)\sin(\theta(\tau))\cos(\theta(\tau))\left(\frac{d}{d\tau}\phi(\tau)\right)^2 - r^2(\tau)\frac{d^2}{d\tau^2}\theta(\tau) - 2r(\tau)\frac{d}{d\tau}\theta(\tau)\frac{d}{d\tau}r(\tau \\ -r^2(\tau)\sin^2(\theta(\tau))\frac{d^2}{d\tau^2}\phi(\tau) - 2r^2(\tau)\sin(\theta(\tau))\cos(\theta(\tau))\frac{d}{d\tau}\phi(\tau)\frac{d}{d\tau}\theta(\tau) - 2r(\tau)\sin^2(\theta(\tau) \end{bmatrix}$$

Die definierte Python-Klasse '*Geodesic*' setzt die Koordinaten automatisch in einen Parametrisierungsmodus (Parametrisierung der Koordinaten nach dem affinen Parameter τ).

In [14]:

```
Parametrization.info()
```

Out[14]:

$$[[x, \quad \tau]]$$

In [15]:

```
x(-All)
```

Out[15]:

$$\begin{bmatrix} t(\tau) & r(\tau) & \theta(\tau) & \phi(\tau) \end{bmatrix}$$

Diese Parametrisierung kann man mit dem folgenden Befehl wieder abstellen:

In [16]:

```
#Parametrization.deactivate(x)
```

Wir lassen nur ebene Bewegungen zu ($\theta = \frac{\pi}{2}, \frac{d\theta}{d\tau} = 0, \frac{d^2\theta}{d\tau^2} = 0$).

In [17]:

```
DGL = w(All).subs([(x(-3).diff(tau,tau),0),(x(-3).diff(tau),0),(x(-3),math.pi/2)]).
DGL
```

Out[17]:

$$\begin{bmatrix} \frac{2M\frac{d}{d\tau}r(\tau)\frac{d}{d\tau}t(\tau)}{r^2(\tau)} + \left(-\frac{2M}{r(\tau)}+1\right)\frac{d^2}{d\tau^2}t(\tau) \\ -\frac{M\left(\frac{d}{d\tau}t(\tau)\right)^2}{r^2(\tau)} + \frac{M\left(\frac{d}{d\tau}r(\tau)\right)^2}{\left(-\frac{2M}{r(\tau)}+1\right)^2 r^2(\tau)} + 1.0r(\tau)\left(\frac{d}{d\tau}\phi(\tau)\right)^2 - \frac{\frac{d^2}{d\tau^2}r(\tau)}{-\frac{2M}{r(\tau)}+1} \\ 6.12323399573677\cdot 10^{-17} r^2(\tau)\left(\frac{d}{d\tau}\phi(\tau)\right)^2 \\ -1.0r^2(\tau)\frac{d^2}{d\tau^2}\phi(\tau) - 2.0r(\tau)\frac{d}{d\tau}\phi(\tau)\frac{d}{d\tau}r(\tau) \end{bmatrix}$$

Zusätzlich kann man sich auf den radial in das schwarze Loch einfallenden Probekörper beschränken ($\frac{d\phi}{d\tau}=0, \frac{d^2\phi}{d\tau^2}=0$).

In [18]:

```
DGL1 = DGL.subs([(x(-4).diff(tau,tau),0),(x(-4).diff(tau),0)])
DGL1
```

Out[18]:

$$\begin{bmatrix} \frac{2M\frac{d}{d\tau}r(\tau)\frac{d}{d\tau}t(\tau)}{r^2(\tau)} + \left(-\frac{2M}{r(\tau)}+1\right)\frac{d^2}{d\tau^2}t(\tau) \\ -\frac{M\left(\frac{d}{d\tau}t(\tau)\right)^2}{r^2(\tau)} + \frac{M\left(\frac{d}{d\tau}r(\tau)\right)^2}{\left(-\frac{2M}{r(\tau)}+1\right)^2 r^2(\tau)} - \frac{\frac{d^2}{d\tau^2}r(\tau)}{-\frac{2M}{r(\tau)}+1} \\ 0 \\ 0 \end{bmatrix}$$

Dieses System von Differentialgleichungen ist zweiter Ordnung in der Eigenzeit τ muss nun gelösst werden. Zunächst eine kurze Zusammenfassung: Lösen von Differentialgleichungen in Python

Lösen von Differentialgleichungen in Python

In Python kann man Differentialgleichungen z.B. mit dem Befehl dsolve lösen (falls eine analytische Lösung existiert). Hier ein einfaches Beispiel:

In [19]:

```
BeispielDGL1=sym.Eq(x(-1).diff(tau,tau), -3*x(-1))
BeispielDGL1
```

Out[19]:

$$\frac{d^2}{d\tau^2}t(\tau) = -3t(\tau)$$

Lösung ohne Anfangsbedingungen:

In [20]:

```
sym.dsolve(BeispielDGL1)
```

Out[20]:

$$t(\tau) = C_1 \sin\left(\sqrt{3}\tau\right) + C_2 \cos\left(\sqrt{3}\tau\right)$$

Lösung mit Anfangsbedingungen:

In [21]:

```
LoesAnalytic = sym.dsolve(BeispielDGL1,x(-1),ics={x(-1).subs(tau,0):3,x(-1).diff(ta
LoesAnalytic
```

Out[21]:

$$t(\tau) = \frac{2\sqrt{3}\sin\left(\sqrt{3}\tau\right)}{3} + 3\cos\left(\sqrt{3}\tau\right)$$

Diese Lösung kann man sich z.B. wie folgt darstellen:

In [22]:

```
import matplotlib.pyplot as plt
import matplotlib

tauvals = np.linspace(0, 10, 1000)
func = sym.lambdify(tau, LoesAnalytic.rhs)
plt.plot(tauvals,func(tauvals),color='blue', linewidth=1, linestyle='-')
```

Out[22]:

```
[<matplotlib.lines.Line2D at 0x7f67f8a5f518>]
```

Manchmal existiert keine analytische Lösung für die vorliegende Differentialgleichungen. In Python kann man Differentialgleichungen z.B. mit dem Befehl "solve_ivp" lösen. Wir demonstrieren die Vorgehensweise zunächst an dem folgenden Beispiel eines Systems von Differntialgleichungen, welches eine Kreisbahn beschreibt:

$$\frac{d}{dt}\vec{r} = \begin{pmatrix} \frac{dy_0(t)}{dt} \\ \frac{dy_1(t)}{dt} \end{pmatrix} = \begin{pmatrix} \cos(y_1(t)) \\ \sin(y_0(t)) \end{pmatrix}$$

Zunächst importieren wir scipy, definieren unser Systems von Differntialgleichungen und lösen dieses dann mit den Anfangswerten $\vec{r}(t=0) = [y_1(t=0) = 0, y_2(t=0) = 1]$ im Bereich $t \in [0, 6]$ mit 100 Punkten:

In [23]:

```
from scipy.integrate import solve_ivp
```

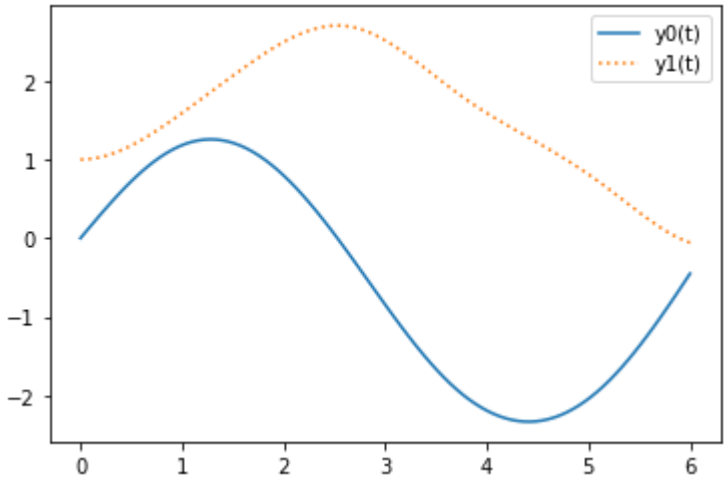
In [30]:

```
tspan=np.linspace(0,6,100)
def f(t, r):
    y0, y1 = r
    dy0dt = np.cos(y1)
    dy1dt = np.sin(y0)
    return dy0dt, dy1dt
sol = solve_ivp(f, (0,6), (0,1), t_eval=tspan)
```

Die Lösung können wir uns grafisch als zeitliche Funktionen von y_0 und y_1 darstellen

In [27]:

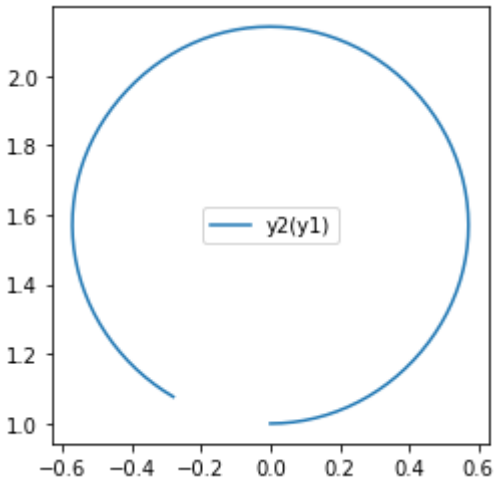
```
plt.clf()
fig, ax = plt.subplots(num=1)
ax.plot(tspan, sol.y[0], '-', label='y0(t)')
ax.plot(tspan, sol.y[1], ':', label='y1(t)')
ax.legend(loc='best')
plt.show()
```



... oder als Raumkurve $\vec{r}(t)$ in einem y_0-y_1 Diagram darstellen:

In [25]:

```
plt.clf()
fig, ax = plt.subplots(num=1)
ax.plot(sol.y[0], sol.y[1], '-', label='y2(y1)')
ax.legend(loc='center')
plt.axis("scaled")
plt.show()
```



Lösen der Geodätengleichung in Python

Um das der Geodätengleichung zugrundeliegende System von Differentialgleichungen zweiter Ordnung in Python zu lösen, muss man es zunächst in ein System erster Ordnung umschreiben und es dann dannach mit solve_ivp lösen -> Hausaufgabe!