

Computational Physics / Numerische Verfahren in der Physik; SoSe 2023

Lorenz von Smekal, Ralf-Arno Tripolt, Robin Kehr, Johannes Roth, Leon Sieke



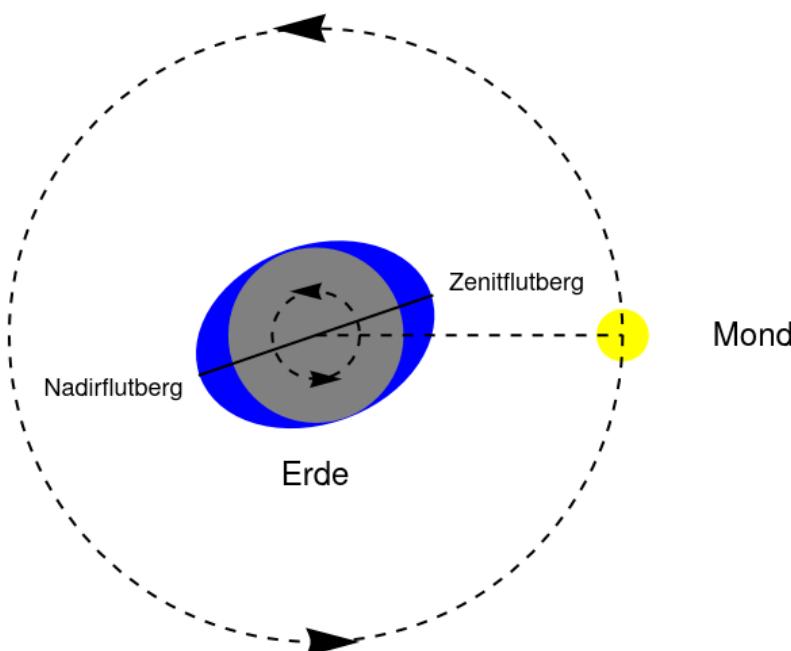
Projekt 3: Gezeitenreibung

Bearbeitet von Finn Bietz, Florian Adamczyk und Finn Wagner

Einführung

In diesem Projekt soll die Gezeitenreibung zwischen Erde und Mond studiert werden. In der folgenden Abbildung ist die Ausprägung zweier Flutberge (der Zenitflutberg ist dem Mond zugewandt, der Nadirflutberg ist dem Mond abgewandt) skizziert. Diese entstehen durch die Anziehungskraft des Mondes und der Zentrifugalkraft, welche durch die Rotation der Erde um den Schwerpunkt des Erde-Mond Systems erzeugt wird. Da die intrinsische Rotationsperiode der Erde wesentlich kürzer ist als die Umlaufperiode des Mondes, dreht sich die Erde unter den Flutbergen hinweg.

Die dabei entstehende Reibungskraft zwischen den Flutbergen und der mit der Erde rotierenden Materie führt zu einer Verschiebung der Flutberge, so dass sie sich nicht, wie man ohne Reibung erwarten würde, auf der Verbindungsline des Erd- und Mondmittelpunktes befinden, sondern in Richtung der intrinsischen Erdrotation verschoben sind. Der Zenitflutberg eilt also der Bewegung des Mondes voraus und beschleunigt durch seine Anziehungskraft die Bahnbewegung des Mondes, der sich infolge dessen auf einer größeren Umlaufbahn bewegt. Die intrinsische Rotation der Erde wiederum wird durch die Reibung mit den Flutbergen abgebremst. Diesen Effekt bezeichnet man als Gezeitenreibung. Er ist Gegenstand dieses Projektes.



Zur Vereinfachung betrachten wir die Gezeitenreibung nur für das Erde-Mond System und vernachlässigen den Einfluss der Sonne. In der folgenden Zelle befinden sich verschiedene astronomische Größen des Erde-Mond Systems, welche während des Projektes benötigt werden.

```
In [ ]: mErde = 5.9721986*10**24      # kg          : Masse der Erde
mOzean = 0.0014*10**24            # kg          : Masse der Ozeane auf der Erde
RErde = 6.3675*10**6              # kg          : Radius der Erde

mMond = 7.3459*10**22            # kg          : Masse des Mondes
rMondBahn = 3.836*10**8           # m          : Bahnradius des Mondes, die Umlaufbahn des Mondes um die Erde kann als Kreisbahn
TMondBahn = 27.32166140*24*3600 # s          : Heutige Periodendauer der Mondumlaufbahn um die Erde

TErdRotation = 86164.100          # s          : Heutige Periodendauer der intrinsischen Erdrotation
tau = 0.0021                      # s/100a     : Zunahme der Tageslänge auf der Erde in 100 Jahren

G = 6.67430*10**(-11)            # m^3/(kg*s^2): Gravitationskonstante
```

Benötigte Pakete und Einstellungen

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
import math
import sympy
from scipy.constants import pi
import scipy
from scipy.integrate._ivp.ivp import OdeResult
from matplotlib.animation import FuncAnimation
from IPython.display import Image

# Beautiful plots
plt.rcParams['text.usetex'] = True
plt.rcParams['font.family'] = "serif"
plt.rcParams['font.serif'] = "Computer Modern Roman"

fastExecution = True    # Executing everything all the time takes very long. If True this skips animations and reduces the tim
```

Konventionen für dieses Notebook

Wir haben uns in unserer Gruppe dazu entschieden, die Markdownzellen (den Text) auf Deutsch zu schreiben, im Code jedoch größtenteils Englisch für Kommentare, Erklärungen und auch Variablennamen zu benutzen da:

- **Einerseits ist Python sowieso schon Englisch, da führt es nur zu Verwirrung im Code ständig die Sprache zu wechseln**
- **Andererseits um die Verständlichkeit auch für andere (nicht deutsche) Leser oder bearbeitende Personen zu verbessern.**
- **Auch wenn das in speziell diesem Projekt evtl. keine so große Rolle spielt ist es trotzdem "good practice".**

\providecommand{\e}[1]{\ensuremath{\cdot 10^{#1}}}

Numerische Integration und Lösen von Anfangswertproblemen

Bevor wir mit der eigentlichen Aufgabe beginnen, müssen wir uns noch mit ein paar numerischen Methoden vertraut machen. Dazu werden wir uns in diesem Abschnitt mit der numerischen Integration und dem Lösen von Anfangswertproblemen beschäftigen. Wir implementieren dazu die Euler-Methode und die Runge-Kutta-Methode 4. Ordnung. Später vergleichen wir die beiden Methoden und vergleichen sie mit der vorgefertigten Funktion `scipy.integrate.solve_ivp`.

Euler-Verfahren

Das explizite Euler-Verfahren, auch eulersches Polygonzugverfahren, wurde von Leonhard Euler 1768 veröffentlicht. \newline

Gegeben sei ein Anfangswertproblem der Form:

$$\frac{dy}{dt} = f(t, y(t)) \quad \text{mit} \quad y(t_0) = y_0$$

Die Differentialgleichungen unseres physikalischen Problems sind gewöhnlich. Man betrachtet nun nur noch diskrete Zeitpunkte $t_k = t_0 + k \cdot h$ mit $k \in \mathbb{N}_0$, wobei h die Diskretisierungsschrittweite ist. \newline

Herleitung der Gleichung: \newline

Wir schreiben zunächst die äquivalente Integralgleichung auf:

$$y(t) = y(t_0) + \int_{t_0}^t f(s, y(s)) \, ds$$

Setzen wir hier nun eine Zeitpunkt t_{k+1} ein, so erhalten wir:

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s)) \, ds$$

Wir nähern nun das Integral, indem wir die Funktion $f(s, y(s))$ durch eine konstante Funktion $f(t_k, y(t_k))$ (wir wählen den Wert am linken Rand des Bereichs) ersetzen.

$$y(t_{k+1}) = y(t_k) + (t_{k+1} - t_k) \cdot f(t_k, y(t_k))$$

Die Differenz $t_{k+1} - t_k$ ist die Diskretisierungsschrittweite h .

$$y(t_{k+1}) = y(t_k) + h \cdot f(t_k, y(t_k)) \quad \text{mit} \quad k \in \mathbb{N}_0$$

\newline Quelle: [Wikipedia: Explizites Euler-Verfahren und Skript Kapitel 7](#)

```
In [ ]: def explicit_euler(fun, t_span, y_0, steps, args=()):
    '''Implementation of the explicit Euler method'''
    # Calculate h, the time step, from the number of steps
    h = (t_span[1]-t_span[0])/steps
    # Create an array of the time steps
    t = np.linspace(t_span[0], t_span[1], steps+1, dtype=np.float64)
    # Create an array for the solution
    y = np.zeros((steps+1, len(y_0)), dtype=np.float64)
    # Set the initial conditions
    y[0] = y_0
    # Iterate over the time steps
    for i in range(steps):
        # Calculate the next step using the explicit Euler method
        y[i+1] = y[i] + h*fun(t[i], y[i], *args)
    return OdeResult(t=t, y=y.T)
```

Explizites Runge-Kutta-Verfahren 4. Ordnung

Das explizite Runge-Kutta 4.Ordnung oder auch das "Klassisches Runge-Kutta-Verfahren" (RK4) ist nach Carl Runge und Martin Kutta benannt. \newline **Gegeben sei ein Anfangswertproblem der Form:**

$$\frac{dy}{dt} = f(t, y(t)) \quad \text{mit} \quad y(t_0) = y_0, \quad y : \mathbb{R} \rightarrow \mathbb{R}^n$$

Es werden vier Zwischenwerte verwendet:

$$\begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f\left(t_k + \frac{h}{2}, y_k + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_k + \frac{h}{2}, y_k + \frac{h}{2}k_2\right) \\ k_4 &= f(t_k + h, y_k + hk_3) \end{aligned}$$

Die Näherung für y_{k+1} lautet dann:

$$y_{k+1} = y_k + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

\newline Quelle: [Wikipedia: Klassisches Runge-Kutta-Verfahren und Skript Vorlesung 7: Kapitel 3.1.2: Runge-Kutta Algorithmus](#)

```
In [ ]: def classic_runge_kutta(fun, t_span, y_0, steps, args=()):
    '''Implementation of the classic Runge-Kutta method'''
    # Calculate h, the time step, from the number of steps
    h = (t_span[1]-t_span[0])/steps
    # Create an array of the time steps
    t = np.linspace(t_span[0], t_span[1], steps+1, dtype=np.float64)
    # Create an array for the solution
    y = np.zeros((steps+1, len(y_0)), dtype=np.float64)
    # Set the initial conditions
    y[0] = y_0
    # Iterate over the time steps
    for i in range(steps):
        # Calculate the next step using the Runge-Kutta method
        k1 = h*fun(t[i], y[i], *args)
        k2 = h*fun(t[i]+h/2, y[i]+k1/2, *args)
        k3 = h*fun(t[i]+h/2, y[i]+k2/2, *args)
        k4 = h*fun(t[i]+h, y[i]+k3, *args)
        y[i+1] = y[i] + 1/6*(k1+2*k2+2*k3+k4)
    return OdeResult(t=t, y=y.T)

In [ ]: def solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=None, vectorized=False, args=None, **options):
    if method == 'euler':
        return explicit_euler(fun, t_span, y0, options['steps'], args)
    elif method == 'RK4':
        return classic_runge_kutta(fun, t_span, y0, options['steps'], args)
    else:
        options.pop('steps', None)
        return scipy.integrate.solve_ivp(fun, t_span, y0, method, t_eval, dense_output, events, vectorized, args, **options)
```

Zweikörperproblem

Zur Vereinfachung lassen wir zunächst die intrinsische Rotation der Erde, sowie die Flutberge weg. Wir betrachten die beiden Körper als Punktmassen, die sich nur in der x - y -Ebene bewegen können. Diese Vereinfachungen entsprechen dem klassischen Zweikörperproblem, dass von Johannes Kepler aufgeschrieben und zuerst analytisch von Isaac Newton gelöst wurde. Sie sind ein System von gekoppelten Differentialgleichungen, die die Bewegung der beiden Körper beschreiben. \newline Wir betrachten zwei Körper und stellen dementsprechend zwei Differentialgleichungen auf:

$$\begin{aligned} m_1 \ddot{\vec{z}}_1 &= \vec{F}_{1,2} \\ m_2 \ddot{\vec{z}}_2 &= \vec{F}_{2,1} \end{aligned}$$

$\vec{F}_{2,1}$ steht für die Kraft, die die beiden Körper anzieht. Hierbei handelt es sich um die Schwerkraft:

$$\vec{F}_1 = G \frac{m_1 \cdot m_2}{r^2} \vec{e}_{12} \quad \text{mit } r := |\vec{z}_2 - \vec{z}_1|$$

Zusätzlich gilt nach dem 3ten Newtonschen Gesetz

$$\vec{F}_{1,2} = -\vec{F}_{2,1}$$

Wir setzen das Gravitationsgesetz nun in unsere DGLs ein und erhalten:

$$\begin{aligned} \ddot{\vec{z}}_1 &= \frac{G \cdot m_2}{(|\vec{z}_2 - \vec{z}_1|)^3} \cdot (\vec{z}_2 - \vec{z}_1) \\ \ddot{\vec{z}}_2 &= \frac{G \cdot m_1}{(|\vec{z}_2 - \vec{z}_1|)^3} \cdot (\vec{z}_1 - \vec{z}_2) \end{aligned}$$

Wir erhalten also vier Gleichungen:

$$\begin{aligned} \ddot{x}_1 &= \frac{G \cdot m_2}{(|\vec{x}_2 - \vec{x}_1|)^3} \cdot (x_2 - x_1) \\ \ddot{y}_1 &= \frac{G \cdot m_2}{(|\vec{x}_2 - \vec{x}_1|)^3} \cdot (y_2 - y_1) \\ \ddot{x}_2 &= \frac{G \cdot m_1}{(|\vec{x}_2 - \vec{x}_1|)^3} \cdot (x_1 - x_2) \\ \ddot{y}_2 &= \frac{G \cdot m_1}{(|\vec{x}_2 - \vec{x}_1|)^3} \cdot (y_1 - y_2) \end{aligned}$$

Die Methode `solve_ivp()` kann keine Differentialgleichungen n -ter Ordnung lösen. Wir formen deshalb unser Problem in ein DGL-System erster Ordnung um. \newline Dazu führen wir die neuen Variable v und a ein:

$$\begin{aligned} \frac{d}{dt} x_1(t) &= v_1(t) \\ \frac{d}{dt} v_1(t) &= \frac{G \cdot m_2}{(|\vec{x}_2 - \vec{x}_1|)^3} \cdot (x_2 - x_1) \end{aligned}$$

Dieses Verfahren wenden wir äquivalent für die anderen drei Gleichungen an. \newline Wir schreiben diese Differentialgleichungen nun in eine Funktion, die wir dann an `solve_ivp()` übergeben können. Die Massen der Körper können wir als Parameter übergeben. \newline Außerdem wählen wir die Anfangsbedingungen analog zu den real herschenden Bedingungen so, dass der Schwerpunkt des Zweiteilchensystems im Ursprung ruht. \newline Für einfacheres Aufrufen definieren wir noch eine Funktion zum Erstellen der Anfangsbedingungen, sowie eine Funktion die damit die Differentialgleichungen löst.

```
In [ ]: def baryzentrum(m1, m2, r):
    '''Calculate the barycenter of two bodies (The point around which both bodies orbit)
    Returns the distance of the barycenter from the body with mass m1'''
    return r*m2/(m1+m2)

def iv_stable_orbit_2body():
    '''Initial conditions for the Earth-Moon system in a stable orbit'''
    mass = [mErde, mMond]

    # We want the center of mass to be at the origin
    abstand_baryzentrum_erde = baryzentrum(mErde, mMond, rMondBahn)
    x0_Eerde = [-abstand_baryzentrum_erde, 0]
    x0_Mond = [rMondBahn-abstand_baryzentrum_erde, 0]

    # We choose the starting velocities such that the center of mass is at rest
    vy0_Eerde = 2*pi*abstand_baryzentrum_erde/TMondBahn
    vy0_Mond = 2*pi*(rMondBahn-abstand_baryzentrum_erde)/TMondBahn

    # The moon starts in positive, the earth in negative y-direction
    v0_Eerde = [0, -vy0_Eerde]
    v0_Mond = [0, vy0_Mond]
    return [x0_Eerde, x0_Mond, v0_Eerde, v0_Mond, mass]

def eq_motion_2body(t, state, mass):
    '''This function calculates the derivatives of the state vector for the two body problem to be passed to solve_ivp. \\
    state: state vector is given by [x_1, y_1, x_2, y_2, vx_1, vy_1, vx_2, vy_2] where the 1 and 2 denote the first and second \\
    mass: list of the masses of the two bodys. e.g. [m1, m2]'''
    x_1, y_1, x_2, y_2, vx_1, vy_1, vx_2, vy_2 = state
    dist_em = ((x_2 - x_1)**2 + (y_2 - y_1)**2)**0.5 # Distance between the two bodys
    # Calculate the derivatives of the state vector
    ax_1 = G * mass[1] / (dist_em**3) * (x_2 - x_1)
    ay_1 = G * mass[1] / (dist_em**3) * (y_2 - y_1)
    ax_2 = G * mass[0] / (dist_em**3) * (x_1 - x_2)
    ay_2 = G * mass[0] / (dist_em**3) * (y_1 - y_2)

    return np.array([vx_1, vy_1, vx_2, vy_2, ax_1, ay_1, ax_2, ay_2])

def two_body_problem(pos_body_1: list, pos_body_2: list, vel_body_1: list, vel_body_2: list, mass: list, t_max: float, steps=1
    ''' This Function solves the two body problem numerically using solve_ivp and the function eq_motion_2body.
    The arguments are list holding x and y-components of the position and velocity of the two bodys.
    mass: contains the masses of the bodys'''

    solution = solve_ivp(eq_motion_2body, [t_start, t_max], [*pos_body_1, *pos_body_2, *vel_body_1, *vel_body_2], args=(mass,))
    x1, y1, x2, y2, v_x1, v_y1, v_x2, v_y2 = solution.y
    return [solution.t, x1, y1, x2, y2]
```

Graphische Darstellung

Im folgenden erstellen wir parametrische zeitliche Verläufe der Positionen der beiden Körper, sowie Phasenraumplots. Zusätzlich einen 3d Plot der Bahnen der beiden Körper, sowie eine Animation.

```
In [ ]: # Calculate the solution of the two body problem
t, x_1, y_1, x_2, y_2 = two_body_problem(*iv_stable_orbit_2body(), 3*TMondbahn)
```

```
In [ ]: %matplotlib inline
# Plot the solution of the two body problem as a phase space diagram
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(6, 6)) # Create a figure and a set of subplots
fig.suptitle("Phasenraumdiagramme des Erde-Mond-Systems")

def init_phase_space(ax, a, b, title, xlabel, ylabel, color):
    ax.plot(a, b, color=color)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)

init_phase_space(axes[0,0], t, x_1, "x-Position der Erde", "$t$ / s", "$x_1$ / m", "blue")
init_phase_space(axes[0,1], t, x_2, "x-Position des Mondes", "$t$ / s", "$x_2$ / m", "red")
init_phase_space(axes[1,0], t, y_1, "y-Position der Erde", "$t$ / s", "$y_1$ / m", "blue")
init_phase_space(axes[1,1], t, y_2, "y-Position des Mondes", "$t$ / s", "$y_2$ / m", "red")
init_phase_space(axes[2,0], x_1, y_1, "Rotation der Erde", "$x_1$ / m", "$y_1$ / m", "blue")
init_phase_space(axes[2,1], x_2, y_2, "Rotation des Mondes", "$x_2$ / m", "$y_2$ / m", "red")
plt.tight_layout() # Adjust the spacing between subplots
plt.show()
plt.close()

# Plot a 3d phase space diagram
fig = plt.figure()
fig.suptitle("3D-Phasenraumdiagramm des Erde-Mond-Systems")
ax = fig.add_subplot(111, projection='3d')
ax.plot(x_1, y_1, t, label="Erde")
ax.plot(x_2, y_2, t, label="Mond")
ax.set_xlabel("$x$ / m")
ax.set_ylabel("$y$ / m")
ax.set_zlabel("$t$ / s")
ax.legend()
plt.show()
plt.close()

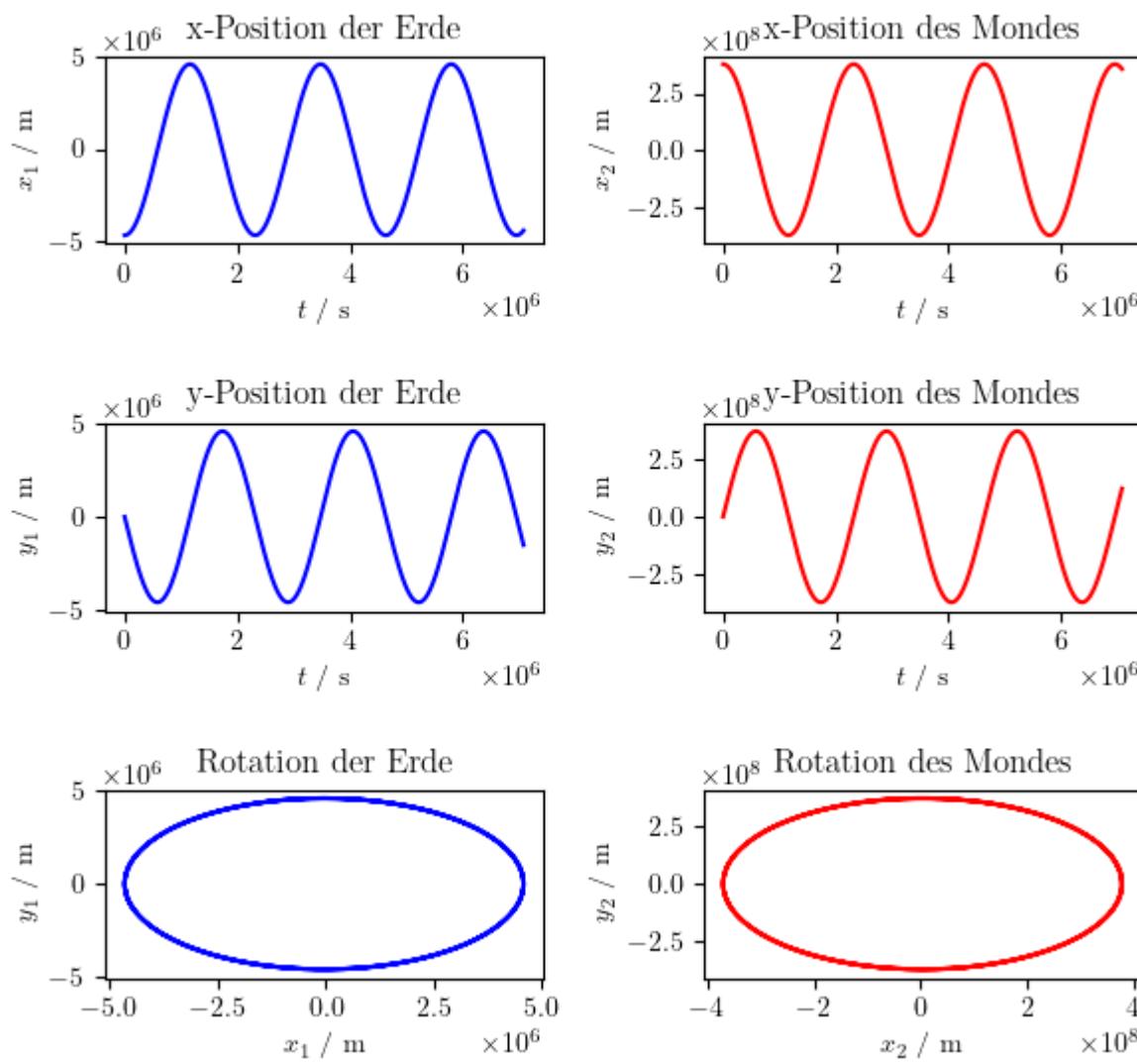
# Animation of the two body problem
if not fastExecution: # Don't render if not necessary
    fig = plt.figure(figsize=(5, 5)) # Square figure, so the circles are actually circles
    fig.suptitle("Animation des Erde-Mond-Systems")
    ax = plt.axes(xlim=(-4e8, 4e8), ylim=(-4e8, 4e8))
    earth_line, = ax.plot([], [], marker='o', lw=0.1, color='blue', label="Erde ($\cdot 10$)")
    moon_line, = ax.plot([], [], marker='o', lw=0.1, color='red', label="Mond")
    ax.legend()

    def animate(i):
        earth_line.set_data([x_1[i]*10], [y_1[i]*10])
        moon_line.set_data([x_2[i]], [y_2[i]])
        return earth_line, moon_line

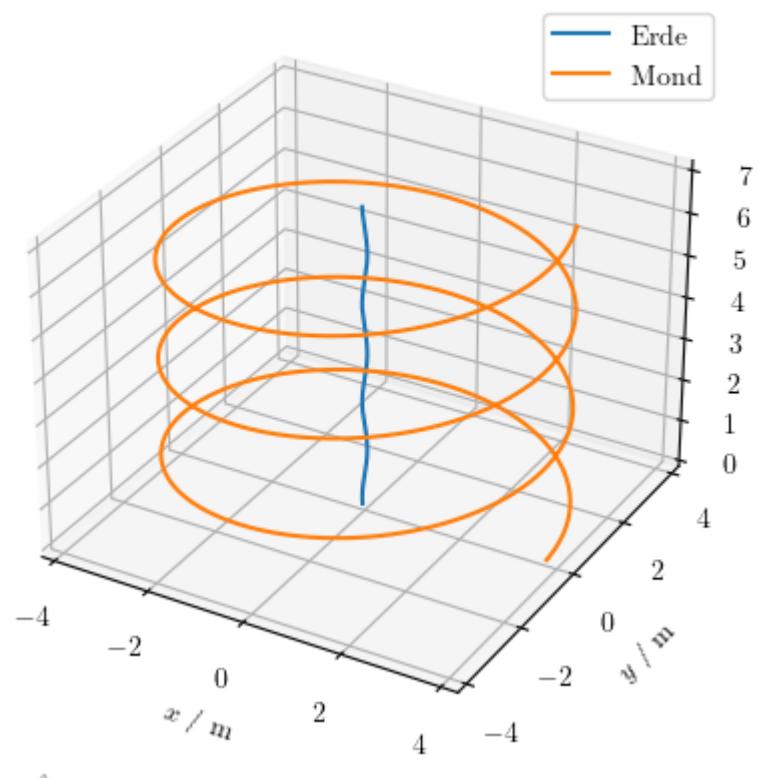
    anim = FuncAnimation(fig, animate, init_func=None, frames=t.shape[0], interval=30, blit=True)
    anim.save('Erde_Mond.gif', writer='pillow')
    plt.close()

display(Image(data=open('Erde_Mond.gif', 'rb').read(), format='png'))
```

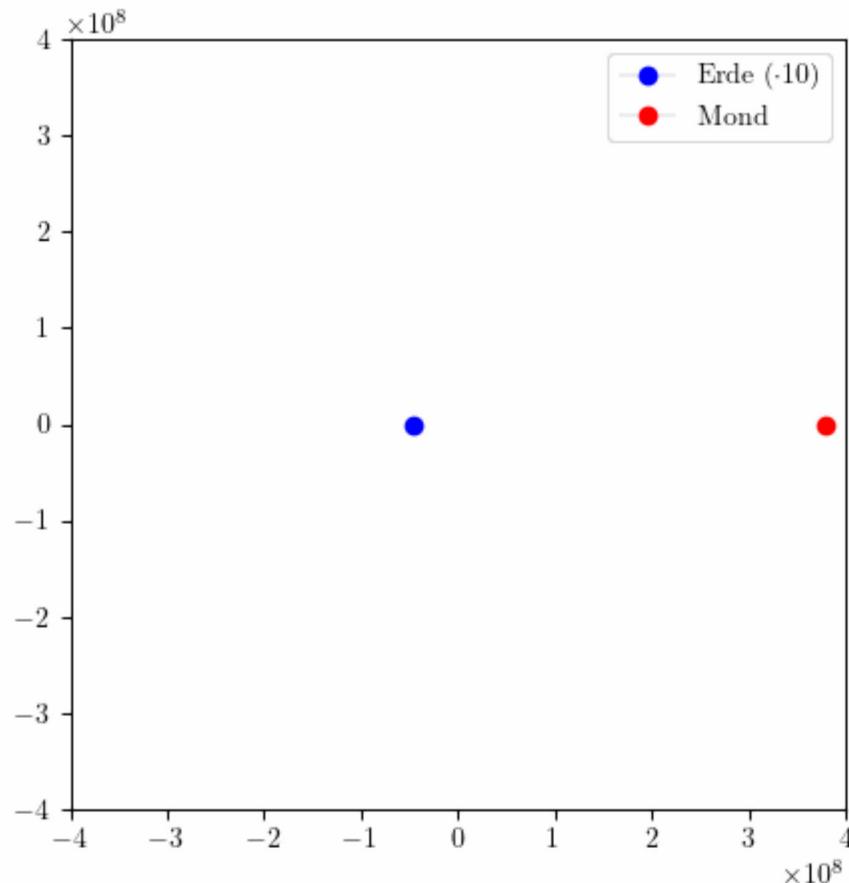
Phasenraumdiagramme des Erde-Mond-Systems



3D-Phasenraumdiagramm des Erde-Mond-Systems



Animation des Erde-Mond-Systems



Überprüfung

Nachdem wir uns unsere Lösungen nun grafisch angeschaut haben, wollen wir diese noch auf ihre Korrektheit überprüfen. Wir betrachten dazu das Baryzentrum. Dabei untersuchen wir was für einen Fehler die numerische Lösung bringt, indem wir verschiedene Verfahren vergleichen.

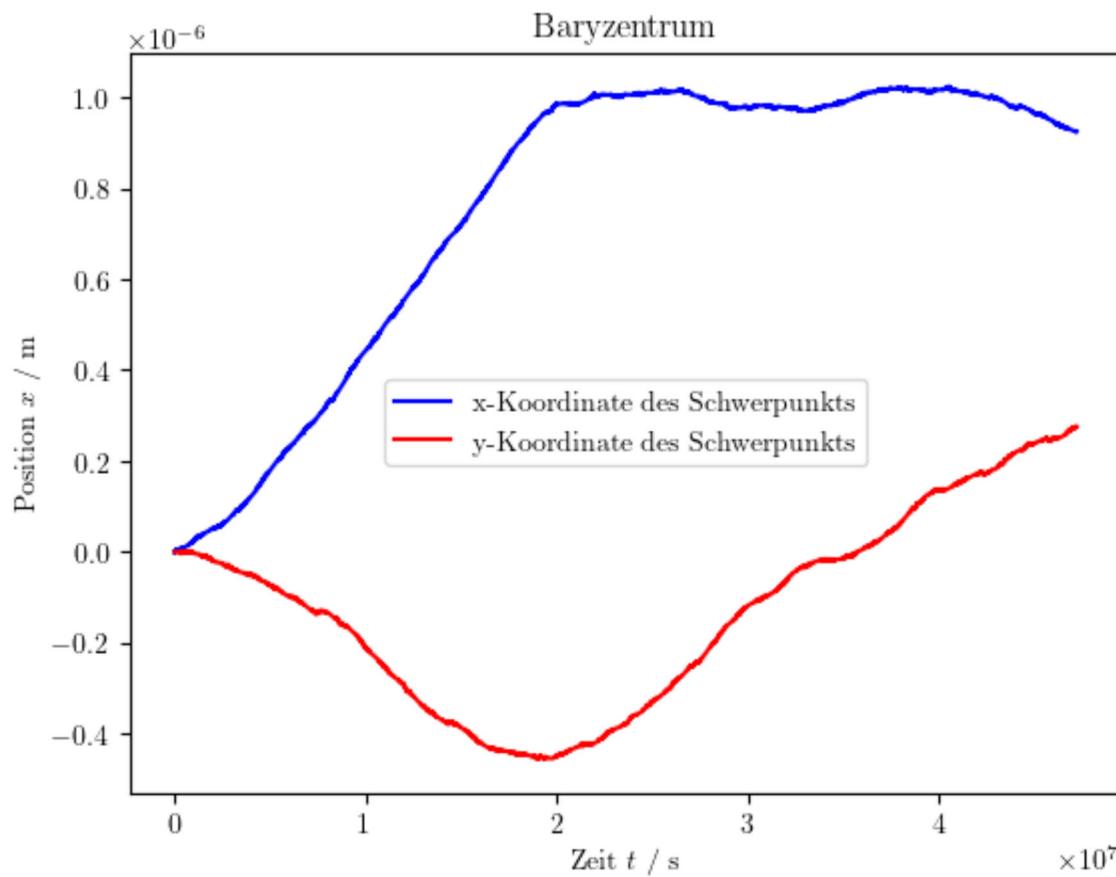
Baryzentrum

Zur Überprüfung unserer Ergebnisse und Anfangsbedingungen betrachten wir die Bewegung des Schwerpunktes. Der Schwerpunkt sollte sich in einem abgeschlossenen System, mit den von uns gewählten Anfangsbedingungen, nicht bewegen. Gegen die Zeit aufgetragen, sollten seine x-, und y-Komponente also eine Konstante bei Null (wegen der Anfangsbedingungen, Schwerpunkt im Ursprung) sein. Wie im Plot jedoch zu sehen ist, ist dies bei unseren Berechnungen nicht der Fall. Die hier bestimmten Werte liegen aber im Bereich von 10^{-5} Metern, sind also um mehrere Größenordnungen kleiner als unsere betrachteten Abstände zwischen Erde und Mond. Diese Fehler entstehen hier durch die numerische Berechnung des Problems. Sie entstehen hier entweder durch das numerische Lösen der DGL oder durch die Ungenauigkeit des float-Datentypen und der angegebenen Anfangswerte.

```
In [ ]: solution = two_body_problem(*iv_stable_orbit_2body(), (1 if fastExecution else 20)*TMondBahn)
t, x_1, y_1, x_2, y_2 = solution

def center_of_mass(m1, m2, x_1, y_1, x_2, y_2):
    '''Calculates the location of the center of mass of two bodies with masses m1 and m2 at positions (x1, y_1) and (x2, y_2)'''
    x_cm = (m1*x_1 + m2*x_2) / (m1 + m2)
    y_cm = (m1*y_1 + m2*y_2) / (m1 + m2)
    return x_cm, y_cm

%matplotlib inline
# Plot the change of the center of mass over time
fig, ax = plt.subplots()
fig.set_label("Bewegung des Baryzentrum des Erde-Mond-Systems")
ax.plot(t, center_of_mass(mErde, mMond, x_1, y_1, x_2, y_2)[0], color='blue', label='x-Koordinate des Schwerpunkts')
ax.plot(t, center_of_mass(mErde, mMond, x_1, y_1, x_2, y_2)[1], color='red', label='y-Koordinate des Schwerpunkts')
ax.set_xlabel('Zeit $t$ / s')
ax.set_ylabel('Position $x$ / m')
ax.set_title('Baryzentrum')
ax.legend()
plt.show()
plt.close()
```



Vergleich von explizitem Euler-Verfahren, Runge-Kutta-Verfahren 4. Ordnung und Runge-Kutta-Verfahren 45 (Standard in `solve_ivp()`)

Zwischen der Lösung mit dem expliziten Euler-Verfahren und der Lösung mit dem Runge-Kutta-Verfahren 4. Ordnung ist ein deutlicher Unterschied zu erkennen. Es ist deutlich zu sehen, dass der Orbit des Mondes mit jedem Umlauf größer wird. Hier wird also eindeutig ein größerer numerischer Fehler gemacht. Beim RK4 und RK45 ist derartiges nicht zu erkennen. Sie sind voneinander nicht zu unterscheiden. Dies ist auch zu erwarten, da RK45 eine Weiterentwicklung von RK4 ist und nur bei sehr großen Schrittweiten Unterschiede zu erkennen sind. Das Euler-Verfahren ist also in der Tat schlechter als das RK4-Verfahren, dafür aber auch deutlich weniger aufwendig. Diese kleine Genauigkeitseinbuße ist aber zu verschmerzen.

Zeitvergleich der Verfahren:

Das Euler Vefahren ist deutlich schneller als das explizite Runge-Kutta Verfahren (RK4). Der kürzere Zeitaufwand macht die etwas größere Ungenauigkeit wieder wett. RK45 ist trotzdem deutlich schneller und am genauesten und darum den anderen Verfahren vorzuziehen. Erstaunlich, da RK4 eigentlich weniger Rechenaufwand bedeuten sollte als RK45. Wahrscheinlich eine sehr effiziente Implementierung von RK45 im SciPy Paket.

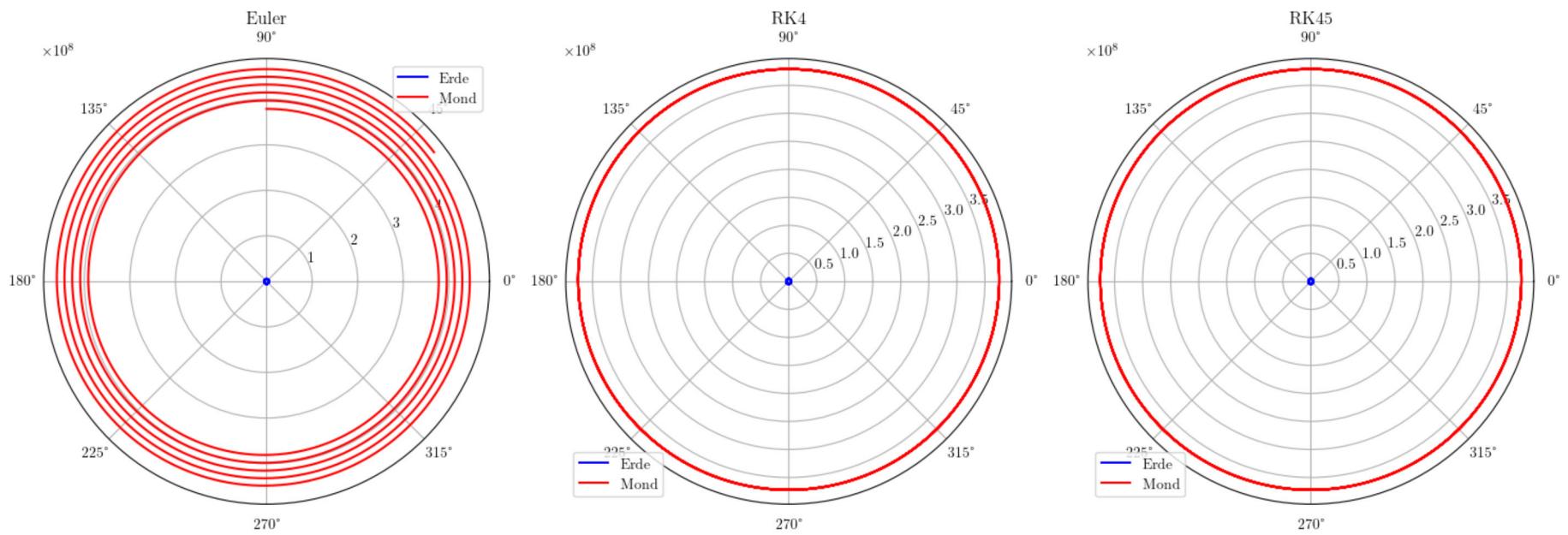
```
In [ ]: %matplotlib inline
revolutions = (1 if fastExecution else 6)
solution_euler = two_body_problem(*iv_stable_orbit_2body(), revolutions*TMondbahn, 10000, method="euler")
solution_rk4 = two_body_problem(*iv_stable_orbit_2body(), revolutions*TMondbahn, 10000, method="RK4")
solution_rk45 = two_body_problem(*iv_stable_orbit_2body(), revolutions*TMondbahn, 10000, method="RK45")

# To plot the solution in polar coordinates we need to convert the cartesian coordinates to polar coordinates
r = lambda x, y: np.sqrt(x**2 + y**2)
theta = lambda x, y: np.arctan2(x, y)

# Create a figure and a set of subplots
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(15, 15), subplot_kw={'projection': 'polar'})

def init_polar(ax, t, x_1, y_1, x_2, y_2, title, color_1, color_2, label_1, label_2):
    ax.plot(theta(x_1, y_1), r(x_1, y_1), color=color_1, label=label_1)
    ax.plot(theta(x_2, y_2), r(x_2, y_2), color=color_2, label=label_2)
    ax.set_title(title)
    ax.legend()

init_polar(ax1, *solution_euler, "Euler", "blue", "red", "Erde", "Mond")
init_polar(ax2, *solution_rk4, "RK4", "blue", "red", "Erde", "Mond")
init_polar(ax3, *solution_rk45, "RK45", "blue", "red", "Erde", "Mond")
plt.tight_layout()
plt.show()
plt.close()
```



```
In [ ]: if fastExecution:
    print("240 ms ± 85.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each) \n843 ms ± 176 ms per loop (mean ± std. dev. of
else:
    %timeit solution_euler = two_body_problem(*iv_stable_orbit_2body(), revolutions*TMondbahn, 10000, method="euler")
    %timeit solution_rk4 = two_body_problem(*iv_stable_orbit_2body(), revolutions*TMondbahn, 10000, method="RK4")
    %timeit solution_rk45 = two_body_problem(*iv_stable_orbit_2body(), revolutions*TMondbahn, 10000, method="RK45")
```

240 ms ± 85.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
843 ms ± 176 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
176 ms ± 14.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Vergleich mit der exakten Lösung des Zweikörperproblems:

Eine exakte analytische Lösung des Zweikörperproblems ist mit genügend Annahmen möglich. Dazu wird im Schwerpunktssystem gearbeitet. Aus den Bewegungsgleichungen der Körper die nur mit der Schwerkraft wechselwirken, kann man folgern, dass der Schwerpunkt des Systems eine gleichförmige Bewegung ausführt. Mit unseren Anfangsbedingungen bewegt dieser sich aber nicht.

Die beiden Körper können sich um ihren Schwerpunkt je nach Anfangsbedingungen auf verschiedenen Bahnen bewegen. Dies lässt sich durch die Exzentrizität beschreiben. Mögliche Bahnen sind Kreise, Ellipsen, Parabeln und Hyperbeln. Wir haben die Anfangsbedingungen so gewählt, dass die Bahn eine Kreisbahn ist. \newline **Unsere Ergebnisse stimmen also mit der exakten Lösung des Zweikörperproblems überein, da wir zwei geschlossene Kreisbahnen um den gemeinsamen Schwerpunkt (mit unseren Anfangsbedingungen der Ursprung) erhalten. Diese Kreisbahnen lassen sich mit einfachen trigonometrischen Funktionen beschreiben.**

$$\mathbf{r}_E(t) = \begin{pmatrix} x_E(t) \\ y_E(t) \end{pmatrix} = \begin{pmatrix} r_{E_0} \cos(\omega_E t) \\ r_{E_0} \sin(\omega_E t) \end{pmatrix} \quad (1)$$

$$\mathbf{r}_M(t) = \begin{pmatrix} x_M(t) \\ y_M(t) \end{pmatrix} = \begin{pmatrix} r_{M_0} \cos(\omega_M t) \\ r_{M_0} \sin(\omega_M t) \end{pmatrix} \quad (2)$$

Diese lassen sich ebenfalls leicht plotten.

Unser numerischer Ansatz ist natürlich in der Lage alle möglichen Bahnen zu berechnen, also auch Ellipsen und Hyperbeln, es ließe sich also auch ein Komet oder ein Sling-Shot-Manöver berechnen. Mit den geeigneten Anfangsbedingungen ließen sich diese auch analytisch berechnen.

Eine weitere Möglichkeit unsere Ergebnisse zu überprüfen ist das dritte Keplersche Gesetz, es besagt, dass die Umlaufzeit T des Zweikörpersystems beträgt:

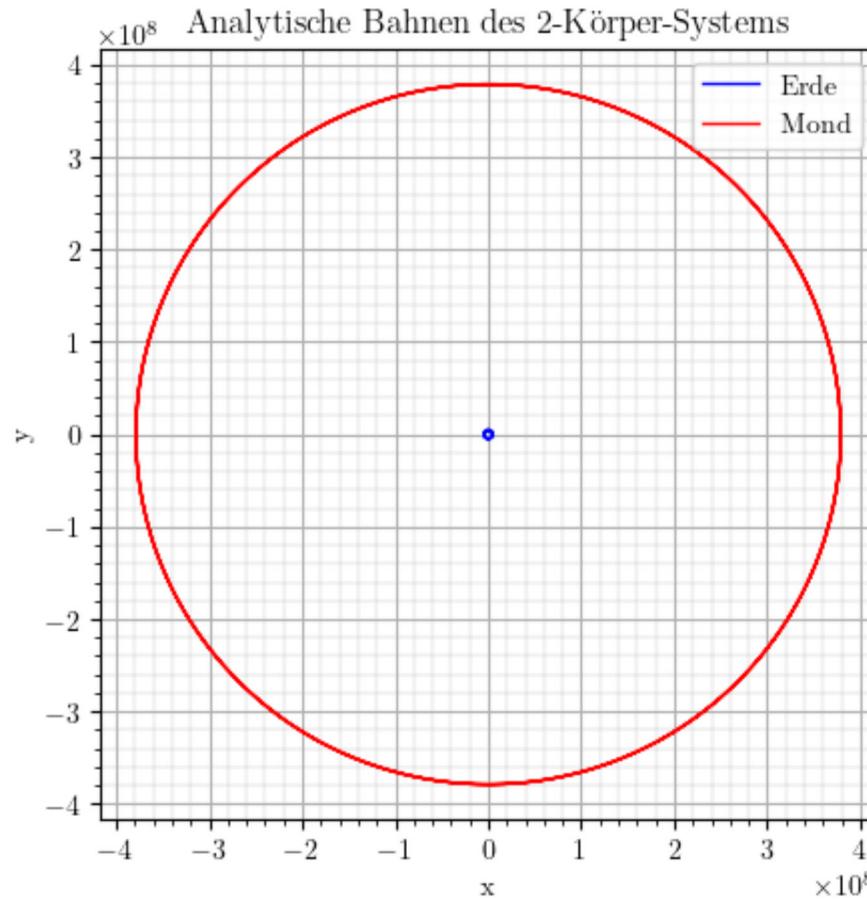
$$T^2 = \frac{4\pi^2 \cdot a^3}{G \cdot M}$$

Mit G als Gravitationskonstante, $M = m_{Erde} + m_{Mond}$ und a als großer Halbachse welche in unserem genäherten Fall der Kreisbahn $a = r_{Mondbahn}$ entspricht.

```
In [ ]: r_E0, r_M0 = iv_stable_orbit_2body()[0][0], iv_stable_orbit_2body()[1][0]
time_points = np.linspace(0, 3*TmondBahn, 1000)
x_E, y_E = np.vectorize(lambda t: r_E0 * np.cos(2*pi/TmondBahn * t))(time_points), np.vectorize(lambda t: r_E0 * np.sin(2*pi/TmondBahn * t))(time_points)
x_M, y_M = np.vectorize(lambda t: r_M0 * np.cos(2*pi/TmondBahn * t))(time_points), np.vectorize(lambda t: r_M0 * np.sin(2*pi/TmondBahn * t))(time_points)

# Plot the analytical solution with our boundary conditions
plt.rc('axes', axisbelow=True)
plt.figure(figsize=(5, 5))
plt.title("Analytische Bahnen des 2-Körper-Systems")
plt.grid('minor', 'minor', linestyle='-', linewidth=0.2)
plt.grid('major', 'major', linestyle='-', linewidth=0.8)
plt.minorticks_on()
plt.xlabel("x")
plt.ylabel("y")
plt.plot(x_E, y_E, 'b', linewidth=1, label='Erde')
plt.plot(x_M, y_M, 'r', linewidth=1, label='Mond')
plt.legend()
plt.show()

print(f'3. Keplersches Gesetze T: {TmondBahn} =~ {(4*pi**2*rMondBahn**3/(G*(mErde+mMond)))**(1/2)}')
```



3. Keplersches Gesetze T: 2360591.54496 =~ 2350028.5085322363

Man erkennt, dass die beiden Verfahren von Runge-Kutta sehr genau an der analytischen Lösung liegen und bis auf kleine numerische Abweichungen sehr genau sind.

Vierkörperproblem mit Randbedingungen

Wir erweitern nun das Zweikörperproblem um die beiden Flutberge, die sich auf der Erdoberfläche befinden. Dazu werden die beiden Flutberge als Punktmassen genähert, deren Abstand zur Erde zu jeder Zeit dem Erdradius entspricht. Die gegenseitige Gravitationskraft der beiden Flutberge ist sehr gering und wird daher im Folgenden vernachlässigt, d.h. $\vec{F}_{21} = \vec{F}_{12} = 0$.

Die radiale Kraftkomponente der Flutberge muss den Abstand zur Erde konstant halten (Erdradius). Dazu werden radiale Zwangskräfte eingeführt, die die ebenfalls radiale Gravitationskraft der Erde beinhalten. \vec{F}_{Zi} ist also Summe von Zwangskraft und Gravitationskraft der Erde auf Flutberg i , sie wirkt entgegengestellt auch auf die Erde.

Damit ergibt sich für die Gesamtkräfte \vec{F}_E , \vec{F}_M , \vec{F}_1 , \vec{F}_2 auf Erde, Mond, Flutberg 1 bzw. Flutberg 2 folgendes Gleichungssystem:

\newline

$$\begin{aligned}\vec{F}_E &= \vec{F}_{ME} - \vec{F}_{Z1} - \vec{F}_{Z2} \\ \vec{F}_M &= \vec{F}_{EM} + \vec{F}_{1M} + \vec{F}_{2M} \\ \vec{F}_1 &= \vec{F}_{M1} + \vec{F}_{Z1} \\ \vec{F}_2 &= \vec{F}_{M2} + \vec{F}_{Z2}\end{aligned}$$

Die Koordinaten des i -ten Flutbergs lassen sich am elegantesten in Polarkoordinaten relativ zu den Erdkoordinaten \vec{r}_E angeben: $r_i = r_E + R_E \dot{e}_{r,i}$. Hierfür werden φ_i der Winkel des i -ten Flutbergs relativ zur Erde, m_i die Masse, $\dot{e}_{r,i}$ der radiale Einheitsvektor und $\dot{e}_{\varphi,i}$ der Einheitsvektor in φ_i -Richtung in Bezug auf den i -ten Flutberg, verwendet.

Die Kraft auf den i -ten Flutberg lässt sich nun in einen radialen Anteil und einen Winkelanteil aufteilen: \newline

$$\begin{aligned}\vec{F}_i &= (\vec{F}_{Mi})_{\varphi,i} \dot{e}_{\varphi,i} + (\vec{F}_{Mi})_{r,i} \dot{e}_{r,i} + F_{Zi} \dot{e}_{r,i} \\ &= (\vec{F}_{Mi})_{\varphi,i} \dot{e}_{\varphi,i} + ((\vec{F}_{Mi})_{r,i} + F_{Zi}) \dot{e}_{r,i}.\end{aligned}$$

Die Beschleunigung des i -ten Flutbergs in Richtung Erde muss der Zentripetalbeschleunigung für stabile Kreisbahnen abzüglich der Beschleunigung der Erde in radiale Richtung betragen, damit die Zwangsbedingung (ZB) erfüllt ist, d.h. \newline

$$(\vec{F}_{Mi})_{r,i} + F_{Zi} \stackrel{\text{ZB}}{=} -m_i \left(R_E \dot{\varphi}_i^2 - (\ddot{\vec{r}}_E)_{r,i} \right),$$

Die so aufgestellten Differentialgleichungen sind noch gekoppelt, sie müssen also noch nach den einzelnen Beschleunigungen aufgelöst werden um an `solve_ivp()` übergeben werden zu können. Im Folgenden werden die Gleichungen hergeleitet.

\newline

Für die Flutberge werden Polarkoordinaten verwendet, es gilt also: \newline

$$\begin{aligned}x_i &= x_E + R_E \cos \varphi_i \\y_i &= y_E + R_E \sin \varphi_i\end{aligned}$$

Stellt man die Zwangsbedingung nach der Zwangskraft um und setzt in die Kraft auf die Erde ein, so erhält man: \newline

$$\vec{F}_E = \vec{F}_{ME} + (m_1 \left(R_E \dot{\varphi}_1^2 - (\ddot{r}_E)_{r,1} \right) + (\vec{F}_{M1})_{r,1}) \vec{e}_{r,1} + (m_2 \left(R_E \dot{\varphi}_2^2 - (\ddot{r}_E)_{r,2} \right) + (\vec{F}_{M2})_{r,2}) \vec{e}_{r,2}$$

Die r-Komponente erhält man durch Projektion mit dem entsprechenden Einheitsvektor $\vec{e}_{r,i}$, und es ergibt sich mit umstellen der Gleichung: \newline

$$\begin{aligned}\ddot{x}_E &= \frac{1}{\mu_x} (a(x_M - x_E) - b\ddot{y}_E + c_1 \cos \varphi_1 + c_2 \cos \varphi_2) \\ \ddot{y}_E &= \frac{1}{\mu_y} (a(y_M - y_E) - b\ddot{x}_E + c_1 \sin \varphi_1 + c_2 \sin \varphi_2)\end{aligned}$$

Mit den Abkürzungen: \newline

$$\begin{aligned}\mu_x &= m_E + m_1 \cos \varphi_1^2 + m_2 \cos \varphi_2^2 \\ \mu_y &= m_E + m_1 \sin \varphi_1^2 + m_2 \sin \varphi_2^2 \\ a &= \frac{G m_M m_E}{((x_M - x_E)^2 + (y_M - y_E)^2)^{\frac{3}{2}}} \\ b &= m_1 \sin \varphi_1 \cos \varphi_1 + m_2 \sin \varphi_2 \cos \varphi_2 \\ c_i &= m_i R_E \dot{\varphi}_i^2 + \frac{G m_M m_i ((x_M - x_i) \cos \varphi_i + (y_M - y_i) \sin \varphi_i)}{((x_M - x_i)^2 + (y_M - y_i)^2)^{\frac{3}{2}}}\end{aligned}$$

Um diese Differentialgleichungen an `solve_ivp()` zu übergeben müssen sie noch entkoppelt werden. Hierzu setzen wir \ddot{y}_E in die Gleichung für \ddot{x}_E ein, und stellen nach \ddot{x}_E um (\ddot{y}_E erhalten wir dann durch einsetzen vom berechneten \ddot{x}_E): \newline

$$\ddot{x}_E = \frac{a(x_M - x_E) - \frac{b}{\mu_y} (a(y_M - y_E) + c_1 \sin \varphi_1 + c_2 \sin \varphi_2) + c_1 \cos \varphi_1 + c_2 \cos \varphi_2}{\mu_x - \frac{b^2}{\mu_y}}$$

Die Kraft auf den Mond lässt sich mit der Gravitationskraft einfach durch einsetzen bestimmen, was zu folgenden Gleichungen führt: \newline

$$\begin{aligned}\ddot{x}_M &= G \left(\frac{m_E(x_E - x_M)}{((x_E - x_M)^2 + (y_E - y_M)^2)^{\frac{3}{2}}} + \frac{m_1(x_1 - x_M)}{((x_1 - x_M)^2 + (y_1 - y_M)^2)^{\frac{3}{2}}} + \frac{m_2(x_2 - x_M)}{((x_2 - x_M)^2 + (y_2 - y_M)^2)^{\frac{3}{2}}} \right) \\ \ddot{y}_M &= G \left(\frac{m_E(y_E - y_M)}{((x_E - x_M)^2 + (y_E - y_M)^2)^{\frac{3}{2}}} + \frac{m_1(y_1 - y_M)}{((x_1 - x_M)^2 + (y_1 - y_M)^2)^{\frac{3}{2}}} + \frac{m_2(y_2 - y_M)}{((x_2 - x_M)^2 + (y_2 - y_M)^2)^{\frac{3}{2}}} \right)\end{aligned}$$

Für die Kraft auf die Flutberge kann die Zwangskraft in der Form, in der sie angegeben ist eingesetzt werden: \newline

$$\vec{F}_i = (\vec{F}_{Mi})_{\varphi,i} \vec{e}_{\varphi,i} - (m_i \left(R_E \dot{\varphi}_i^2 - (\ddot{r}_E)_{r,i} \right)) \vec{e}_{r,i}.$$

Die r- bzw. φ -Komponente erhält man wieder mit den entsprechenden Einheitsvektoren, und wir erhalten: \newline

$$\begin{aligned}\ddot{x}_i &= -\frac{G m_M ((y_M - y_i) \cos \varphi_i - (x_M - x_i) \sin \varphi_i)}{((x_M - x_i)^2 + (y_M - y_i)^2)^{\frac{3}{2}}} \sin \varphi_i - (R_E \dot{\varphi}_i^2 - (\ddot{x}_E \cos \varphi_i + \ddot{y}_E \sin \varphi_i)) \cos \varphi_i \\ \ddot{y}_i &= \frac{G m_M ((y_M - y_i) \cos \varphi_i - (x_M - x_i) \sin \varphi_i)}{((x_M - x_i)^2 + (y_M - y_i)^2)^{\frac{3}{2}}} \cos \varphi_i - (R_E \dot{\varphi}_i^2 - (\ddot{x}_E \cos \varphi_i + \ddot{y}_E \sin \varphi_i)) \sin \varphi_i\end{aligned}$$

Wir brauchen aber $\ddot{\varphi}_i$, dazu leiten wir die Koordinaten der Flutberge zwei mal nach der Zeit ab: \newline

$$\begin{aligned}\ddot{r}_i &= \ddot{r}_E + R_E (\ddot{\varphi}_i \vec{e}_{\varphi,i} - \dot{\varphi}_i^2 \vec{e}_{r,i}) \\ \ddot{\varphi}_i \vec{e}_{\varphi,i} &= \frac{\ddot{r}_i - \ddot{r}_E}{R_E} + \dot{\varphi}_i^2 \vec{e}_{r,i}\end{aligned}$$

Durch Multiplikation mit $\vec{e}_{\varphi,i}$ erhält man: \newline

$$\ddot{\varphi}_i = \frac{-(\ddot{x}_i - \ddot{x}_E) \sin \varphi_i + (\ddot{y}_i - \ddot{y}_E) \cos \varphi_i}{R_E}$$

Lösen mit SymPy

Für das lösen/entkoppeln der Differentialgleichungen kann alternativ zur Berechnung per Hand, wie oben durchgeführt, SymPy verwendet werden.

Hierzu werden zunächst die oben aufgestellten Differentialgleichungen in Pythoncode geschrieben. Bei der Definition können wir SymPy die verschiedenen Ableitungen, Skalarprodukte und anderen per Hand sehr zeitaufwendigen Schritte direkt berechnen lassen. Danach fassen wir für die Erde noch Teile der Gleichung zur Variablen zusammen, um `linsolve()` die Arbeit zu erleichtern. Dieser Schritt ist sehr wichtig, da er die Rechenzeit um mehr als einen Faktor 100 reduziert!

Nachdem wir die Differentialgleichungen entkoppelt haben und aufgelöst haben, werden sie mit `lambdify()` in numerische Funktionen umgewandelt. Der Umwandlungsprozess ist ebenfalls sehr zeitaufwendig und beträgt mehr als die Hälfte der Gesamtaufzeit. Er ist aber notwendig, um die Berechnung mit `solve_ivp()` durchführen zu können.

Die hier mit SymPy berechneten Funktionen sind leider sehr lang und kompliziert; das vereinfachen mit `simplify()` dauert leider sehr lange und lohnt sich hier nicht, da die unten per Hand definierten Funktionen sowieso schneller sind als alles was SymPy erreichen könnte.

```

In [ ]: t = sympy.symbols('t', real=True, positive=True) # Time variable for all functions
x_E, y_E, x_M, y_M, phi_1, phi_2 = [f(t) for f in sympy.symbols('x_E, y_E, x_M, y_M, phi_1, phi_2', real=True, cls=sympy.Function)]
# Variables for the masses and distances between the bodies
mErdeS, mMondS, m1S, m2S = sympy.symbols('m_Eerde, m_Mond, m_1, m_2', real=True, positive=True)
dist_em, dist_m1, dist_m2 = sympy.symbols('d_em, d_m1, d_m2', real=True, positive=True)

# Coordinate vectors of the tides
r_1 = sympy.Matrix([x_E + RErde*sympy.cos(phi_1), y_E + RERde*sympy.sin(phi_1)])
r_2 = sympy.Matrix([x_E + RErde*sympy.cos(phi_2), y_E + RERde*sympy.sin(phi_2)])

# Force vectors
F_ME = -G*mErde*mMondS / (dist_em**3) * sympy.Matrix([x_E-x_M, y_E-y_M])
F_1M = -G*mMond*m1S / (dist_m1**3) * (sympy.Matrix([x_M, y_M]) - r_1)
F_2M = -G*mMond*m2S / (dist_m2**3) * (sympy.Matrix([x_M, y_M]) - r_2)

# Polar unit vectors for phi_1 and phi_2
e_r_1, e_phi_1 = sympy.Matrix([sympy.cos(phi_1), sympy.sin(phi_1)]), sympy.Matrix([-sympy.sin(phi_1), sympy.cos(phi_1)])
e_r_2, e_phi_2 = sympy.Matrix([sympy.cos(phi_2), sympy.sin(phi_2)]), sympy.Matrix([-sympy.sin(phi_2), sympy.cos(phi_2)])

# The radial component of the earth's acceleration vector
a_E_r = sympy.Matrix([x_E.diff(t, t), y_E.diff(t, t)])

# Constraint force vectors on the tides
F_Z1 = (F_1M.dot(e_r_1) + m1S*(RErde*(phi_1.diff(t))**2 - a_E_r.dot(e_r_1))) * e_r_1
F_Z2 = (F_2M.dot(e_r_2) + m2S*(RErde*(phi_2.diff(t))**2 - a_E_r.dot(e_r_2))) * e_r_2

# Forces on the tides
F_1 = (-F_1M).dot(e_phi_1) * e_phi_1 - (m1S*(RErde*phi_1.diff(t)**2 - a_E_r.dot(e_r_1))) * e_r_1
F_2 = (-F_2M).dot(e_phi_2) * e_phi_2 - (m2S*(RErde*phi_2.diff(t)**2 - a_E_r.dot(e_r_2))) * e_r_2

# We now write the equations of motion
eq1 = sympy.Eq(mErdeS * x_E.diff(t, t), (F_ME[0] + F_Z1[0] + F_Z2[0]))
eq2 = sympy.Eq(mErdeS * y_E.diff(t, t), (F_ME[1] + F_Z1[1] + F_Z2[1]))
eq3 = sympy.Eq(mMondS * x_M.diff(t, t), (-F_ME[0] + F_1M[0] + F_2M[0]))
eq4 = sympy.Eq(mMondS * y_M.diff(t, t), (-F_ME[1] + F_1M[1] + F_2M[1]))
eq5 = sympy.Eq(m1S*(r_1[0]).diff(t, t), F_1[0])
eq6 = sympy.Eq(m1S*(r_1[1]).diff(t, t), F_1[1])
eq7 = sympy.Eq(m2S*(r_2[0]).diff(t, t), F_2[0])
eq8 = sympy.Eq(m2S*(r_2[1]).diff(t, t), F_2[1])

# From here on we replace the variables with symbols and actually solve the equations
# Replace the functions and their derivatives with symbols
function_variables = [x_E, y_E, x_M, y_M, phi_1, phi_2, x_E.diff(t), y_E.diff(t), x_M.diff(t), y_M.diff(t), phi_1.diff(t), phi_2.diff(t)]
variables = sympy.symbols('x_E, y_E, x_M, y_M, phi_1, phi_2 \dot{x}_E, \dot{y}_E, \dot{x}_M, \dot{y}_M, \dot{\phi}_1, \dot{\phi}_2')
variable_replacements = {key: value for key, value in zip(function_variables, variables)}
equations_var = [eq.subs(variable_replacements) for eq in [eq1, eq2, eq3, eq4, eq5, eq6, eq7, eq8]] # Replace the variables in
x_E, y_E, x_M, y_M, phi_1, phi_2, x_E_dot, y_E_dot, x_M_dot, y_M_dot, phi_1_dot, phi_2_dot, x_E_ddot, y_E_ddot, x_M_ddot, y_M_ddot

# Distances as equations for replacing the constants
dist_em_eq = sympy.sqrt((x_E-x_M)**2+(y_E-y_M)**2)
dist_m1_eq = sympy.sqrt((x_M-x_E-RErde*sympy.cos(phi_1))**2+(y_M-y_E-RERde*sympy.sin(phi_1))**2)
dist_m2_eq = sympy.sqrt((x_M-x_E-RErde*sympy.cos(phi_2))**2+(y_M-y_E-RERde*sympy.sin(phi_2))**2)

# Solving the equations for the moon
moon_acc = sympy.linsolve(equations_var[2:4], variables[14:16])
moon_acc = [eq.subs({dist_em: dist_em_eq, dist_m1: dist_m1_eq, dist_m2: dist_m2_eq}) for eq in moon_acc.args[0]]

# Solving the equations for the earth
# Replace parts of the equations with symbols. These are similar but not exactly the same as the symbols in the description/canonical
mu_xs, mu_ys, part_a, part_b, c11, c13, c21, c23, c31, c33, c41, c43 = sympy.symbols('mu_xs, mu_ys, a, b, c_11, c_13, c_21, c_23')

earth_x_rhs = (equations_var[0].rhs)
mu_x_eq = earth_x_rhs.expand().collect(x_E_ddot).coeff(x_E_ddot)
earth_x_rhs = earth_x_rhs.expand().collect(x_E_ddot).subs({mu_x_eq: mu_xs})
b_eq = earth_x_rhs.expand().collect(y_E_ddot).coeff(y_E_ddot)
earth_x_rhs = earth_x_rhs.expand().collect(y_E_ddot).subs({b_eq: part_b})
a_eq = earth_x_rhs.expand().collect(x_M).collect(x_E).coeff(x_M)
earth_x_rhs = earth_x_rhs.expand().collect(x_M).collect(x_E).subs({a_eq: part_a})
exprCos = earth_x_rhs.expand().collect(sympy.cos(phi_1)).collect(sympy.cos(phi_2))
c11_eq = exprCos.coeff(sympy.cos(phi_1))
c21_eq = exprCos.coeff(sympy.cos(phi_2))
c13_eq = exprCos.coeff(sympy.cos(phi_1)**3)*sympy.cos(phi_1)**2
c23_eq = exprCos.coeff(sympy.cos(phi_2)**3)*sympy.cos(phi_2)**2
earth_x_rhs = earth_x_rhs.expand().collect(sympy.cos(phi_1)).collect(sympy.cos(phi_2)).subs({c11_eq: c11, c21_eq: c21, c13_eq: c13, c23_eq: c23})

earth_y_rhs = (equations_var[1].rhs)
mu_y_eq = earth_y_rhs.expand().collect(y_E_ddot).coeff(y_E_ddot)
earth_y_rhs = earth_y_rhs.expand().collect(y_E_ddot).subs({mu_y_eq: mu_ys})
b_eq = earth_y_rhs.expand().collect(x_E_ddot).coeff(x_E_ddot)
earth_y_rhs = earth_y_rhs.expand().collect(x_E_ddot).subs({b_eq: part_b})
a_eq = earth_y_rhs.expand().collect(y_M).collect(y_E).coeff(y_M)
earth_y_rhs = earth_y_rhs.expand().collect(y_M).collect(y_E).subs({a_eq: part_a})
exprSin = earth_y_rhs.expand().collect(sympy.sin(phi_1)).collect(sympy.sin(phi_2))
c31_eq = exprSin.coeff(sympy.sin(phi_1))
c41_eq = exprSin.coeff(sympy.sin(phi_2))
c33_eq = exprSin.coeff(sympy.sin(phi_1)**3)*sympy.sin(phi_1)**2

```

```

c43_eq = exprSin.coeff(sympy.sin(phi_2)**3)*sympy.sin(phi_2)**2
earth_y_rhs = earth_y_rhs.expand().collect(sympy.sin(phi_1)).collect(sympy.sin(phi_2)).subs({c31_eq: c31, c41_eq: c41, c33_eq: c33})

# Rewrite the equations with the substituted symbols and solve
simplified_earth_x = sympy.Eq(equations_var[0].lhs, earth_x_rhs)
simplified_earth_y = sympy.Eq(equations_var[1].lhs, earth_y_rhs)
earth_acc = sympy.linsolve([simplified_earth_x, simplified_earth_y], [x_E_ddot, y_E_ddot])
earth_acc = [eq.subs({mu_xs: mu_x_eq, mu_ys: mu_y_eq, part_a: a_eq, part_b: b_eq, c11: c11_eq, c13: c13_eq, c21: c21_eq, c23: c23}) for eq in equations]

# Solve the equations for the tides
phi_1_acc = (((-equations_var[4].rhs/m1S - earth_acc[0])*sympy.sin(phi_1) + (equations_var[5].rhs/m1S - earth_acc[1])*sympy.cos(phi_1)) / (m1S - earth_acc[0]))**2
phi_2_acc = (((-equations_var[6].rhs/m2S - earth_acc[0])*sympy.sin(phi_2) + (equations_var[7].rhs/m2S - earth_acc[1])*sympy.cos(phi_2)) / (m2S - earth_acc[0]))**2

# Lambdify the equations for the numerical integration
lambdified_accelerations = [sympy.lambdify(['x_E', 'y_E', 'x_M', 'y_M', 'phi_1', 'phi_2', 'vx_E', 'vy_E', 'vx_M', 'vy_M', 'omega_1', 'omega_2'], acc, modules='numpy') for acc in earth_acc]

```

Gleichungen per Hand

Wie oben bereits gesagt, ist es sinnvoller die Gleichungen per Hand zu lösen, da `sympy` die Gleichungen nicht in "normaler" Zeit vereinfachen kann, das heißt, dass man ein wenig Rechenzeit sparen kann, wenn man es per Hand macht. Dieses "nicht vereinfachen" der Gleichungen führt zu einer Verlängerung der Laufzeit um ungefähr 730%, diese nicht vereinfachten Gleichungen führen auch zu mehr numerischen Fehlern.

```

In [ ]: def tide_acceleration(x_M, y_M, x_i, y_i, phi_i, ax_E, ay_E, omega_i, m_M):
    '''Calculates the angular acceleration of a tide, using the equation above'''
    part_a = G*m_M* ((y_M - y_i)*np.cos(phi_i) - (x_M - x_i)*np.sin(phi_i)) / ((x_M - x_i)**2 + (y_M - y_i)**2)**(3/2)
    part_b = (REerde*omega_i**2 - (ax_E*np.cos(phi_i) + ay_E*np.sin(phi_i)))
    ax_i = -part_a*np.sin(phi_i) - part_b*np.cos(phi_i)
    ay_i = part_a*np.cos(phi_i) - part_b*np.sin(phi_i)

    alpha_i = ((-ax_i - ax_E)*np.sin(phi_i) + (ay_i - ay_E)*np.cos(phi_i))/REerde
    return alpha_i

def eq_motion_4body(t, state, mass, calculate_tides=True):
    '''This function calculates the derivatives of the state vector for a specific 4 body problem (Earth-Moon with tidal force)
    state: state vector is given by [x_E, y_E, x_M, y_M, phi_1, phi_2, vx_E, vy_E, vx_M, vy_M, omega_1, omega_2].
    mass: list of the masses of the two bodys. e.g. [m_E, m_M, m1, m2]'''
    x_E, y_E, x_M, y_M, phi_1, phi_2, vx_E, vy_E, vx_M, vy_M, omega_1, omega_2 = state
    m_E, m_M, m_1, m_2 = mass

    # Variables as defined in the explanation above
    x_1, y_1, x_2, y_2 = x_E + REerde*np.cos(phi_1), y_E + REerde*np.sin(phi_1), x_E + REerde*np.cos(phi_2), y_E + REerde*np.sin(phi_2)
    mu_x = m_E + m_1*np.cos(phi_1)**2 + m_2*np.cos(phi_2)**2
    mu_y = m_E + m_1*np.sin(phi_1)**2 + m_2*np.sin(phi_2)**2
    a = (G*m_M*m_E)/((x_M - x_E)**2 + (y_M - y_E)**2)**(3/2)
    b = m_1*np.sin(phi_1)*np.cos(phi_1) + m_2*np.sin(phi_2)*np.cos(phi_2)
    c1 = m_1*REerde*omega_1**2 + (G*m_M*m_1*((x_M - x_1)*np.cos(phi_1) + (y_M - y_1)*np.sin(phi_1)))/((x_M - x_1)**2 + (y_M - y_1)**2)
    c2 = m_2*REerde*omega_2**2 + (G*m_M*m_2*((x_M - x_2)*np.cos(phi_2) + (y_M - y_2)*np.sin(phi_2)))/((x_M - x_2)**2 + (y_M - y_2)**2)

    # The earth's acceleration:
    ax_E = 1/(mu_x - b**2/mu_y) * (a*(x_M - x_E) - b/mu_y*(a*(y_M - y_E) + c1*np.sin(phi_1) + c2*np.sin(phi_2)) + c1*np.cos(phi_1) - c2*np.cos(phi_2))
    ay_E = 1/mu_y*(a*(y_M - y_E) - b*ax_E + c1*np.sin(phi_1) + c2*np.sin(phi_2))

    # The moon's acceleration:
    ax_M = G*((m_E*(x_E - x_M))/((x_E - x_M)**2 + (y_E - y_M)**2)**(3/2) + (m_1*(x_1 - x_M))/((x_1 - x_M)**2 + (y_1 - y_M)**2))
    ay_M = G*((m_E*(y_E - y_M))/((x_E - x_M)**2 + (y_E - y_M)**2)**(3/2) + (m_1*(y_1 - y_M))/((x_1 - x_M)**2 + (y_1 - y_M)**2))

    if not calculate_tides: return [vx_E, vy_E, vx_M, vy_M, omega_1, omega_2, ax_E, ay_E, ax_M, ay_M, 0, 0] # For code reuse we can reuse the same code for both cases

    # The tide's angular acceleration:
    alpha_1 = tide_acceleration(x_M, y_M, x_1, y_1, phi_1, ax_E, ay_E, omega_1, m_M)
    alpha_2 = tide_acceleration(x_M, y_M, x_2, y_2, phi_2, ax_E, ay_E, omega_2, m_M)

    return[vx_E, vy_E, vx_M, vy_M, omega_1, omega_2, ax_E, ay_E, ax_M, ay_M, alpha_1, alpha_2]

```

Die Anfangsbedingungen müssen auf das erweiterte System angepasst werden. Wir teilen die Masse des Ozeans gleichmäßig auf die beiden Flutberge auf und lassen die beiden Flutberge auf der Verbindungsgeraden zwischen Erde und Mond mit der Umlaufperiode des Mondes starten. Wir berechnen den Schwerpunkt des Systems und legen ihn in den Ursprung. Da die Flutberge auf gegenüberliegenden Seiten der Erde sind spielt ihr Abstand hier keine Rolle; ihre Masse könnte auch einfach zur Erde dazu addiert werden.

```
In [ ]: def center_of_mass(mass, position_x , position_y):
    '''Returns the coordinates of the center of mass of the system'''
    x = np.sum(mass*position_x)/np.sum(mass)
    y = np.sum(mass*position_y)/np.sum(mass)
    return [x, y]

def iv_stable_orbit_4body():
    '''Initial conditions for the Earth-Moon-Tides system in a stable orbit'''
    mass = [mErde, mMond, mOzean/2, mOzean/2] # Each of the tides gets half of the ocean's mass

    # We want the center of mass to be at the origin, the tides count towards the earth
    abstand_baryzentrum_erde = center_of_mass(mass, np.array([0, rMondBahn, -REerde, REerde]), np.array([0, 0, 0, 0]))[0]
    x0_Eerde = [-abstand_baryzentrum_erde, 0]
    x0_Mond = [rMondBahn - abstand_baryzentrum_erde, 0]

    # We choose the starting velocities such that the center of mass is at rest
    vy0_Eerde = 2*pi*abstand_baryzentrum_erde/TMondBahn
    vy0_Mond = 2*pi*(rMondBahn-abstand_baryzentrum_erde)/TMondBahn

    # The moon starts in positive, the earth in negative y-direction
    v0_Eerde = [0, -vy0_Eerde]
    v0_Mond = [0, vy0_Mond]

    # The tides start on the x-axis on opposite sides of the earth with the angular velocity of the moon
    phi_1, phi_2 = 0, pi
    omega_1, omega_2 = 2*pi/TMondBahn, 2*pi/TMondBahn

    return [x0_Eerde, x0_Mond, [phi_1, phi_2], v0_Eerde, v0_Mond, [omega_1, omega_2], mass]

def four_body_problem(pos_body_1: list, pos_body_2: list, phi_i: list, vel_body_1: list, vel_body_2: list, omega_i: list, mass
solution = solve_ivp(fun=fun, t_span=[t_start, t_max], y0=[*pos_body_1, *pos_body_2, *phi_i, *vel_body_1, *vel_body_2, *omega_i], t_eval=ts, x_E, y_E, x_M, y_M, phi_1, phi_2, vx_E, vy_E, vx_M, vy_M, omega_1, omega_2 = solution.y
return [solution.t, x_E, y_E, x_M, y_M, phi_1, phi_2, omega_1, omega_2]
```

Graphische Darstellung

Wir wollen die für sympy und per Hand berechneten Funktionen vergleichen, dazu plotten wir mehrere Phasenraumplots der beiden Simulationen.

```
In [ ]: # We execute the sympy and the direct solution to compare them
t, x_E, y_E, x_M, y_M, phi_1, phi_2, _, _ = four_body_problem(*iv_stable_orbit_4body(), 3*TMondBahn)

In [ ]: ts, x_ES, y_ES, x_MS, y_MS, phi_1S, phi_2S, _, _ = four_body_problem(*iv_stable_orbit_4body(), 3*TMondBahn, fun=eq_motion_4body)
```

```
In [ ]: # The tides' positions in cartesian coordinates
x_1, y_1, x_2, y_2 = x_E + RErde*np.cos(phi_1), y_E + RErde*np.sin(phi_1), x_E + RErde*np.cos(phi_2), y_E + RErde*np.sin(phi_2)
x_1S, y_1S, x_2S, y_2S = x_ES + RErde*np.cos(phi_1S), y_ES + RErde*np.sin(phi_1S), x_ES + RErde*np.cos(phi_2S), y_ES + RErde*np.sin(phi_2S)

# Plot the solution of the four body problem as a phase space diagram
fig, axes = plt.subplots(nrows=5, ncols=4, figsize=(12, 12)) # Create a figure and a set of subplots
fig.suptitle("Phasenraumdiagramme des Erde-Mond-Flutberg-Systems: per-Hand und mit Sympy")

def init_phase_space(ax, a, b, title, xlabel, ylabel, color):
    ax.plot(a, b, color=color)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_title(title)

init_phase_space(axes[0,0], t, x_E, "x-Position Erde", "$t$ / s", "$x_1$ / m", "blue")
init_phase_space(axes[0,1], t, x_M, "x-Position Mond", "$t$ / s", "$x_2$ / m", "red")
init_phase_space(axes[1,0], t, y_E, "y-Position Erde", "$t$ / s", "$y_1$ / m", "blue")
init_phase_space(axes[1,1], t, y_M, "y-Position Mond", "$t$ / s", "$y_2$ / m", "red")
init_phase_space(axes[2,0], x_E, y_E, "Rotation Erde", "$x_E$ / m", "$y_E$ / m", "blue")
init_phase_space(axes[2,1], x_M, y_M, "Rotation Mond", "$x_M$ / m", "$y_M$ / m", "red")
init_phase_space(axes[3,0], t, phi_1, "Winkel Flutberg 1", "$t$ / s", "$\phi_1$ / rad", "blue")
init_phase_space(axes[3,1], t, phi_2, "Winkel Flutberg 2", "$t$ / s", "$\phi_2$ / rad", "red")
init_phase_space(axes[4,0], x_1, y_1, "Rotation Flutberg 1", "$x_1$ / m", "$x_2$ / m", "blue")
init_phase_space(axes[4,1], x_2, y_2, "Rotation Flutberg 2", "$y_1$ / m", "$y_2$ / m", "red")

init_phase_space(axes[0,2], ts, x_ES, "x-Position Erde Sympy", "$t$ / s", "$x_1$ / m", "blue")
init_phase_space(axes[0,3], ts, x_MS, "x-Position Mond Sympy", "$t$ / s", "$x_2$ / m", "red")
init_phase_space(axes[1,2], ts, y_ES, "y-Position Erde Sympy", "$t$ / s", "$y_1$ / m", "blue")
init_phase_space(axes[1,3], ts, y_MS, "y-Position Mond Sympy", "$t$ / s", "$y_2$ / m", "red")
init_phase_space(axes[2,2], x_ES, y_ES, "Rotation Erde Sympy", "$x_E$ / m", "$y_E$ / m", "blue")
init_phase_space(axes[2,3], x_MS, y_MS, "Rotation Mond Sympy", "$x_M$ / m", "$y_M$ / m", "red")
init_phase_space(axes[3,2], ts, phi_1S, "Winkel Flutberg 1 Sympy", "$t$ / s", "$\phi_1$ / rad", "blue")
init_phase_space(axes[3,3], ts, phi_2S, "Winkel Flutberg 2 Sympy", "$t$ / s", "$\phi_2$ / rad", "red")
init_phase_space(axes[4,2], x_1S, y_1S, "Rotation Flutberg 1 Sympy", "$x_1$ / m", "$x_2$ / m", "blue")
init_phase_space(axes[4,3], x_2S, y_2S, "Rotation Flutberg 2 Sympy", "$y_1$ / m", "$y_2$ / m", "red")

plt.tight_layout() # Adjust the spacing between subplots
plt.show()
plt.close()

# Plot a 3d phase space diagram
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_title("3d-Plot des 4-Körper-Problems")
ax.plot(x_E, y_E, t, 'k', label='Erde')
ax.plot(x_M/10, y_M/10, t, 'r', label='Mond')
ax.plot(x_1, y_1, t, 'b', label='Flutberg 1')
ax.plot(x_2, y_2, t, 'g', label='Flutberg 2')
ax.set_xlabel("$x$ / m")
ax.set_ylabel("$y$ / m")
ax.set_zlabel("$t$ / s")
ax.legend()

plt.show()
plt.close()

plt.rc('axes', axisbelow=True)
plt.figure(figsize=(5, 5))
plt.title("Bahnen des 4-Körper-Problems")
plt.grid('minor', 'minor', linestyle='-', linewidth=0.2)
plt.grid('major', 'major', linestyle='-', linewidth=0.8)
plt.minorticks_on()
plt.xlabel("x")
plt.ylabel("y")

plt.plot(x_E, y_E, 'k', linewidth=1, label='Erde')
plt.plot(x_M/10, y_M/10, 'r', linewidth=1, label='Mond (/10)')
plt.plot(x_1, y_1, 'b', linewidth=1, label='Flutberg 1')
plt.plot(x_2, y_2, 'g', linewidth=1, label='Flutberg 2')

plt.legend()
plt.show()
plt.close()

# Animation of the four body problem
if not fastExecution: # Don't render if not necessary
    fig = plt.figure(figsize=(5, 5)) # Square figure, so the circles are actually circles
    fig.suptitle("Animation des Vier-Körper-Problems")
    ax = plt.axes(xlim=(-4e8, 4e8), ylim=(-4e8, 4e8))
    earth_line, = ax.plot([], [], marker='o', lw=0.1, color='blue', label='Erde (*10)')
    moon_line, = ax.plot([], [], marker='o', lw=0.1, color='red', label='Mond')
    tide_1_line, = ax.plot([], [], marker='o', lw=0.1, color='green', label='Flutberg 1 (*10)')
    tide_2_line, = ax.plot([], [], marker='o', lw=0.1, color='violet', label='Flutberg 2 (*10)')
    ax.legend()

    def animate(i):
        earth_line.set_data([x_E[i]*10], [y_E[i]*10])

```

```

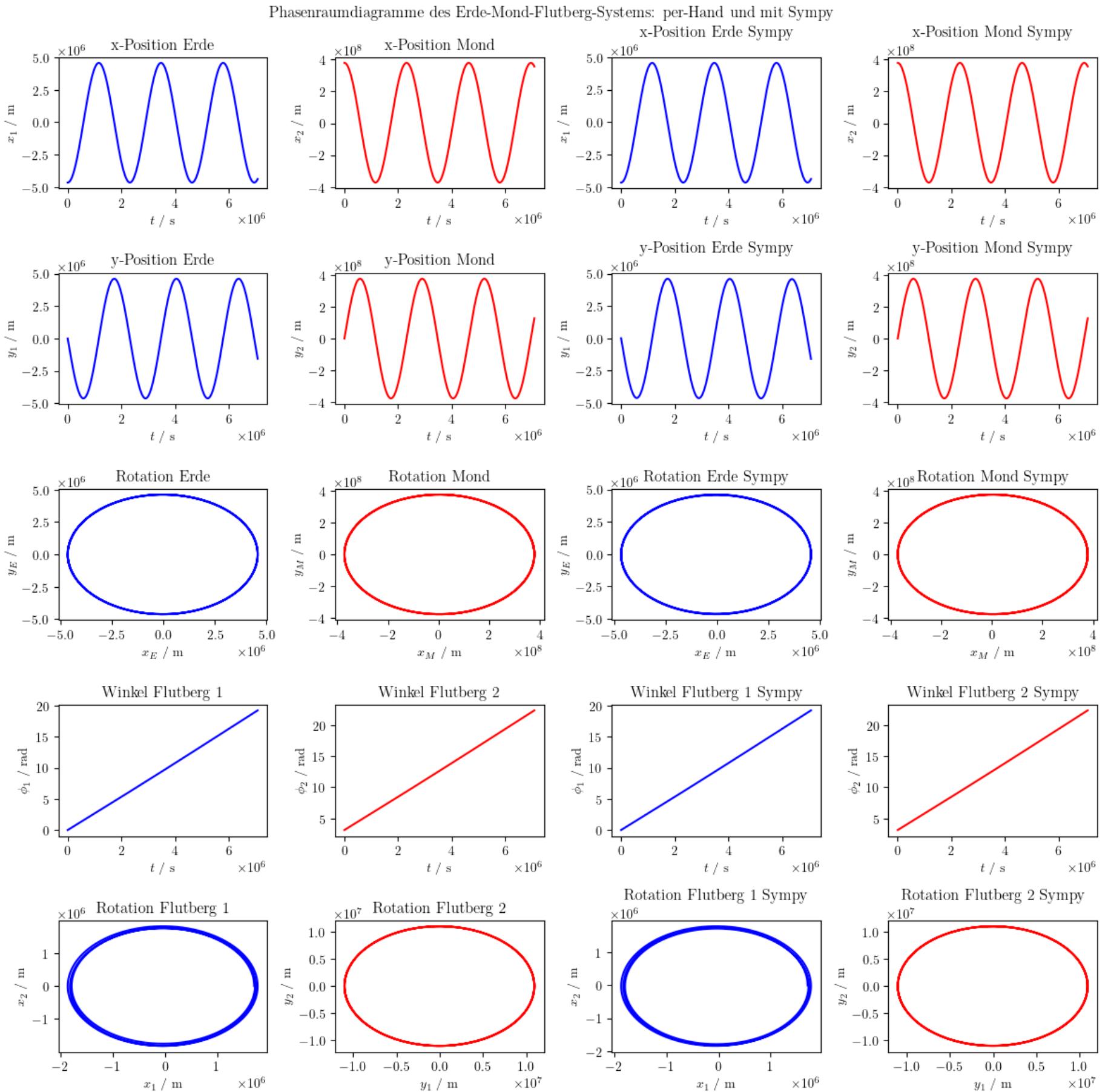
    moon_line.set_data([x_M[i]], [y_M[i]])
    tide_1_line.set_data([x_1[i]*10], [y_1[i]*10])
    tide_2_line.set_data([x_2[i]*10], [y_2[i]*10])
    return earth_line, moon_line

anim = FuncAnimation(fig, animate, init_func=None, frames=t.shape[0], interval=30, blit=True)
anim.save('Erde_Mond_Flutberge.gif', writer='pillow')
plt.close()

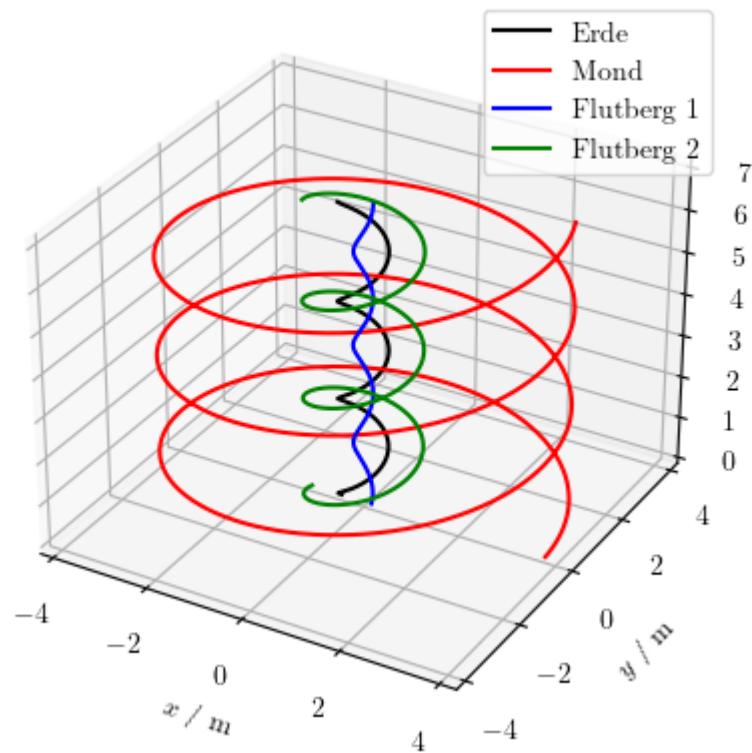
display(Image(data=open('Erde_Mond_Flutberge.gif','rb').read(), format='png'))

# Compare the numerical output of the two methods
state = [123, 456, 789, 101112, 131415, 161718, 192021, 12422, 2348, 34142, 23455, 56463]
print(f'Ausgabe der per-Hand-Methode: {eq_motion_4body(0, state, [mErde, mMond, mOzean/2, mOzean/2])}')
print(f'Ausgabe der Sympy-Methode: {eq_motion_4body_sympy(0, state, [mErde, mMond, mOzean/2, mOzean/2])}')

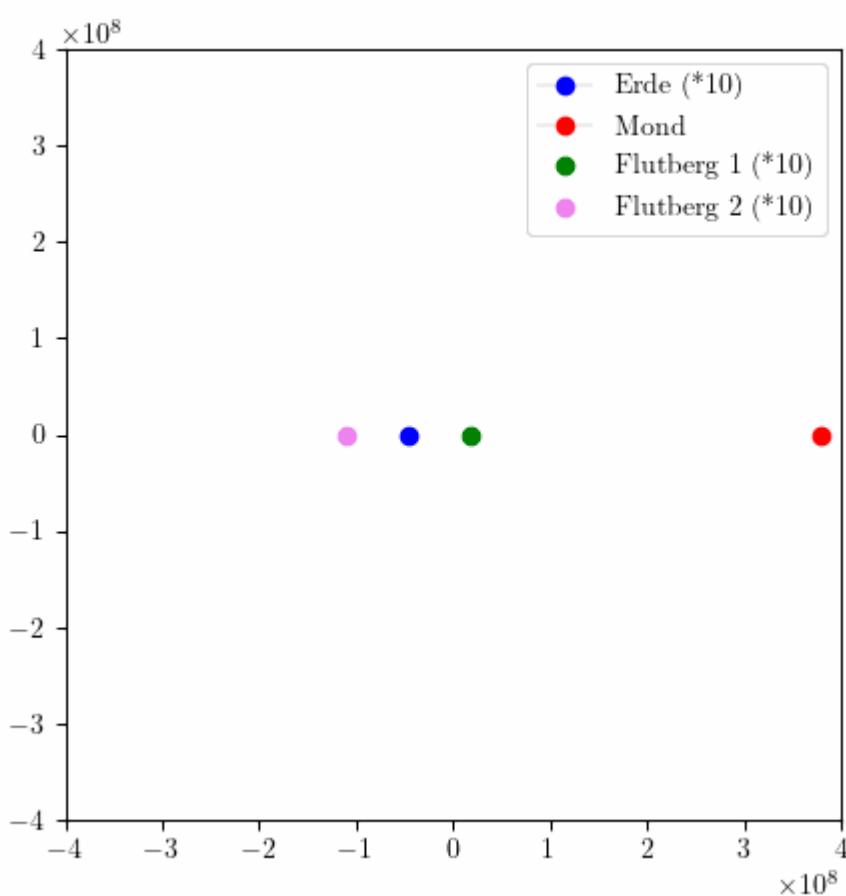
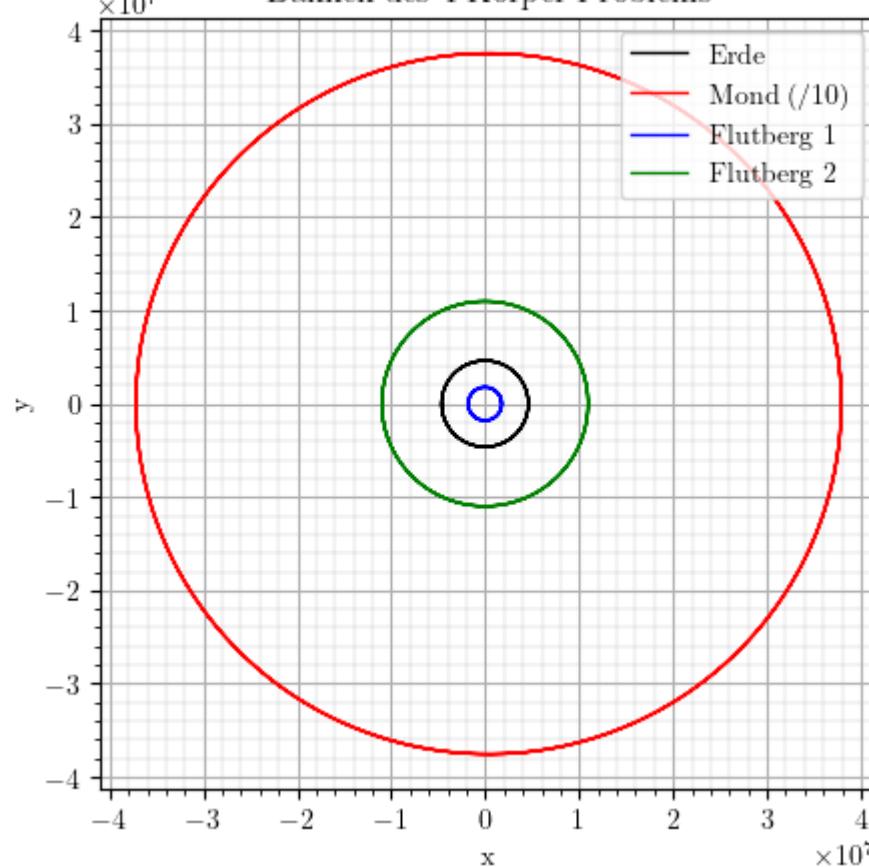
```



3d-Plot des 4-Körper-Problems



Bahnen des 4-Körper-Problems



Ausgabe der per-Hand-Methode: [192021, 12422, 2348, 34142, 23455, 56463, 224619815116.04947, 2671003691970.6772, -260.295958864 6813, -39339.79448240955, 268732.74115708674, -46351.59623989007]

Ausgabe der Sympy-Methode: [192021, 12422, 2348, 34142, 23455, 56463, 224619815116.04947, 2671003691970.677, -260.2959588646 814, -39339.79448240956, 268732.7411570847, -46351.59623992443]

Überprüfung

An den Laufzeiten der beiden Zellen für die Simulation sieht man noch einmal deutlich den Unterschied zwischen den beiden Methoden.

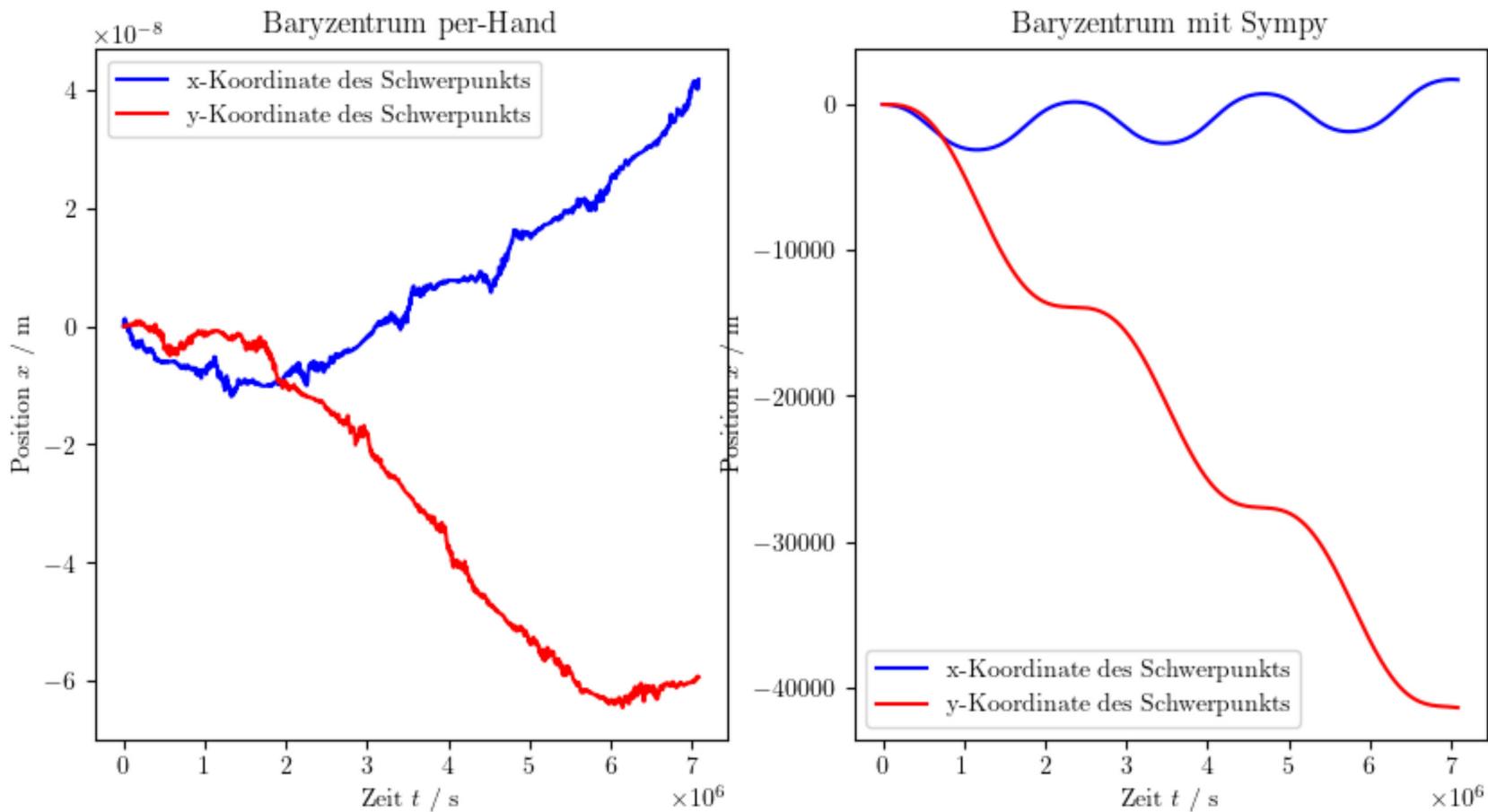
Wie in den Graphen unschwer zu erkennen ist, liefern die beiden Methoden die gleichen Ergebnisse. Am numerischen Output erkennt man aber, dass es kleine Abweichungen gibt. Auf den im Bild erkennbaren Größenordnungen sieht es gut aus, um den Unterschied zu quantifizieren wollen wir uns erneut das Baryzentrum anschauen.

```
In [ ]: def center_of_mass(mass, position_x , position_y):
    '''Returns the coordinates of the center of mass of the system'''
    x = np.sum(mass * position_x, axis=1) / np.sum(mass)
    y = np.sum(mass * position_y, axis=1) / np.sum(mass)
    return [x, y]

# Plot the change of the center of mass over time
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))
fig.suptitle("Bewegung des Baryzentrums des Erde-Mond-Flutberg-Systems")
axes[0].plot(t, center_of_mass([mErde, mMond, mOzean/2, mOzean/2], np.array([x_E, x_M, x_1, x_2]).T, np.array([y_E, y_M, y_1, y_2]).T), np.array([x_ES, x_MS, x_1S, x_2S]).T, np.array([y_ES, y_MS, y_1S, y_2S]).T, np.array([x_ES, x_MS, x_1S, x_2S]).T, np.array([y_ES, y_MS, y_1S, y_2S]).T)
axes[0].set_xlabel('Zeit $t$ / s')
axes[0].set_ylabel('Position $x$ / m')
axes[0].set_title('Baryzentrum per-Hand')
axes[0].legend()
axes[1].plot(tS, center_of_mass([mErde, mMond, mOzean/2, mOzean/2], np.array([x_E, x_M, x_1, x_2]).T, np.array([y_E, y_M, y_1, y_2]).T), np.array([x_ES, x_MS, x_1S, x_2S]).T, np.array([y_ES, y_MS, y_1S, y_2S]).T, np.array([x_ES, x_MS, x_1S, x_2S]).T, np.array([y_ES, y_MS, y_1S, y_2S]).T)
axes[1].set_xlabel('Zeit $t$ / s')
axes[1].set_ylabel('Position $x$ / m')
axes[1].set_title('Baryzentrum mit Sympy')
axes[1].legend()

plt.show()
plt.close()
```

Bewegung des Baryzentrums des Erde-Mond-Flutberg-Systems



SymPy macht deutlich mehr Fehler als die per Hand berechneten Funktionen, aus den bereits oben genannten Gründen.

Vierkörperproblem mit Randbedingungen, intrinsischer Rotation und Reibung

Im Folgenden wollen wir das 4-Körper-Problem um die Reibung des Ozeans (der Flutberge) mit der Erde erweitern. Dies verlangsamt die intrinsische Erdrotation und sorgt für länger werdende Tage und eine größere Umlaufbahn des Mondes. Um dies zu beschreiben müssen wir mehrere zusätzliche Komponenten herleiten und mit ihnen unser Modell erweitern.

Die Reibung beschreiben wir zunächst mit der Reibungskonstante $k = 2 \cdot 10^{-12} \frac{1}{\text{m}}$.

Herleitung der zusätzlichen Bewegungsgleichung und der Reibungskraft

Wir wollen als erstes die Reibungskraft zwischen der Erde und den Flutbergen bestimmen. Wir wählen dafür folgenden Ansatz:
 $\vec{F}_{R,i}(\vec{v}_i) = -km_i|\vec{v}_i|\vec{v}_i$. Hierbei ist \vec{v}_i die Geschwindigkeit des i -ten Flutbergs relativ zur rotierenden Erdoberfläche und k eine effektive Reibungskonstante.

Die Geschwindigkeit des Flutbergs bestimmen wir über seine Winkelgeschwindigkeit relativ zur Erdoberfläche ($\omega_E - \omega_i$), mit der er sich um die z-Achse dreht. Hierbei ist ω_E die Winkelgeschwindigkeit der Eigenrotation der Erde und ω_i die Winkelgeschwindigkeit des Flutbergs.

Die Geschwindigkeit ist allgemein definiert als $\vec{v}_i = \vec{\omega}_i \times \vec{r}_i$:

$$\vec{v}_i = (\omega_E - \omega_i) \cdot \vec{e}_z \times r_E \cdot \vec{e}_{r,i} = (\omega_E - \omega_i) \vec{e}_z \times (r_E \cdot \cos(\phi_i) \vec{e}_x + r_E \cdot \sin(\phi_i) \vec{e}_y) = -r_E \cdot \sin(\phi_i) \cdot (\omega_E - \omega_i) \vec{e}_x + r_E \cdot \cos(\phi_i) \cdot (\omega_E - \omega_i) \vec{e}_y$$

Damit berechnen wir den Betrag:

$$|\vec{v}_i| = \sqrt{r_E^2 \cdot (\omega_E - \omega_i)^2 \cdot \sin^2(\phi_i) + r_E^2 \cdot (\omega_E - \omega_i)^2 \cdot \cos^2(\phi_i)} = r_E \cdot |\omega_E - \omega_i|$$

Also ist die Reibungskraft:

$$\vec{F}_{R,i}(\vec{v}_i) = -km_i|\vec{v}_i|\vec{v}_i = -km_i r_E \cdot |\omega_E - \omega_i| \cdot -r_E \cdot \sin(\phi_i) \cdot (\omega_E - \omega_i) \vec{e}_x + r_E \cdot \cos(\phi_i) \cdot (\omega_E - \omega_i) \vec{e}_y = -km_i r_E^2 \cdot |\omega_E - \omega_i| (\omega_E - \omega_i) \cdot$$

Durch teilen durch die Masse des i-ten Flutbergs erhalten wir die Beschleunigung, die wir auf die Differentialgleichung des i-ten Flutbergs aus dem 4-Körper-Problem addieren.

Als nächstes wollen wir die Differentialgleichung für die intrinsische Rotation der Erde herleiten. Es gilt die Drehimpulsrelation $L = I \cdot \omega$. Wir nehmen die Erde als Kugel mit homogener Massenverteilung an, damit hat sie ein Trägheitsmoment von $I = \frac{2}{5}m_E r_E^2$. Für Drehimpuls und Drehmoment folgt:

$$\begin{aligned}\vec{L} &= \frac{2}{5}m_E r_E^2 \vec{\omega}_E \\ \vec{M} &= \frac{d\vec{L}}{dt} = \frac{2}{5}m_E r_E^2 \dot{\vec{\omega}}_E\end{aligned}$$

Das Drehmoment welches durch die Reibungskräfte entsteht ist:

$$\vec{M}_R = \sum_{i=1}^2 \vec{r}_i \times \vec{F}_{R,i}$$

Dies berechnet sich für zwei Flutberge zu:

$$\vec{M}_R = -kr_E^3 (m_1 \cdot |\omega_E - \omega_1| (\omega_E - \omega_1) + m_2 \cdot |\omega_E - \omega_2| (\omega_E - \omega_2)) \vec{e}_z$$

Setzen wir die z-Komponenten der Drehmomente gleich (nur diese sind $\neq 0$), erhalten wir folgende Bewegungsgleichung für die intrinsische Rotation der Erde:

$$\begin{aligned}\frac{2}{5}m_E r_E^2 \dot{\omega}_E &= -kr_E^3 (m_1 \cdot |\omega_E - \omega_1| (\omega_E - \omega_1) + m_2 \cdot |\omega_E - \omega_2| (\omega_E - \omega_2)) \\ \dot{\omega}_E &= -\frac{5kr_E}{2m_E} (m_1 \cdot |\omega_E - \omega_1| (\omega_E - \omega_1) + m_2 \cdot |\omega_E - \omega_2| (\omega_E - \omega_2))\end{aligned}$$

Um dies im Code umzusetzen verwenden wir, wenn es geht die schon implementierten Funktionen. Daher wird hier wie zuvor vorgegangen, nur die Beschleunigung auf die Flutberge wird wie oben erklärt erweitert, und die zusätzliche Differentialgleichung wird hinzugefügt

```
In [ ]: # redefining some functions, otherwise the later code will not work
def center_of_mass(mass, position_x, position_y):
    return [np.sum(mass*position_x)/np.sum(mass), np.sum(mass*position_y)/np.sum(mass)]

def iv_stable_orbit_4body():
    mass = [mErde, mMond, mOzean/2, mOzean/2] # Each of the tides gets half of the ocean's mass
    abstand_baryzentrum_erde = center_of_mass(mass, np.array([0, rMondBahn, -REerde, REerde]), np.array([0, 0, 0, 0]))[0]
    x0_Erde = [-abstand_baryzentrum_erde, 0]
    x0_Mond = [rMondBahn - abstand_baryzentrum_erde, 0]
    vy0_Erde = 2*pi*abstand_baryzentrum_erde/TMondBahn
    vy0_Mond = 2*pi*(rMondBahn-abstand_baryzentrum_erde)/TMondBahn
    v0_Erde = [0, -vy0_Erde]
    v0_Mond = [0, vy0_Mond]
    phi_1, phi_2 = 0, pi
    omega_1, omega_2 = 2*pi/TMondBahn, 2*pi/TMondBahn
    return [x0_Erde, x0_Mond, [phi_1, phi_2], v0_Erde, v0_Mond, [omega_1, omega_2], mass]

def four_body_problem(pos_body_1: list, pos_body_2: list, phi_i: list, vel_body_1: list, vel_body_2: list, omega_i: list, mass
    solution = solve_ivp(fun=fun, t_span=[t_start, t_max], y0=[*pos_body_1, *pos_body_2, *phi_i, *vel_body_1, *vel_body_2, *omega_i], x_E, y_E, x_M, y_M, phi_1, phi_2, vx_E, vy_E, vx_M, vy_M, omega_1, omega_2 = solution.y
    return [solution.t, x_E, y_E, x_M, y_M, phi_1, phi_2, omega_1, omega_2]
```

```
In [ ]: def iv_stable_orbit_4body_friction():
    '''Initial conditions for the Earth-Moon-Tides system in a stable orbit'''
    x0_Eerde, x0_Mond, phi_i, v0_Eerde, v0_Mond, omega_i, mass = iv_stable_orbit_4body()
    phi0_E = 0 # The starting point of the earth's rotation is unconsequentially defined as 0
    omega0_E = 2*pi / TErdRotation # Angular velocity of the earth
    return [x0_Eerde, x0_Mond, phi_i, phi0_E, v0_Eerde, v0_Mond, omega_i, omega0_E, mass]

def tides_acceleration_friction(x_M, y_M, x_i, y_i, phi_i, ax_E, ay_E, omega_i, omega_E, m_M, k):
    '''Calculates the angular acceleration of a tide, adding the frictional force caused by the earth's rotation'''
    ax_friction = k * RErde**2 * np.abs(omega_i - omega_E) * (omega_i - omega_E) * np.sin(phi_i)
    ay_friction = -k * RErde**2 * np.abs(omega_i - omega_E) * (omega_i - omega_E) * np.cos(phi_i)
    part_a = G*m_M* ((y_M - y_i)*np.cos(phi_i) - (x_M - x_i)*np.sin(phi_i)) / ((x_M - x_i)**2 + (y_M - y_i)**2)**(3/2)
    part_b = (REerde*omega_i**2 - (ax_E*np.cos(phi_i) + ay_E*np.sin(phi_i)))
    ax_i = -part_a*np.sin(phi_i) - part_b*np.cos(phi_i) + ax_friction
    ay_i = part_a*np.cos(phi_i) - part_b*np.sin(phi_i) + ay_friction

    alpha_i = (-(ax_i - ax_E)*np.sin(phi_i) + (ay_i - ay_E)*np.cos(phi_i))/REerde
    return alpha_i

def eq_motion_4body_friction(t, state, mass, k):
    x_E, y_E, x_M, y_M, phi_1, phi_2, phi_E, vx_E, vy_E, vx_M, vy_M, omega_1, omega_2, omega_E = state
    x_1, y_1, x_2, y_2 = x_E + RErde*np.cos(phi_1), y_E + RErde*np.sin(phi_1), x_E + RErde*np.cos(phi_2), y_E + RErde*np.sin(phi_2)
    m_E, m_M, m_1, m_2 = mass

    # Reuse the equations of motion from the 4-body problem for the earth and the moon:
    _, _, _, _, _, ax_E, ay_E, ax_M, ay_M, _, _ = eq_motion_4body(t, np.array([x_E, y_E, x_M, y_M, phi_1, phi_2, vx_E, vy_E, vx_M, vy_M, omega_1, omega_2, omega_E]), mass)

    # Tides angular acceleration with friction:
    alpha_1 = tides_acceleration_friction(x_M, y_M, x_1, y_1, phi_1, ax_E, ay_E, omega_1, omega_E, m_M, k)
    alpha_2 = tides_acceleration_friction(x_M, y_M, x_2, y_2, phi_2, ax_E, ay_E, omega_2, omega_E, m_M, k)

    # The earth's angular acceleration:
    alpha_E = (5*k*REerde)/(2*m_E) * (m_1*np.abs(omega_1 - omega_E)*(omega_1 - omega_E) + m_2*np.abs(omega_2 - omega_E)*(omega_2 - omega_E))

    return[vx_E, vy_E, vx_M, vy_M, omega_1, omega_2, omega_E, ax_E, ay_E, ax_M, ay_M, alpha_1, alpha_2, alpha_E]

def four_body_problem_friction(pos_body_1: list, pos_body_2: list, phi_i: list, phi_E: float, vel_body_1: list, vel_body_2: list):
    solution = solve_ivp(fun=eq_motion_4body_friction, t_span=[t_start, t_max], y0=[*pos_body_1, *pos_body_2, *phi_i, phi_E, *x_E, y_E, x_M, y_M, phi_1, phi_2, phi_E, vx_E, vy_E, vx_M, vy_M, omega_1, omega_2, omega_E])
    return [solution.t, x_E, y_E, x_M, y_M, phi_1, phi_2, phi_E, omega_1, omega_2, omega_E]
```

Graphische Darstellung und Vergleich zu ohne Reibung

```
In [ ]: if not fastExecution:
    # 4 body without friction:
    t, x_E, y_E, x_M, y_M, phi_1, phi_2, _, _ = four_body_problem(*iv_stable_orbit_4body(), 0.75*TMondBahn, rtol=1e-12, atol=1
    x_1, y_1, x_2, y_2 = x_E + RErde*np.cos(phi_1), y_E + RErde*np.sin(phi_1), x_E + RErde*np.cos(phi_2), y_E + RErde*np.sin(phi_2)
    t_1, x_E_1, y_E_1, x_M_1, y_M_1, phi_1_1, phi_2_1, omega_1_1, omega_2_1 = four_body_problem(*iv_stable_orbit_4body(), 100*
    x_1_1, y_1_1, x_2_1, y_2_1 = x_E_1 + RErde*np.cos(phi_1_1), y_E_1 + RErde*np.sin(phi_1_1), x_E_1 + RErde*np.cos(phi_2_1),
    # 4 body with friction short time: 0.3s
    t_f, x_E_f, y_E_f, x_M_f, y_M_f, phi_1_f, phi_2_f, phi_E_f, omega_1_f, omega_2_f, omega_E_f = four_body_problem_friction(*
    x_1_f, y_1_f, x_2_f, y_2_f = x_E_f + RErde*np.cos(phi_1_f), y_E_f + RErde*np.sin(phi_1_f), x_E_f + RErde*np.cos(phi_2_f),
    # 4 body with friction for 100 years: 11min
    t_f1, x_E_f1, y_E_f1, x_M_f1, y_M_f1, phi_1_f1, phi_2_f1, phi_E_f1, omega_1_f1, omega_2_f1, omega_E_f1 = four_body_problem(
    x_1_f1, y_1_f1, x_2_f1, y_2_f1 = x_E_f1 + RErde*np.cos(phi_1_f1), y_E_f1 + RErde*np.sin(phi_1_f1), x_E_f1 + RErde*np.cos(phi_2_f1)
```

```
In [ ]: if not fastExecution:
    # Plot the solution of the four body problem as a phase space diagram
    fig, axes = plt.subplots(nrows=5, ncols=4, figsize=(12, 12)) # Create a figure and a set of subplots
    fig.suptitle("Phasenraumdiagramme des Erde-Mond-Flutberg-Systems: ohne Reibung und mit Reibung")

    def init_phase_space(ax, a, b, title, xlabel, ylabel, color):
        ax.plot(a, b, color=color)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.set_title(title)

    init_phase_space(axes[0,0], t, x_E, "x-Position Erde", "$t$ / s", "$x_1$ / m", "blue")
    init_phase_space(axes[0,1], t, x_M, "x-Position Mond", "$t$ / s", "$x_2$ / m", "red")
    init_phase_space(axes[1,0], t, y_E, "y-Position Erde", "$t$ / s", "$y_1$ / m", "blue")
    init_phase_space(axes[1,1], t, y_M, "y-Position Mond", "$t$ / s", "$y_2$ / m", "red")
    init_phase_space(axes[2,0], x_E, y_E, "Rotation Erde", "$x_E$ / m", "$y_E$ / m", "blue")
    init_phase_space(axes[2,1], x_M, y_M, "Rotation Mond", "$x_M$ / m", "$y_M$ / m", "red")
    init_phase_space(axes[3,0], t, phi_1, "Winkel Flutberg 1", "$t$ / s", "$\phi_1$ / rad", "blue")
    init_phase_space(axes[3,1], t, phi_2, "Winkel Flutberg 2", "$t$ / s", "$\phi_2$ / rad", "red")
    init_phase_space(axes[4,0], x_1, y_1, "Rotation Flutberg 1", "$x_1$ / m", "$x_2$ / m", "blue")
    init_phase_space(axes[4,1], x_2, y_2, "Rotation Flutberg 2", "$y_1$ / m", "$y_2$ / m", "red")

    init_phase_space(axes[0,2], t_f, x_E_f, "x-Position Erde mit Reibung", "$t$ / s", "$x_1$ / m", "blue")
    init_phase_space(axes[0,3], t_f, x_M_f, "x-Position Mond mit Reibung", "$t$ / s", "$x_2$ / m", "red")
    init_phase_space(axes[1,2], t_f, y_E_f, "y-Position Erde mit Reibung", "$t$ / s", "$y_1$ / m", "blue")
    init_phase_space(axes[1,3], t_f, y_M_f, "y-Position Mond mit Reibung", "$t$ / s", "$y_2$ / m", "red")
    init_phase_space(axes[2,2], x_E_f, y_E_f, "Rotation Erde mit Reibung", "$x_E$ / m", "$y_E$ / m", "blue")
    init_phase_space(axes[2,3], x_M_f, y_M_f, "Rotation Mond mit Reibung", "$x_M$ / m", "$y_M$ / m", "red")
    init_phase_space(axes[3,2], t_f, phi_1_f, "Winkel Flutberg 1 mit Reibung", "$t$ / s", "$\phi_1$ / rad", "blue")
    init_phase_space(axes[3,3], t_f, phi_2_f, "Winkel Flutberg 2 mit Reibung", "$t$ / s", "$\phi_2$ / rad", "red")
    init_phase_space(axes[4,2], x_1_f, y_1_f, "Rotation Flutberg 1 mit Reibung", "$x_1$ / m", "$x_2$ / m", "blue")
    init_phase_space(axes[4,3], x_2_f, y_2_f, "Rotation Flutberg 2 mit Reibung", "$y_1$ / m", "$y_2$ / m", "red")

    plt.tight_layout()
    plt.savefig("4body_friction_comp0.png", dpi=300)
    plt.close()

    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10))
    plt.rc('axes', axisbelow=True)
    plt.suptitle("Vergleich 4 Körper Problem mit Reibung / ohne Reibung")

    def plot_subplot(ax, lines, title, xlabel, ylabel):
        for x, y, label, color in lines:
            ax.plot(x, y, color, linewidth=1, label=label)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.set_title(title)

        ax.grid('minor', 'minor', linestyle='-', linewidth=0.2)
        ax.grid('major', 'major', linestyle='-', linewidth=0.8)
        ax.minorticks_on()
        ax.legend()

    # Distance from the origin for earth and moon
    r_Earth = np.sqrt(x_E_l**2 + y_E_l**2)
    r_Moon = np.sqrt(x_M_l**2 + y_M_l**2)
    r_Earth_f = np.sqrt(x_E_f1**2 + y_E_f1**2)
    r_Moon_f = np.sqrt(x_M_f1**2 + y_M_f1**2)

    plot_subplot(axes[0][0], [(x_E_l*10, y_E_l*10, 'Erde (*10)', 'k'), (x_M_l, y_M_l, 'Mond', 'r'), (x_1_l*10, y_1_l*10, 'Flutberg 1', 'k'), (x_2_l*10, y_2_l*10, 'Flutberg 2', 'r')], "Abstand des Mondes zum Ursprung ohne Reibung", "t / s", "r / m")
    plot_subplot(axes[0][1], [(t_l, r_Moon, 'Mond', 'r')], "Abstand der Erde zum Ursprung ohne Reibung", "t / s", "r / m")
    plot_subplot(axes[0][2], [(t_l, r_Earth, 'Erde', 'k')], "Abstand des Mondes zum Ursprung mit Reibung", "t / s", "r / m")
    plot_subplot(axes[1][0], [(x_E_f1*10, y_E_f1*10, 'Erde (*10)', 'k'), (x_M_f1, y_M_f1, 'Mond', 'r'), (x_1_f1*10, y_1_f1*10, 'Flutberg 1', 'k'), (x_2_f1*10, y_2_f1*10, 'Flutberg 2', 'r')], "Abstand des Mondes zum Ursprung ohne Reibung", "t / s", "r / m")
    plot_subplot(axes[1][1], [(t_f1, r_Moon_f, 'Mond', 'r')], "Abstand der Erde zum Ursprung mit Reibung", "t / s", "r / m")
    plot_subplot(axes[1][2], [(t_f1, r_Earth_f, 'Erde', 'k')], "Abstand der Erde zum Ursprung mit Reibung", "t / s", "r / m")
    plt.tight_layout()
    plt.savefig("4body_friction_comp1.png", dpi=300)
    plt.close()

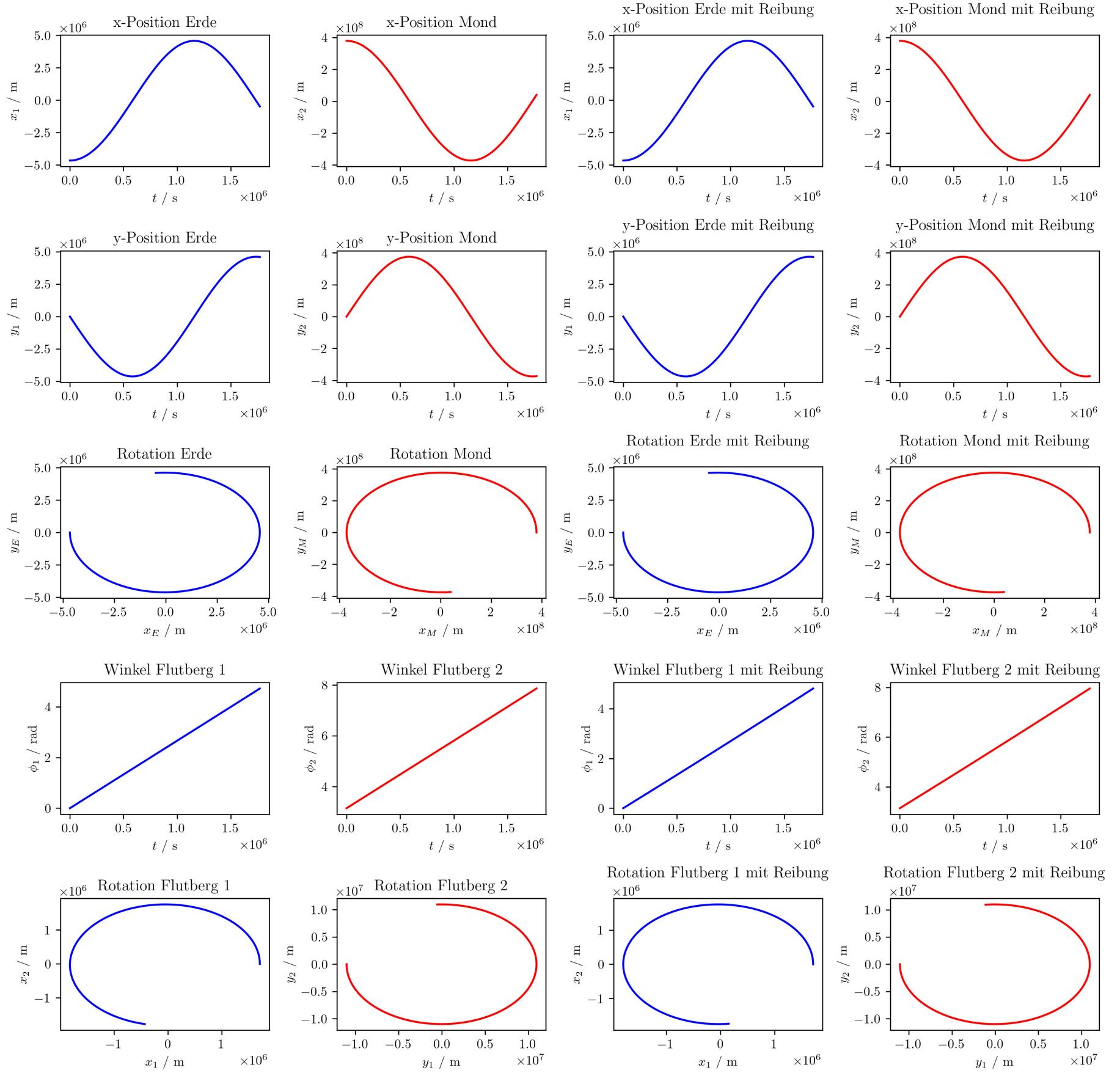
    fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 5))
    plt.rc('axes', axisbelow=True)
    plt.suptitle("Vergleich 4 Körper Problem mit Reibung / ohne Reibung")

    plot_subplot(axes[0][0], [(t_l, omega_1_l, '$\omega_1$ Flutberg 1', 'k'), (t_l, omega_2_l, '$\omega_2$ Flutberg 2', 'r'), (t_l, [2*pi/TErdRotation]*np.shape(t_l)[0], '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Flutberg 1 und 2 ohne Reibung", "t / s", "rad/s")
    plot_subplot(axes[0][1], [(t_f1, omega_1_f1, '$\omega_1$ Flutberg 1', 'k'), (t_f1, omega_2_f1, '$\omega_2$ Flutberg 2', 'r'), (t_f1, [2*pi/TErdRotation]*np.shape(t_f1)[0], '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Flutberg 1 und 2 mit Reibung", "t / s", "rad/s")
    plot_subplot(axes[1][0], [(t_f1, omega_E_f1, '$\omega_E$ Winkelgeschwindigkeit Erde', 'k'), (t_f1, [2*pi/TErdRotation]*np.shape(t_f1)[0], '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Erde ohne und mit Reibung", "t / s", "rad/s")
    plot_subplot(axes[1][1], [(t_f1, omega_E_f1, '$\omega_E$ Winkelgeschwindigkeit Erde', 'k'), (t_f1, [2*pi/TErdRotation]*np.shape(t_f1)[0], '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Erde ohne und mit Reibung", "t / s", "rad/s")

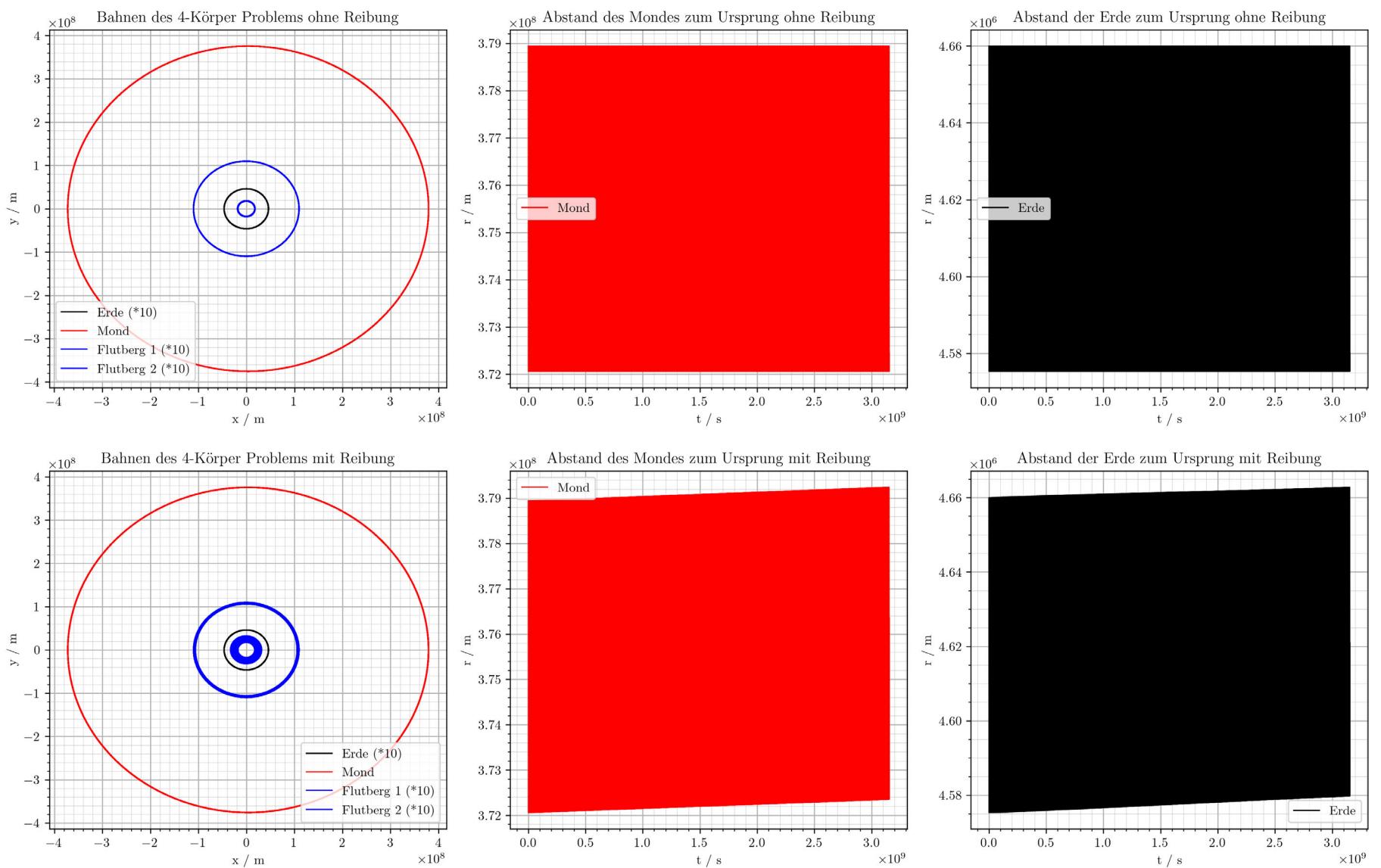
    plt.tight_layout()
    plt.savefig("4body_friction_comp2.png", dpi=300)
    plt.close()

    # Display image set the size to be 500x500 pixels
    display(Image(filename='4body_friction_comp0.png', width=1000, height=2000))
    display(Image(filename='4body_friction_comp1.png', width=1000, height=500))
    display(Image(filename='4body_friction_comp2.png', width=1000, height=500))
```

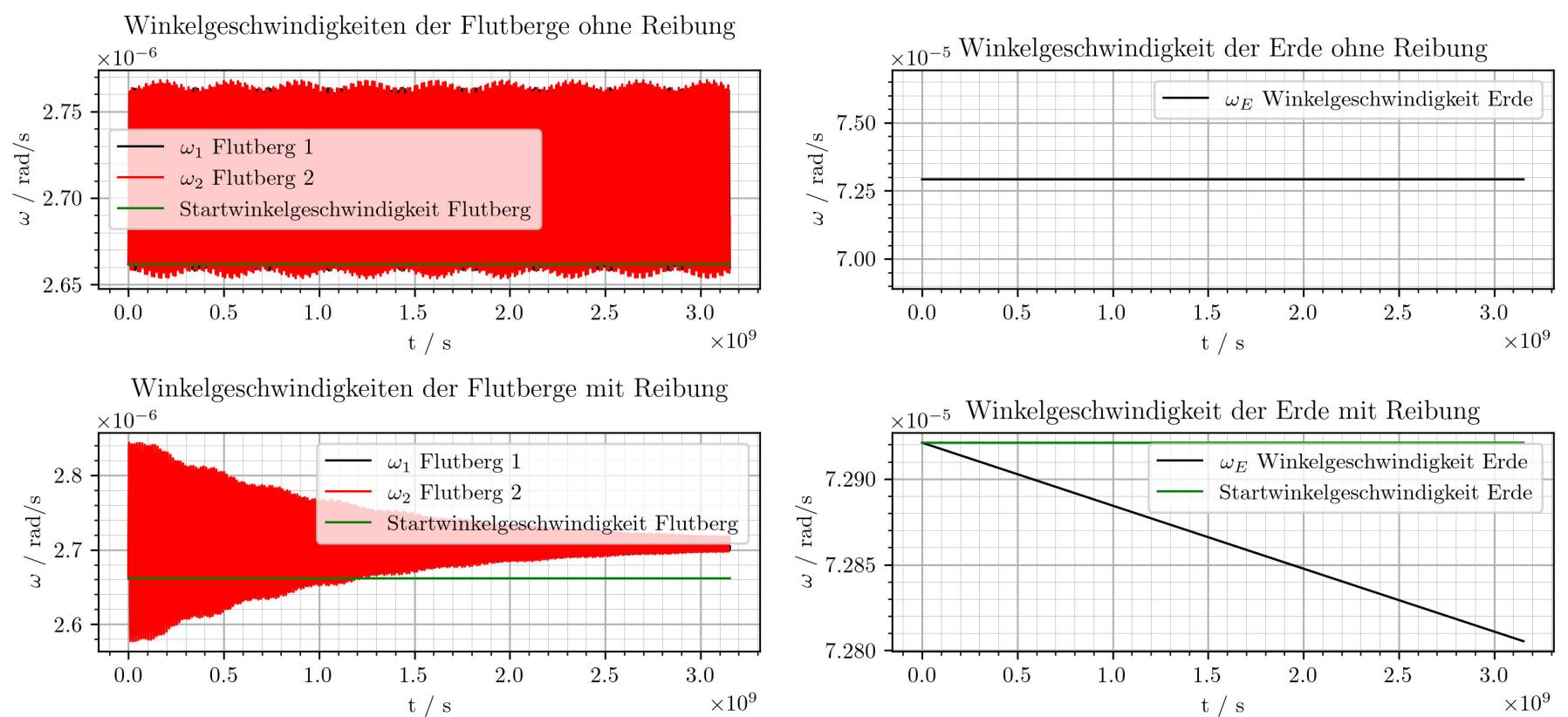
Phasenraumdiagramme des Erde-Mond-Flutberg-Systems: ohne Reibung und mit Reibung



Vergleich 4 Körper Problem mit Reibung / ohne Reibung



Vergleich 4 Körper Problem mit Reibung / ohne Reibung



Im Fall ohne Reibung ist ω_E natürlich konstant, und im Fall mit Reibung nimmt ω_E mit der Zeit ab, das heißt, dass die Tage auf der Erde länger werden. Hier sieht die Abnahme linear aus, es handelt sich aber um einen exponentiellen Abfall, welcher aufgrund der betrachteten Zeitskala aber linear erscheint.

Zudem ist zu sehen, dass Erde und Mond sich mit Reibung von ihrem gemeinsamen Schwerpunkt weg bewegen, was ohne Reibung nicht passiert. Das ist eine Folge davon, dass die Erdrotation schneller als die Umlaufbahn des Mondes ist. Die Flutberge werden aufgrund der Reibung vor die Verbindungslinie zwischen Erde und Mond gedrückt, was die Wirkung der Flutberge auf den Mond ändert. Der Mond erfährt eine Radialbeschleunigung, wodurch er auf eine weiter entfernte Bahn gelangt.

Vergleicht man die Winkelgeschwindigkeiten der Flutberge, so erkennt man, dass diese in beiden Fällen mit einer schwach modulierten Oszillation starten. Mit der Zeit schwächt sich dies mit Reibung ab, ohne Reibung bleibt der Effekt zeitlich gleich. Die Bewegung ist Folge davon, dass die Umlaufbahn nicht perfekt ein Kreis ist (Wie man auch in den Plots vom Abstand zum Ursprung erkennen kann), weshalb der Mond die Flutberge zu unterschiedlichen Zeitpunkten unterschiedlich stark beschleunigt. Mit Reibung wird diese Bewegung abgebremst (Energiedissipation), weshalb die Amplitude der Oszillation abnimmt. Die Amplitude ist zu Beginn jedoch größer, weil die Flutberge durch die Reibung in Richtung der Erdrotation beschleunigt werden.

Wir stellen noch dar, wie sich der Mond von der Erde entfernt:

In []: #Show the increase of the moons orbit

```

if not fastExecution:
    #Moon as seen from Earth:
    x = x_M_fl - x_E_fl
    y = y_M_fl - y_E_fl

    fig = plt.figure(figsize=(12,6))
    fig.suptitle('Vergrößerung der Mondbahn über 100 Jahre')
    ax = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)

    ax.set_title('Mond im Erdsystem')
    ax.set_xlabel("$x$ / m")
    ax.set_ylabel("$y$ / m")
    ax.set_aspect('equal')
    ax.plot(x, y, color='r', linewidth=0.5, label='Mond')

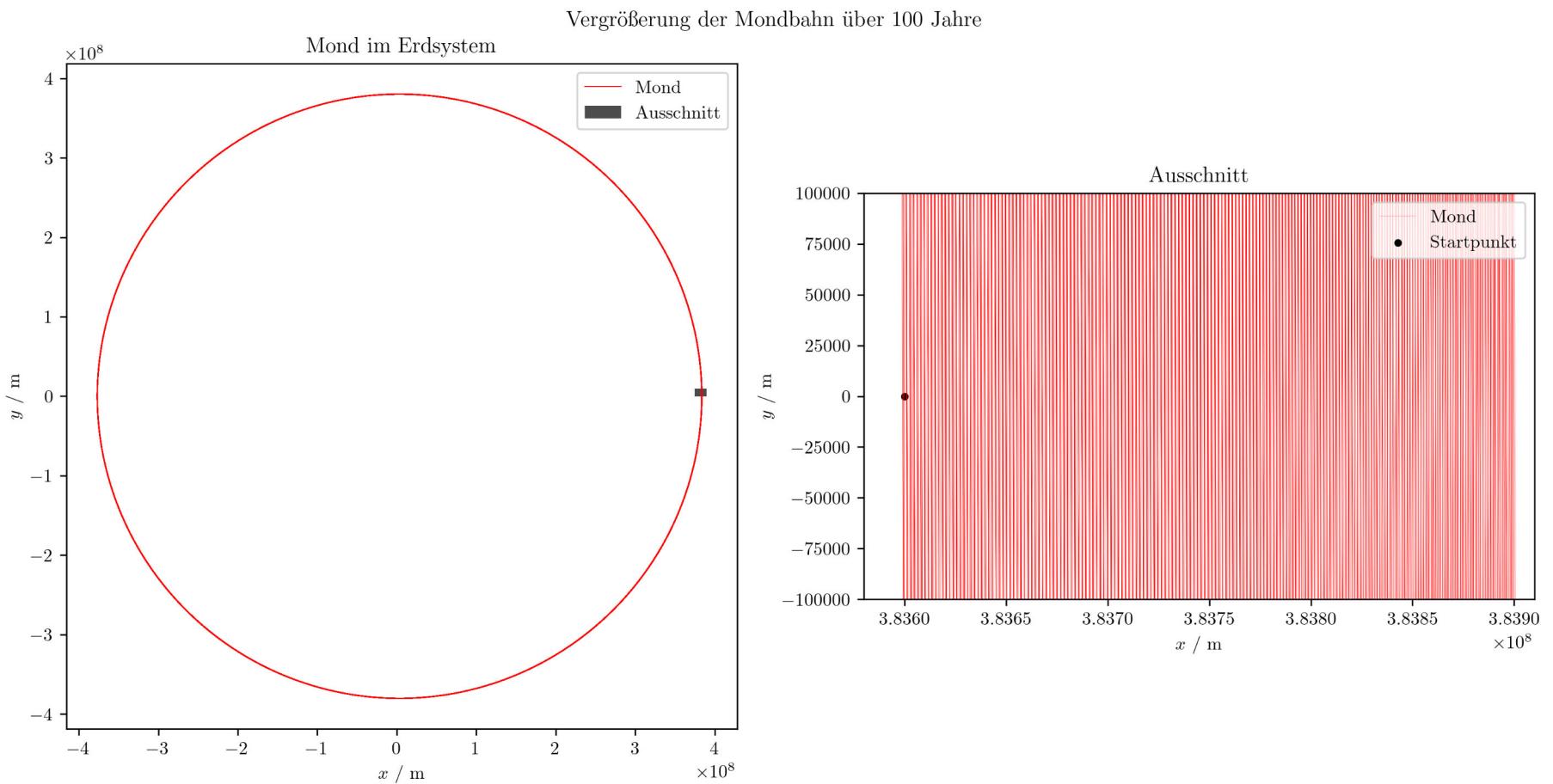
    rect = plt.Rectangle((3.75e8, -1e5), 1.5e7, 1e7, facecolor="black", alpha=0.7, label='Ausschnitt')
    ax.add_patch(rect)
    ax.legend()

    ax2.set_title('Ausschnitt')
    ax2.set_xlabel("$x$ / m")
    ax2.set_ylabel("$y$ / m")
    ax2.set_aspect('equal')
    ax2.plot(x, y, 'r', linewidth=0.1, label='Mond')
    ax2.scatter(x[0], y[0], c='k', s=10, label='Startpunkt')
    ax2.set_xlim([3.8358e8, 3.8391e8])
    ax2.set_ylim([-1e5, 1e5])
    ax2.legend()

    plt.tight_layout()
    plt.savefig("4body_friction_Moon.png", dpi=300)
    plt.close()

```

display(Image(filename='4body_friction_Moon.png', width=1000, height=2000))



Berechnung der Tageslängenänderung

Für die erstmalige Berechnung der Tageslängenänderung verwenden wir die Reibungskonstante $k = 2 \cdot 10^{-12} \frac{1}{\text{m}}$ und Massen für Flutberge von $m_1 = m_2 = \frac{1}{2} m_{\text{Ozean}} = \frac{1}{2} \cdot 0.0014 \cdot 10^{24}$

Die Verlangsamung der Erdrotation kann man auch in Realität beobachten. Deshalb vergleichen wir unsere Ergebnisse mit dem Literaturwert für die Tageslängenänderung über 100 Jahre $\tau = 0.0021 \frac{\text{s}}{100\text{a}}$.

Wie im Graphen für die Winkelgeschwindigkeiten der Flutberge zu erkennen ist braucht das System eine gewisse Einschwingzeit. Wir wählen konservativ 200 Jahre als Einschwingzeit (mit Hilfe des Graphen bestimmt). Im folgenden berechnen wir die Tageslängenänderung für 500 Jahre mit 100 Jahr Zeitschritten.

Die Simulation liefert uns die Winkelgeschwindigkeit der Erde ω_E zu gewissen Zeitpunkten. Diese kann mit der Formel $\omega_E = \frac{2\pi}{T} \Rightarrow T = 2\pi/\omega$ in die Tageslänge T umgerechnet werden. Die Differenz der Tageslängen ergibt die Tageslängenänderung τ .

```
In [ ]: if not fastExecution:
    year_in_seconds = 365*24*3600
    t_f5ha, _, _, _, _, _, _, _, omega_E_f5ha = four_body_problem_friction(*iv_stable_orbit_4body_friction(), k=2e-12, t
    angular_velocities = [omega_E_f5ha[i] for i in np.argmin(np.abs(t_f5ha - j*year_in_seconds)) for j in [0, 100, 200, 300,
    tau_calc = [2*pi/angular_velocities[i+1] - 2*pi/angular_velocities[i] for i in range(5)]
    for i in range (5):
        print(f'\t für ({i}ha-{i+1}ha): {tau_calc[i]:.3f} s/ha')
else:
    print("τ für (0ha-1ha): 137.090 s/ha\nτ für (1ha-2ha): 137.086 s/ha\nτ für (2ha-3ha): 137.082 s/ha\nτ für (3ha-4ha): 137.078 s/ha
    τ für (4ha-5ha): 137.073 s/ha
```

Das berechnete Ergebnis von ca. $137 \frac{s}{100a}$ weicht stark vom Literaturwert τ ab.

Es ist deutlich zu erkennen, dass die Tageslängenänderung mit der Zeit abnimmt, da die Reibungskraft mit der Zeit abnimmt. Auf unseren Zeitskalen ist eine Näherung als linear aber noch vertretbar. Die Einschwingzeit scheint für die Berechnung nicht relevant zu sein, da in den ersten 100 Jahren kein deutlich größerer Unterschied des Ergebnisses zu den folgenden Zeitschritten zu erkennen ist.

Da unser berechneter Wert so stark vom Literaturwert abweicht, möchten wir die Parameter unserer Simulation anpassen um den Literaturwert zu erreichen. Dies verwenden wir für die folgenden Fits, um die Laufzeit zu verkürzen in dem wir die Berechnungen nur für die ersten 100 Jahre durchführen.

Shooting Methode und Variation der Flutbergmasse und Reibungskonstante

Die Reibung der Flutberge auf der Erdoberfläche sorgt für eine Verlangsamung der Erdrotation und einem größeren Orbit für den Mond. Wir haben für diese Berechnung zwei freie Parameter, die wir im folgenden anpassen wollen:

- **Die Masse der Flutberge m_1 und m_2 (die Masse des Ozeans)**
- **Die Reibungskonstante k**

Shooting Methode

Mit Hilfe der Shooting Methode kann man Randwertprobleme numerisch auf Anfangswertprobleme zurückführen. Es wird zu Anfang ein Intervall geraten in dem die mögliche Lösung liegt. Für die Mitte des Intervalls wird das Problem numerisch gelöst und mit dem Randwert verglichen. Es wird nun mit Hilfe des Bisektionverfahrens das Intervall solange halbiert bis die Lösung ausreichend genau gefunden wurde.

Variation der Masse der Flutberge

Wir modifizieren die Massen der Flutberge so, dass sie für die aktuellen Zunahme der Tageslänge pro 100 Jahre mit dem oben angegebenen Literaturwert τ übereinstimmt (hier bleibt k unverändert).

In der Differentialgleichung für die Erdrotationsbeschleunigung sehen wir, dass diese linear mit der Masse der Flutberge zusammenhängt. Die berechnete Erdrotationsbeschleunigung ist zu schnell, deshalb müssen wir die Masse der Flutberge verkleinern. Wir wählen deshalb für die Shooting Methode als Suchintervall Null bis zur aktuellen Ozeanmasse. Es wäre möglich die Massen der Flutberge einzeln zu variieren, wir haben uns aber entschieden sie für die folgenden Berechnungen gleich zu setzen.

Variation der Reibungskonstante

Die verwendete Ozeanmasse entspricht bereits den reellen Begebenheiten, deshalb ist es nicht sinnvoll diese zu fitten.

Alternativ können wir die Reibungskonstante verändern. Auch sie ist aus den gleichen Gründen zu groß und wir wählen als Suchintervall Null bis zum aktuellen Wert. Die Masse der Flutberge lassen wir hier unverändert.

Zum Ausführen der Shooting-Methode müssen wir noch eine Toleranz angeben. Der Literaturwert $\tau = 0.0021$ ist mit einer Genauigkeit von 10^{-5} angegeben, deshalb wählen wir diese als Toleranz.

```
In [ ]: def rotation_period_change_mOcean(mOcean: float):
    '''Calculates the change in rotation_period after 100 years, given a value for mOcean'''
    initial_values = iv_stable_orbit_4body_friction()
    # Change the mass of the two tides to each be half the value of mOcean for the simulation
    initial_values[-1][-1], initial_values[-1][-2] = mOcean/2, mOcean/2
    _, _, _, _, _, _, _, _, _, omega_E = four_body_problem_friction(*initial_values, k=2*10**(-12), t_max=100*365*24*3600)
    return (2*pi/omega_E[-1]) - (2*pi/omega_E[0]) # Calculate the change in rotation period as described above

def rotation_period_change_k(k: float):
    '''Calculates the change in rotation_period after 100 years, given a value for k'''
    _, _, _, _, _, _, _, _, _, omega_E = four_body_problem_friction(*iv_stable_orbit_4body_friction(), k=k, t_max=100*365*2
    return (2*pi/omega_E[-1]) - (2*pi/omega_E[0]) # Calculate the change in rotation period as described above

def shooting_method(func, target: float, val_min: float, val_max: float, tol: float):
    '''Find the value of x for which func(x) is in the interval [target-tol, target+tol]
    Search in the interval [val_min, val_max]'''

    # Check if the solution is in the passed interval
    if func(val_min) > target or func(val_max) < target:
        raise ValueError('The target is not in the passed interval')
    print("Target in interval")

    # Iterate until the solution is within the tolerance
    while True:
        x = (val_min + val_max)/2 # Calculate the midpoint of the interval
        func_val = func(x) # Calculate the value of the function at the midpoint
        if np.abs(func_val - target) < tol: # Solution found if the absolute distance to the solution is smaller than the tolerance
            return x
        elif func_val < target: # The lower bound can be raised
            val_min = x
        else: # The upper bound can be lowered
            val_max = x

    # Fit the value of mOceans so that the change in rotation period is tau
if not fastExecution:
    mOzean_fit = shooting_method(func=rotation_period_change_mOcean, target=tau, val_min=0, val_max=mOzean, tol=0.00001)
else:
    mOzean_fit = 2.13623046875e+16

# Fit the value of k so that the change in rotation period is tau
if not fastExecution:
    k_fit = shooting_method(func=rotation_period_change_k, target=tau, val_min=0, val_max=2*10**(-12), tol=0.00001)
else:
    k_fit = 3.06640625e-17

print(f'mOzean_fit: {mOzean_fit:.3g} kg')
print(f'k_fit: {k_fit:.3g} 1/m')
```

mOzean_fit: 2.14e+16 kg
k_fit: 3.07e-17 1/m

Ergebnisse Fit durch Shooting Methode

Wir erhalten wie erwartet durch den Fit für Ozeanmasse und Reibungskonstante kleinere Werte als unsere Startwerte.

Der gefittete Wert für die Ozeanmasse liegt aber mit $2.14 \cdot 10^{16}$ kg deutlich unter dem vorher verwendeten Literaturwert von $1.4 \cdot 10^{21}$ kg Quelle: [Ocean#Weigth](#).

Die Abweichungen lassen sich durch mehrere Näherungen in der Simulation erklären:

- Korrektheit der Reibungskonstante
- Die Sonne wird nicht berücksichtigt. Sie trägt einen kleinen Teil zu den Gezeiten bei.
- Die Erde verformt sich durch die Gezeitenkräfte
- Landmassen und ihre Effekte auf die Gezeiten werden nicht berücksichtigt

Nach den obigen Betrachtungen scheint es also sinnvoller statt der Ozeanmasse die Reibungskonstante zu fitten. Diese verhält sich auch viel mehr wie ein freier Parameter, da sie viele in der Realität auftretende Effekte zusammenfasst. (z.B. Landmassen, verschiedene Meerestiefen, Strömungen)

Vergleich der gefundenen Reibungskonstante mit der gegebenen

```
In [ ]: if not fastExecution:
    # 4 body with friction for 100 years: 11min
    t_kfit, x_E_kfit, y_E_kfit, x_M_kfit, y_M_kfit, phi_1_kfit, phi_2_kfit, phi_E_kfit, omega_1_kfit, omega_2_kfit, omega_E_kf
    x_1_kfit, y_1_kfit, x_2_kfit, y_2_kfit = x_E_kfit + RErde*np.cos(phi_1_kfit), y_E_kfit + RErde*np.sin(phi_1_kfit), x_E_kfi
```

```
In [ ]: if not fastExecution:
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10))
    plt.rc('axes', axisbelow=True)
    plt.suptitle("Vergleich 4 Körper Problem $k=2*10^{-12} / k=3*10^{-17}$")

    def plot_subplot(ax, lines, title, xlabel, ylabel):
        for x, y, label, color in lines:
            ax.plot(x, y, color, linewidth=1, label=label)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.set_title(title)

        ax.grid('minor', 'minor', linestyle='-', linewidth=0.2)
        ax.grid('major', 'major', linestyle='-', linewidth=0.8)
        ax.minorticks_on()
        ax.legend()

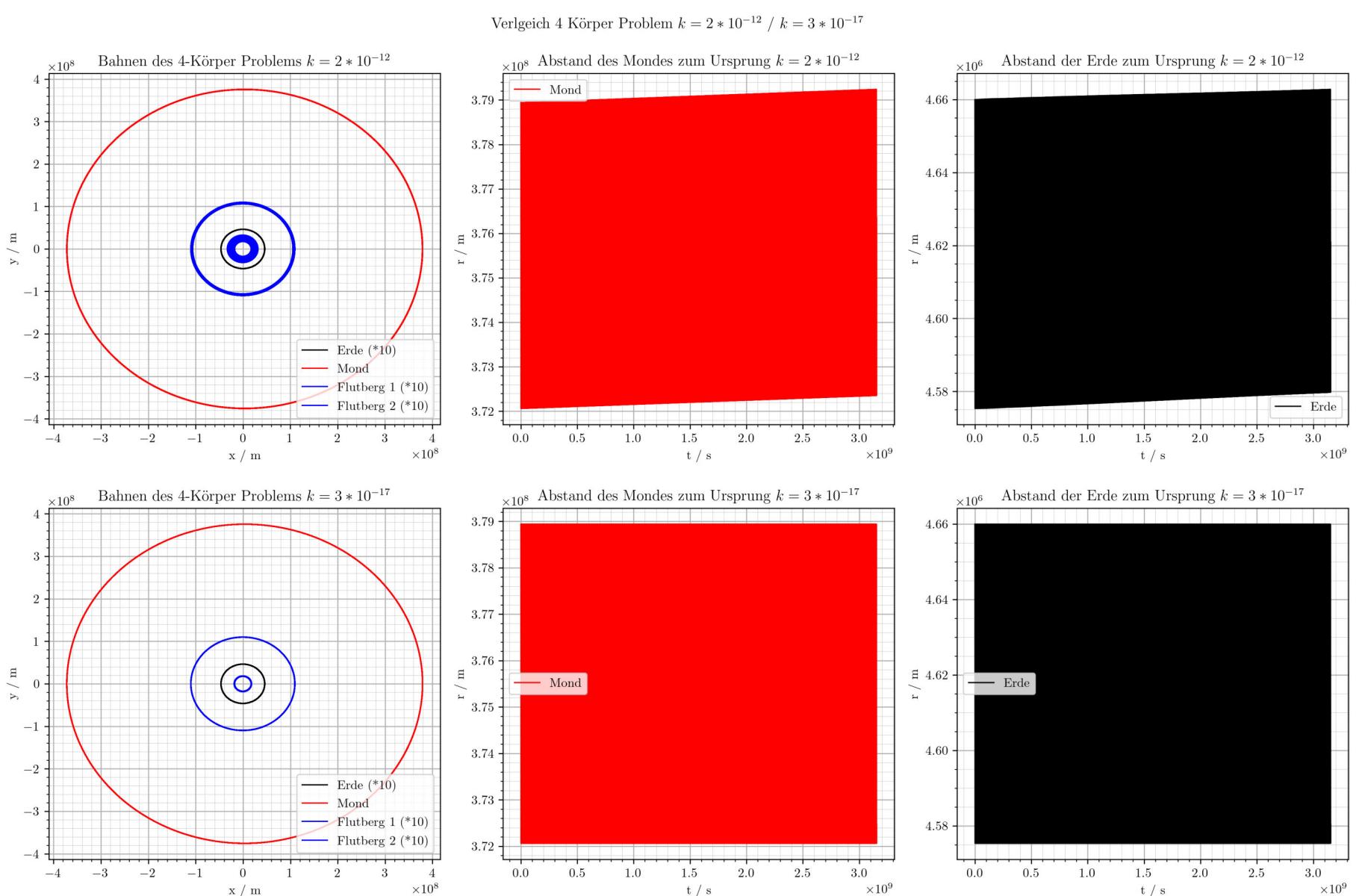
    # Distance from the origin for earth and moon
    r_Earth_fl = np.sqrt(x_E_fl**2 + y_E_fl**2)
    r_Moon_fl = np.sqrt(x_M_fl**2 + y_M_fl**2)
    r_Earth_kfit = np.sqrt(x_E_kfit**2 + y_E_kfit**2)
    r_Moon_kfit = np.sqrt(x_M_kfit**2 + y_M_kfit**2)

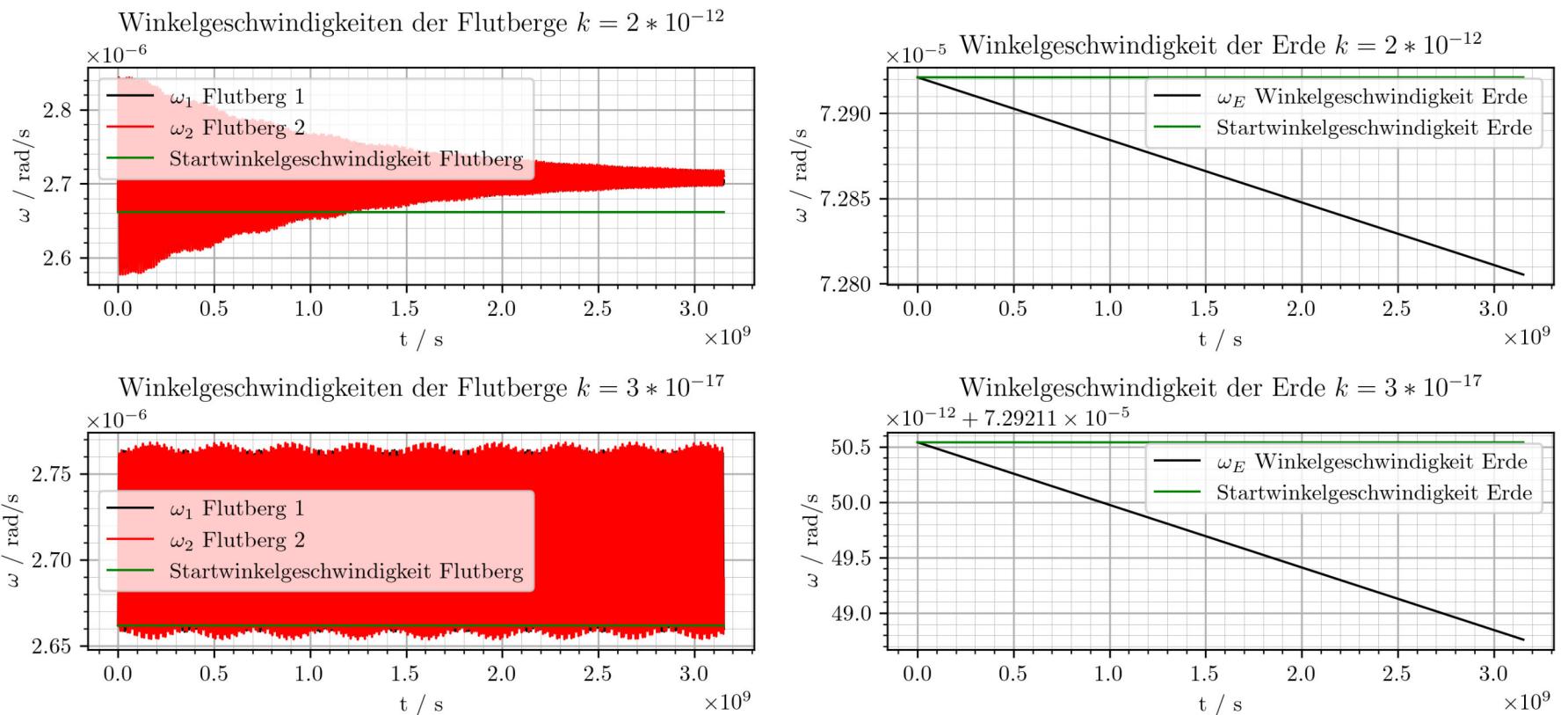
    plot_subplot(axes[0][0], [(x_E_fl*10, y_E_fl*10, 'Erde (*10)', 'k'), (x_M_fl, y_M_fl, 'Mond', 'r'), (x_1_fl*10, y_1_fl*10, 'Flutberg 1 (*10)', 'k'), (x_2_fl*10, y_2_fl*10, 'Flutberg 2 (*10)', 'k')], "Abstand des Mondes zum Ursprung $k=2*10^{-12}$", "t / s", "r / m")
    plot_subplot(axes[0][1], [(t_fl, r_Moon_fl, 'Mond', 'r')], "Abstand der Erde zum Ursprung $k=2*10^{-12}$", "t / s", "r / m")
    plot_subplot(axes[0][2], [(t_fl, r_Earth_fl, 'Erde', 'k')], "Abstand der Erde zum Ursprung $k=2*10^{-12}$", "t / s", "r / m")
    plot_subplot(axes[1][0], [(x_E_kfit*10, y_E_kfit*10, 'Erde (*10)', 'k'), (x_M_kfit, y_M_kfit, 'Mond', 'r'), (x_1_kfit*10, y_1_kfit*10, 'Flutberg 1 (*10)', 'k'), (x_2_kfit*10, y_2_kfit*10, 'Flutberg 2 (*10)', 'k')], "Abstand des Mondes zum Ursprung $k=3*10^{-17}$", "t / s", "r / m")
    plot_subplot(axes[1][1], [(t_kfit, r_Moon_kfit, 'Mond', 'r')], "Abstand der Erde zum Ursprung $k=3*10^{-17}$", "t / s", "r / m")
    plot_subplot(axes[1][2], [(t_kfit, r_Earth_kfit, 'Erde', 'k')], "Abstand der Erde zum Ursprung $k=3*10^{-17}$", "t / s", "r / m")
    plt.tight_layout()
    plt.savefig("4body_friction_comp1_k_fit.png", dpi=300)
    plt.close()

    fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 5))
    plt.rc('axes', axisbelow=True)
    plt.suptitle("Vergleich 4 Körper Problem $k=2*10^{-12} / k=3*10^{-17}$")

    plot_subplot(axes[0][0], [(t_fl, omega_1_fl, '$\omega_1$ Flutberg 1', 'k'), (t_fl, omega_2_fl, '$\omega_2$ Flutberg 2', 'r')], "Winkelgeschwindigkeit Flutberg 1", "t / s", "r / m")
    plot_subplot(axes[0][1], [(t_fl, omega_E_fl, '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Erde", "t / s", "r / m")
    plot_subplot(axes[1][0], [(t_kfit, omega_1_kfit, '$\omega_1$ Flutberg 1', 'k'), (t_kfit, omega_2_kfit, '$\omega_2$ Flutberg 2', 'r')], "Winkelgeschwindigkeit Flutberg 1", "t / s", "r / m")
    plot_subplot(axes[1][1], [(t_kfit, omega_E_kfit, '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Erde", "t / s", "r / m")
    plt.tight_layout()
    plt.savefig("4body_friction_comp2_k_fit.png", dpi=300)
    plt.close()

# Display image set the size to be 500x500 pixels
display(Image(filename='4body_friction_comp1_k_fit.png', width=1000, height=500))
display(Image(filename='4body_friction_comp2_k_fit.png', width=1000, height=500))
```





Da die gefüttete Reibungskonstante um fünf Größenordnungen kleiner ist, sind die zuvor diskutierten Reibungseffekte sehr viel kleiner. Die Tageslängenänderung lässt sich jedoch trotzdem erkennen. Die anderen Effekte (Entfernung des Mondes von der Erde, Einschwingen), sind jedoch so klein, dass sie in den Plots nicht zu sehen sind.

Vielteilchenproblem mit Randbedingungen, intrinsischer Rotation und Reibung

Nun betrachten wir das Erde-Mond System mit N Massepunkten auf der Erdoberfläche, welche die Wassermassen darstellen. Ziel ist es das Ausbilden der Flutberge zu simulieren. Für die Mond- und Erdkoordinaten verwenden wir die Differentialgleichung des Zweikörperproblems, als Näherung um den Rechenaufwand zu reduzieren. Die Näherung wird später genauer diskutiert. Die Kraft auf die Massenpunkte ist gleich zur Kraft auf die zwei Flutberge mit Reibung, welche oben erläutert wurde. Die Differentialgleichung für ω_E wird mit N Massepunkten, analog zum Fall mit zwei Flutbergen, zu:

$$\dot{\omega}_E = \frac{5kR_E}{2m_E} \sum_{i=1}^N m_i |\omega_i - \omega_E| (\omega_i - \omega_E)$$

In []: `N = 25 # Amount of tides`

Wir verwenden wenn möglich die bereits implementierten Gleichungen. Die Winkel und Winkelbeschleunigungen der Flutberge werden als Liste übergeben, um N leicht ändern zu können.

```
In [ ]: def eq_motion_Nbody(t, state, mass, k):
    '''This function calculates the derivatives of the state vector the 2 body problem with N Tides to be passed to solve_ivp.
    state: state vector has the following structure: [x_E, y_E, x_M, y_M, phi_E, vx_E, vy_E, vx_M, vy_M, omega_E, phi_1, phi_2
    mass: list of masses e.g. [m_E, m_M, m1, m2, ..., mN]'''

    # Unpack the state vector and masses:
    x_E, y_E, x_M, y_M, phi_E, vx_E, vy_E, vx_M, vy_M, omega_E, phi_tide, omega_tide = *state[0:10], [state[10+i] for i in range(N)]
    m_E, m_M, m_tide = *mass[0:2], [mass[2+i] for i in range(N)]

    # Earth and moon acceleration from the 2 body problem:
    _, _, _, _, ax_E, ay_E, ax_M, ay_M = eq_motion_2body(t, [x_E, y_E, x_M, y_M, vx_E, vy_E, vx_M, vy_M], [m_E, m_M])

    # Angular momentum of the earth:
    alpha_E = (5*k*RErde)/(2*m_E) * np.sum([m_tide[i]*abs(omega_tide[i] - omega_E)*(omega_tide[i] - omega_E) for i in range(N)])

    # The tides' cartesian coordinates:
    x_tide, y_tide = [x_E + RErde*np.cos(phi_tide[i]) for i in range(N)], [y_E + RErde*np.sin(phi_tide[i]) for i in range(N)]

    # The earth's angular acceleration:
    alpha_tide = [tides_acceleration_friction(x_M, y_M, x_tide[i], y_tide[i], phi_tide[i], ax_E, ay_E, omega_tide[i], omega_E, k) for i in range(N)]

    return [vx_E, vy_E, vx_M, vy_M, omega_E, ax_E, ay_E, ax_M, ay_M, alpha_E] + omega_tide + alpha_tide
```

Da für die Erd- und Mondkoordinaten die Zweikörper-Differentialgleichung verwendet wird wählen wir für diese dieselben Anfangsbedingungen. Die Massepunkte starten gleichmäßig verteilt auf der Erdoberfläche mit der mittleren Winkelgeschwindigkeit der Flutberge mit Reibung.

```
In [ ]: if not fastExecution:
    _, _, _, _, _, _, _, omega_Niv, _, _ = four_body_problem_friction(*iv_stable_orbit_4body_friction(), k=k_fit, t_max=100)
    omega_i0 = np.mean(omega_Niv)
else:
    omega_i0 = 2.7107783823217206e-06
```

```
In [ ]: def iv_stable_orbit_Nbody():
    '''Initial conditions for the Earth-Moon-NTides system in a stable orbit'''
    # Each of the tides gets an nth of the oceans mass
    mass = [mErde, mMond] + [mOzean/N for i in range(N)]
    # Because the DEQ for the 2 body problem is used for the earth and moon movement, we can reuse the initial conditions (Tides)
    x0_Eerde, x0_Mond, v0_Eerde, v0_Mond, _, _ = iv_stable_orbit_2body()
    _, _, _, phi0_E, _, _, omega0_E, _ = iv_stable_orbit_4body_friction()

    phi0 = [(2*pi*i)/N for i in range(N)] # The tides all start spaced evenly around the earth
    #omega0 = [2*pi/TMondBahn]*N # The tides all start with the same angular velocity as the moon
    omega0 = [omega_i0]*N

    return [x0_Eerde, x0_Mond, phi0_E, v0_Eerde, v0_Mond, omega0_E, phi0, omega0, mass]

def N_body_problem(pos_body_1: list, pos_body_2: list, phi_E: float, vel_body_1: list, vel_body_2: list, omega_E: float, phi_tide):
    solution = solve_ivp(fun=eq_motion_Nbody, t_span=[t_start, t_max], y0=[*pos_body_1, *pos_body_2, phi_E, *vel_body_1, *vel_body_2])
    x_E, y_E, x_M, y_M, phi_E, vx_E, vy_E, vx_M, vy_M, omega_E = solution.y[0:10]
    phi_tide = [solution.y[10+i] for i in range(N)]
    omega_tide = [solution.y[10+N+i] for i in range(N)]
    return [solution.t, x_E, y_E, x_M, y_M, phi_tide, omega_E, omega_tide]
```

```
In [ ]: t, x_E, y_E, x_M, y_M, phi_tide, omega_E, omega_tide = N_body_problem(*iv_stable_orbit_Nbody(), k=k_fit, t_max=20*TMondBahn)
```

```
In [ ]: # Cartesian coordinates of the tides:
x_tide, y_tide = [x_E + REerde*np.cos(phi_tide[i]) for i in range(N)], [y_E + REerde*np.sin(phi_tide[i]) for i in range(N)]

# Plot the movement of the earth, moon and tides
plt.rc('axes', axisbelow=True)
plt.figure(figsize=(5, 5))
plt.title("Bahnen des N-Flutberg-System")
plt.grid('minor', 'minor', linestyle='-', linewidth=0.2)
plt.grid('major', 'major', linestyle='-', linewidth=0.8)
plt.minorticks_on()
plt.xlabel("x")
plt.ylabel("y")
plt.plot(x_M/10, y_M/10, 'r', linewidth=1, label='Mond (/10)')
for i in range(N):
    plt.plot(x_tide[i], y_tide[i], 'b', linewidth=0.5, alpha=0.2)
plt.plot(x_E, y_E, 'k', linewidth=1, label='Erde')
plt.legend()
plt.show()
plt.close()

# Plot the angles of the tides over time, subtracting the angle of the moon
plt.rc('axes', axisbelow=True)
plt.figure(figsize=(5, 5))
plt.title("Winkel der Flutberge über Zeit, abzüglich  $t \cdot \omega_0$ ")
plt.grid('minor', 'minor', linestyle='-', linewidth=0.2)
plt.grid('major', 'major', linestyle='-', linewidth=0.8)
plt.minorticks_on()
plt.xlabel("$t / s$")
plt.ylabel("$\phi - t \cdot \omega_0 / rad$")

for i in range(N):
    plt.plot(t, phi_tide[i] - t*2*pi/TMondBahn, 'k', linewidth=0.7, alpha=0.7)

plt.show()
plt.close()

# Plot the angular velocity of the earth over time
plt.rc('axes', axisbelow=True)
plt.figure(figsize=(5, 5))
plt.title("Winkelgeschwindigkeit der Erde über Zeit")
plt.grid('minor', 'minor', linestyle='-', linewidth=0.2)
plt.grid('major', 'major', linestyle='-', linewidth=0.8)
plt.minorticks_on()
plt.xlabel("$t / s$")
plt.ylabel("$\omega_E / rad$")
plt.plot(t, omega_E, 'k', linewidth=1, label='$\omega_E$')

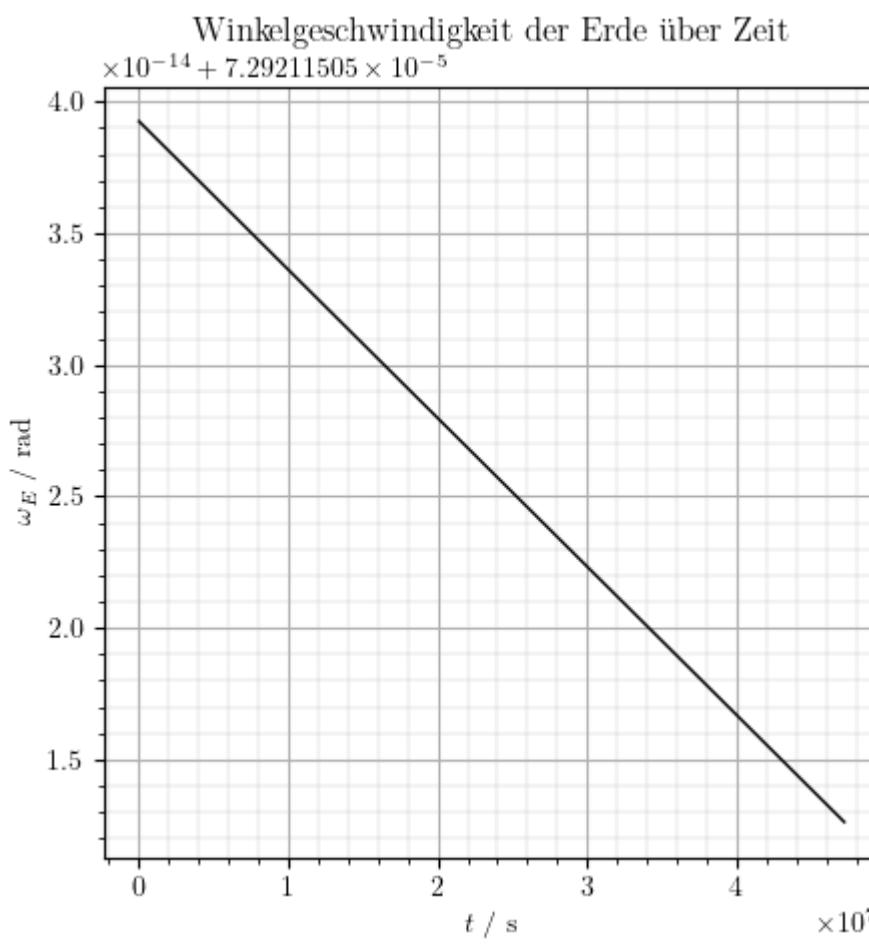
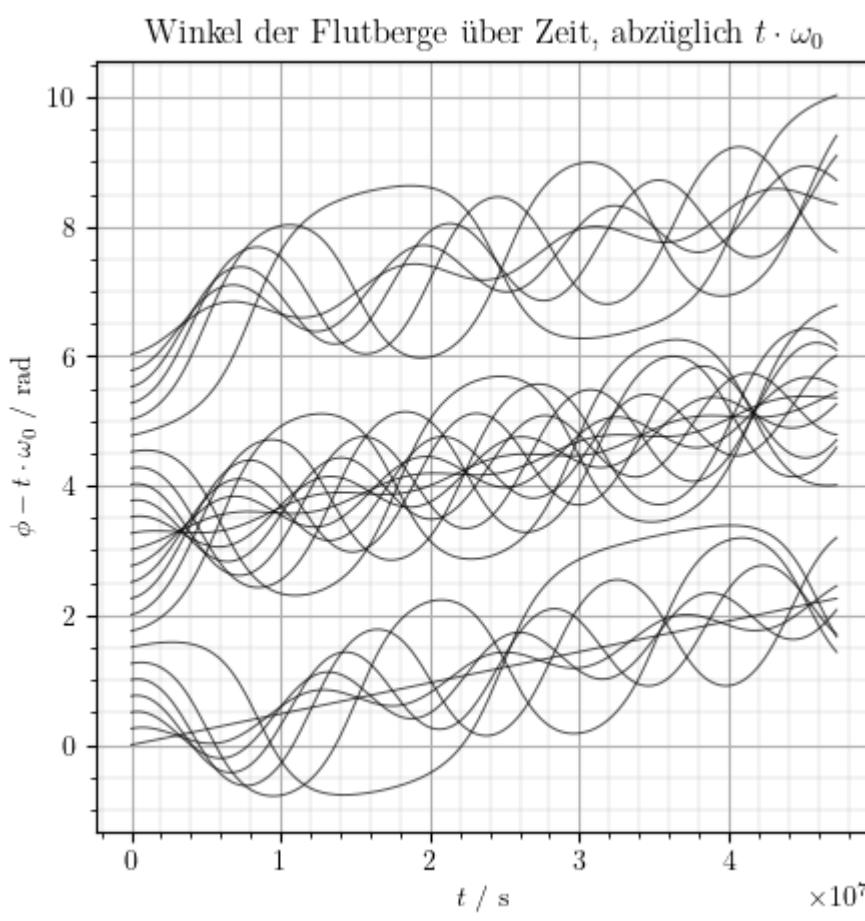
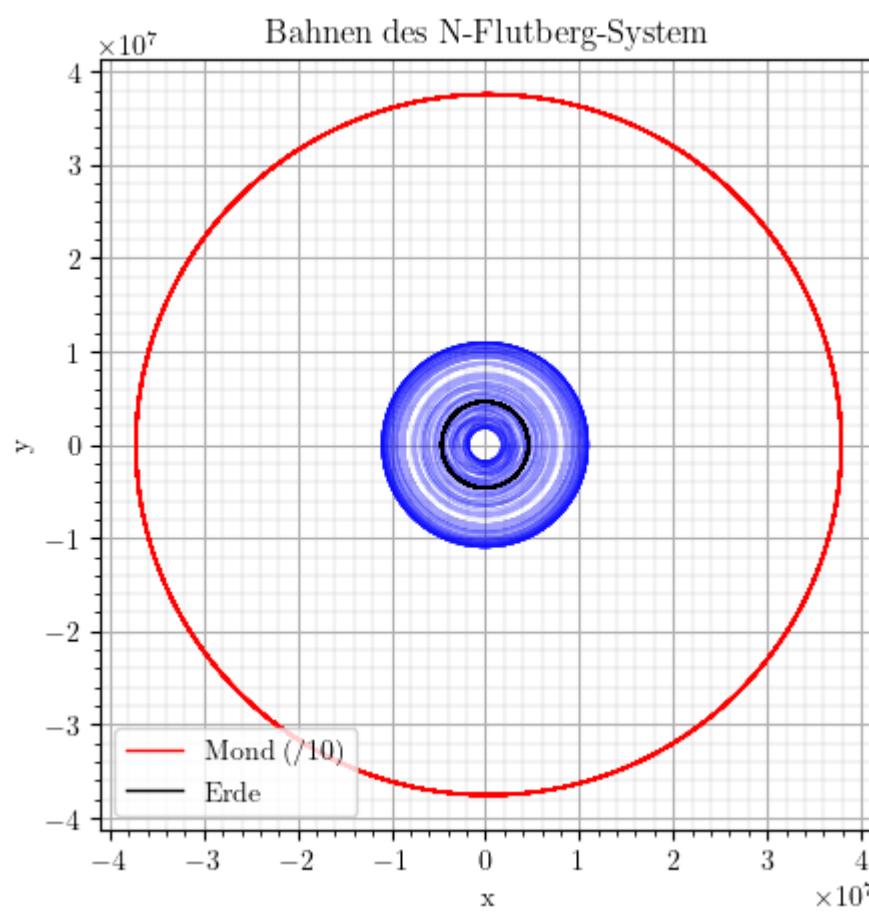
plt.show()
plt.close()

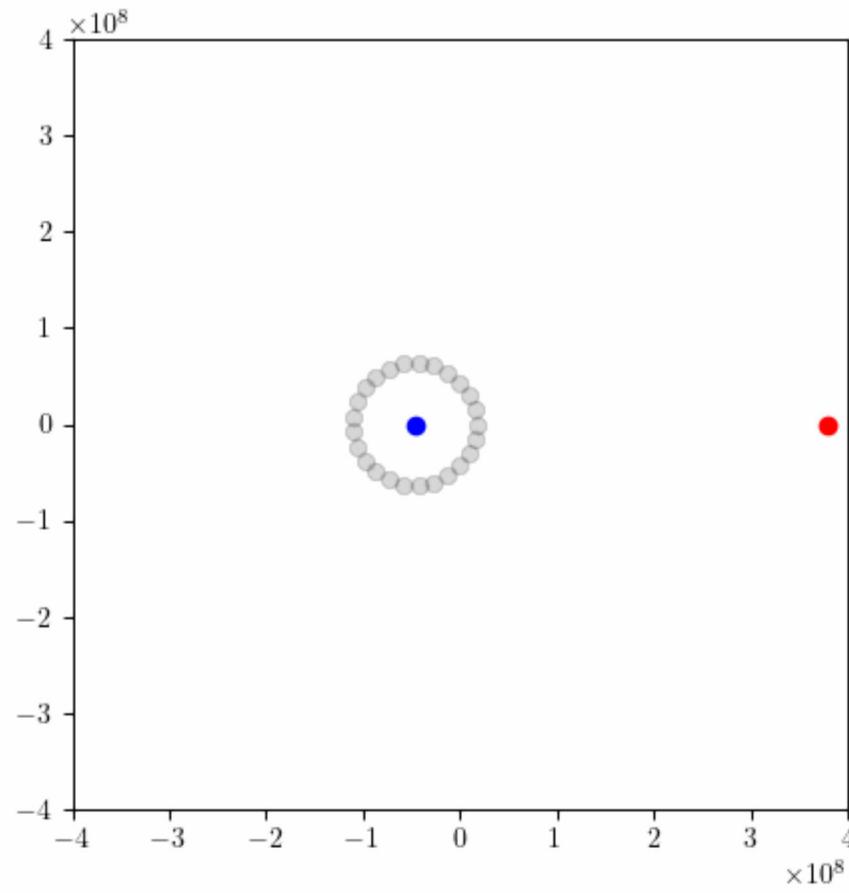
# Animation of the N body problem
if not fastExecution:
    fig = plt.figure(figsize=(5, 5))
    ax = plt.axes(xlim=(-4e8, 4e8), ylim=(-4e8, 4e8))
    ax.set_title("Animation des N-Flutberg-Systems")
    earth_line, = ax.plot([], [], marker='o', lw=0.1, color='blue', label='Erde (*10)')
    moon_line, = ax.plot([], [], marker='o', lw=0.1, color='red', label='Mond')
    tide_lines = [ax.plot([], [], marker='o', lw=0.1, color='black', label='Flutberg {i} (*10)', alpha=0.15)[0] for i in range(N)]

    def animate(i):
        earth_line.set_data([x_E[i]*10], [y_E[i]*10])
        moon_line.set_data([x_M[i]], [y_M[i]])
        for j in range(N):
            tide_lines[j].set_data([x_tide[j][i]*10], [y_tide[j][i]*10])
        return earth_line, moon_line, *tide_lines

    anim = FuncAnimation(fig, animate, init_func=None, frames=t.shape[0], interval=30, blit=True)
    anim.save('Erde_Mond_NFlutberge.gif', writer='pillow')
    plt.close()

display(Image(data=open('Erde_Mond_NFlutberge.gif', 'rb').read(), format='png'))
```





Die Flutberge bilden sich aus, sind aber nicht sehr stabil. Ein paar Massen sind immer außerhalb der Flutberge, die Mehrzahl sammelt sich jedoch dort, wo die Flutberge sein sollten. Sie oszillieren auch unterschiedlich stark um die theoretische Position der Flutberge (abhängig vom Startpunkt).

Die Näherung ist gerechtfertigt, da einerseits die Gesamtmasse der Ozeane relativ zur Erd- und Mondmasse recht klein ist. Andererseits verkleinert sich der Effekt der Flutberge auf die Erd- und Mondkoordinaten durch die Verteilung der Masse. Auf die Erde wirken die Flutberge so, dass ihre Radialbeschleunigung gegen die Erde "drückt" beziehungsweise an ihr "zieht". Da die Massen hier verteilt sind mitteln sich diese Beschleunigungen zum Teil, daher kann dieser Effekt besser vernachlässigt werden als im Fall von 2 Flutbergen. Auf den Mond wirkt die Anziehungskraft der Flutmassen, welche bewirkt, dass der Mond ein Drehmoment erfährt. Dieser Effekt ist relativ klein, wodurch der Einfluss auf die Bewegung der Flutberge ebenfalls sehr klein ist. Dadurch entfernt sich der Mond hier nicht von der Erde, was für große Zeiten ein relevanter Effekt ist.

```
In [ ]: # Two tides to compare to the previous result:
if not fastExecution:
    N = 2
    t_N, x_E_N, y_E_N, x_M_N, y_M_N, phi_tides_N, omega_E_N, omega_tides_N = N_body_problem(*iv_stable_orbit_Nbody(), k=k_fit,
    phi_1_N, phi_2_N = phi_tides_N[0], phi_tides_N[1]
    omega_1_N, omega_2_N = omega_tides_N[0], omega_tides_N[1]
    x_1_N, y_1_N, x_2_N, y_2_N = x_E_N + RErde*np.cos(phi_1_N), y_E_N + RErde*np.sin(phi_1_N), x_E_N + RErde*np.cos(phi_2_N),
```

```
In [ ]: if not fastExecution:
    fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10))
    plt.rc('axes', axisbelow=True)
    plt.suptitle("Vergleich 4 Körper Problem Exakte DGL / N=2 Näherung")

    def plot_subplot(ax, lines, title, xlabel, ylabel):
        for x, y, label, color in lines:
            ax.plot(x, y, color, linewidth=1, label=label)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.set_title(title)

        ax.grid('minor', 'minor', linestyle='-', linewidth=0.2)
        ax.grid('major', 'major', linestyle='-', linewidth=0.8)
        ax.minorticks_on()
        ax.legend()

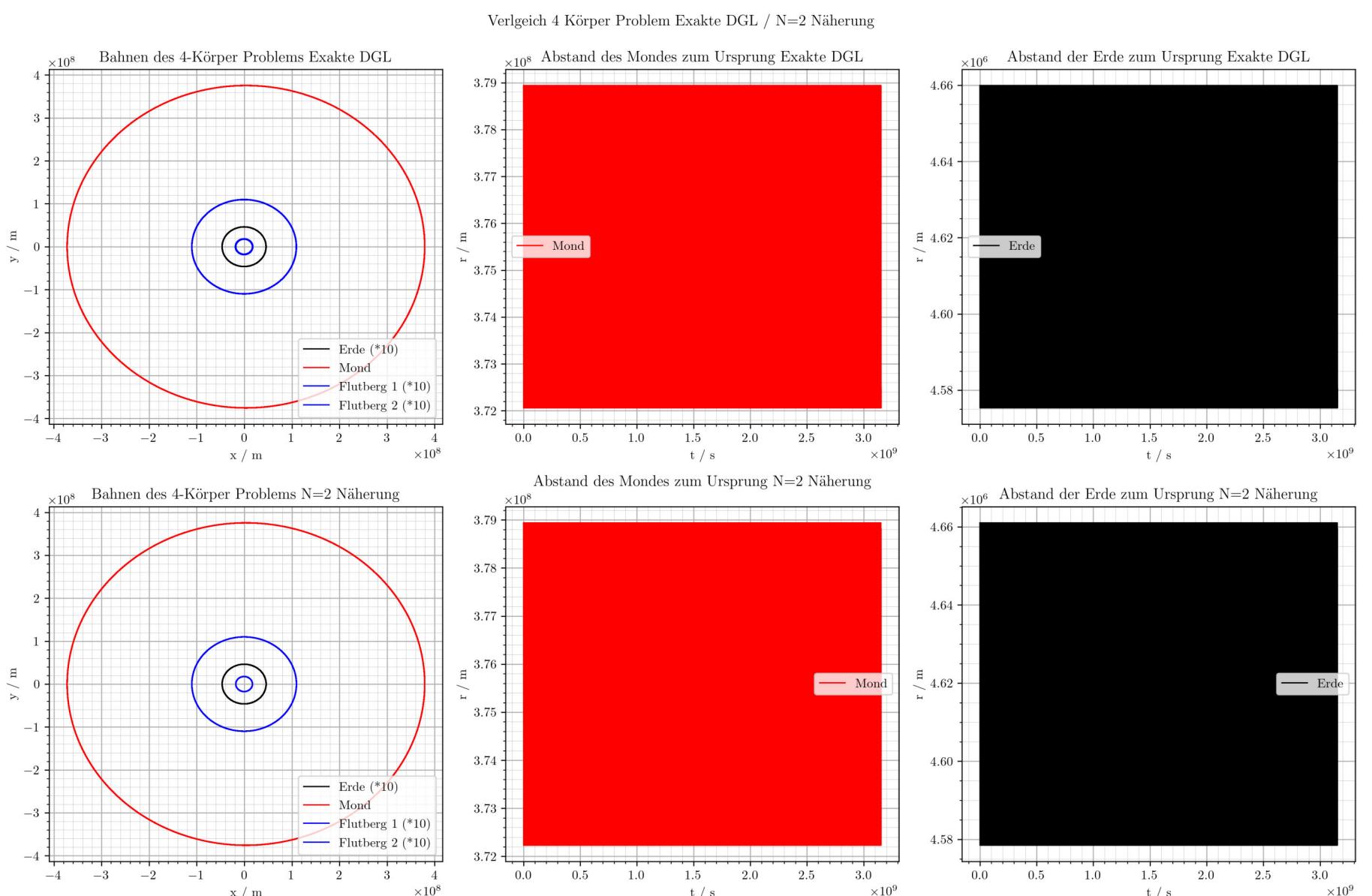
    # Distance from the origin for earth and moon
    r_Earth_kfit = np.sqrt(x_E_kfit**2 + y_E_kfit**2)
    r_Moon_kfit = np.sqrt(x_M_kfit**2 + y_M_kfit**2)
    r_Earth_N = np.sqrt(x_E_N**2 + y_E_N**2)
    r_Moon_N = np.sqrt(x_M_N**2 + y_M_N**2)

    plot_subplot(axes[0][0], [(x_E_kfit*10, y_E_kfit*10, 'Erde (*10)', 'k'), (x_M_kfit, y_M_kfit, 'Mond', 'r'), (x_1_kfit*10, y_1_kfit*10, 'Flutberg 1 (*10)', 'k'), (x_2_kfit*10, y_2_kfit*10, 'Flutberg 2 (*10)', 'k')], "Abstand des Mondes zum Ursprung Exakte DGL", "t / s", "r / m")
    plot_subplot(axes[0][1], [(t_kfit, r_Moon_kfit, 'Mond', 'r')], "Abstand der Erde zum Ursprung Exakte DGL", "t / s", "r / m")
    plot_subplot(axes[0][2], [(t_kfit, r_Earth_kfit, 'Erde', 'k')], "Abstand der Erde zum Ursprung Exakte DGL", "t / s", "r / m")
    plot_subplot(axes[1][0], [(x_E_N*10, y_E_N*10, 'Erde (*10)', 'k'), (x_M_N, y_M_N, 'Mond', 'r'), (x_1_N*10, y_1_N*10, 'Flutberg 1 (*10)', 'k'), (x_2_N*10, y_2_N*10, 'Flutberg 2 (*10)', 'k')], "Abstand des Mondes zum Ursprung N=2 Näherung", "t / s", "r / m")
    plot_subplot(axes[1][1], [(t_N, r_Moon_N, 'Mond', 'r')], "Abstand des Mondes zum Ursprung N=2 Näherung", "t / s", "r / m")
    plot_subplot(axes[1][2], [(t_N, r_Earth_N, 'Erde', 'k')], "Abstand der Erde zum Ursprung N=2 Näherung", "t / s", "r / m")
    plt.tight_layout()
    plt.savefig("4body_friction_comp1_N=2.png", dpi=300)
    plt.close()

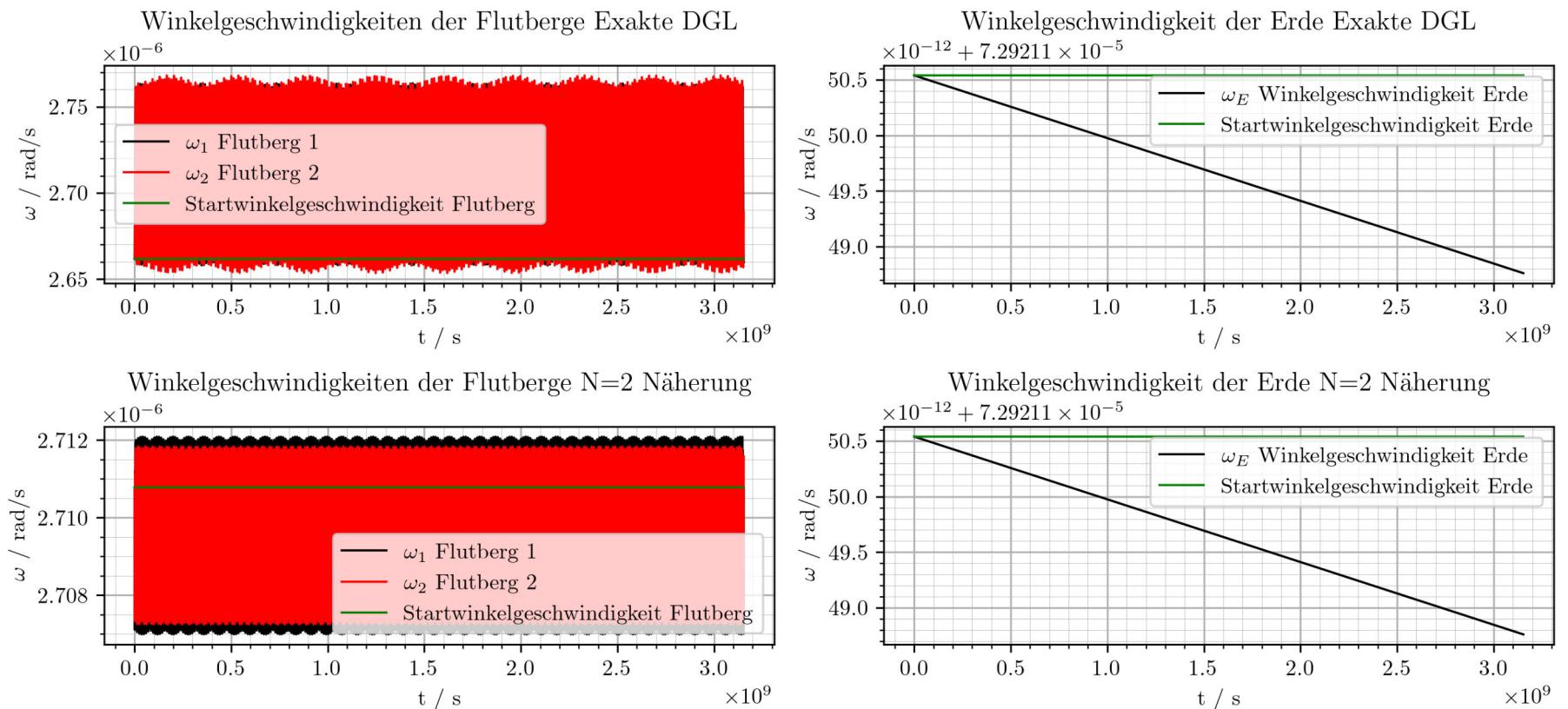
    fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 5))
    plt.rc('axes', axisbelow=True)
    plt.suptitle("Vergleich 4 Körper Problem Exakte DGL / N=2 Näherung")

    plot_subplot(axes[0][0], [(t_kfit, omega_1_kfit, '$\omega_1$ Flutberg 1', 'k'), (t_kfit, omega_2_kfit, '$\omega_2$ Flutberg 2', 'k')], "Winkelgeschwindigkeit Flutberg 1", "t / s", "r / rad")
    plot_subplot(axes[0][1], [(t_kfit, omega_E_kfit, '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Erde", "t / s", "r / rad")
    plot_subplot(axes[1][0], [(t_N, omega_1_N, '$\omega_1$ Flutberg 1', 'k'), (t_N, omega_2_N, '$\omega_2$ Flutberg 2', 'r')], "Winkelgeschwindigkeit Flutberg 1", "t / s", "r / rad")
    plot_subplot(axes[1][1], [(t_N, omega_E_N, '$\omega_E$ Winkelgeschwindigkeit Erde', 'k')], "Winkelgeschwindigkeit Erde", "t / s", "r / rad")
    plt.tight_layout()
    plt.savefig("4body_friction_comp2_N=2.png", dpi=300)
    plt.close()

# Display image set the size to be 500x500 pixels
display(Image(filename='4body_friction_comp1_N=2.png', width=1000, height=500))
display(Image(filename='4body_friction_comp2_N=2.png', width=1000, height=500))
```



Vergleich 4 Körper Problem Exakte DGL / N=2 Näherung



```
In [ ]: #Show the increase of the moons orbit
if not fastExecution:
    #Moon as seen from Earth:
    x = x_M_N - x_E_N
    y = y_M_N - y_E_N

    fig = plt.figure(figsize=(12,6))
    fig.suptitle('Veränderung der Mondbahn über 100 Jahre')
    ax = fig.add_subplot(121)
    ax2 = fig.add_subplot(122)

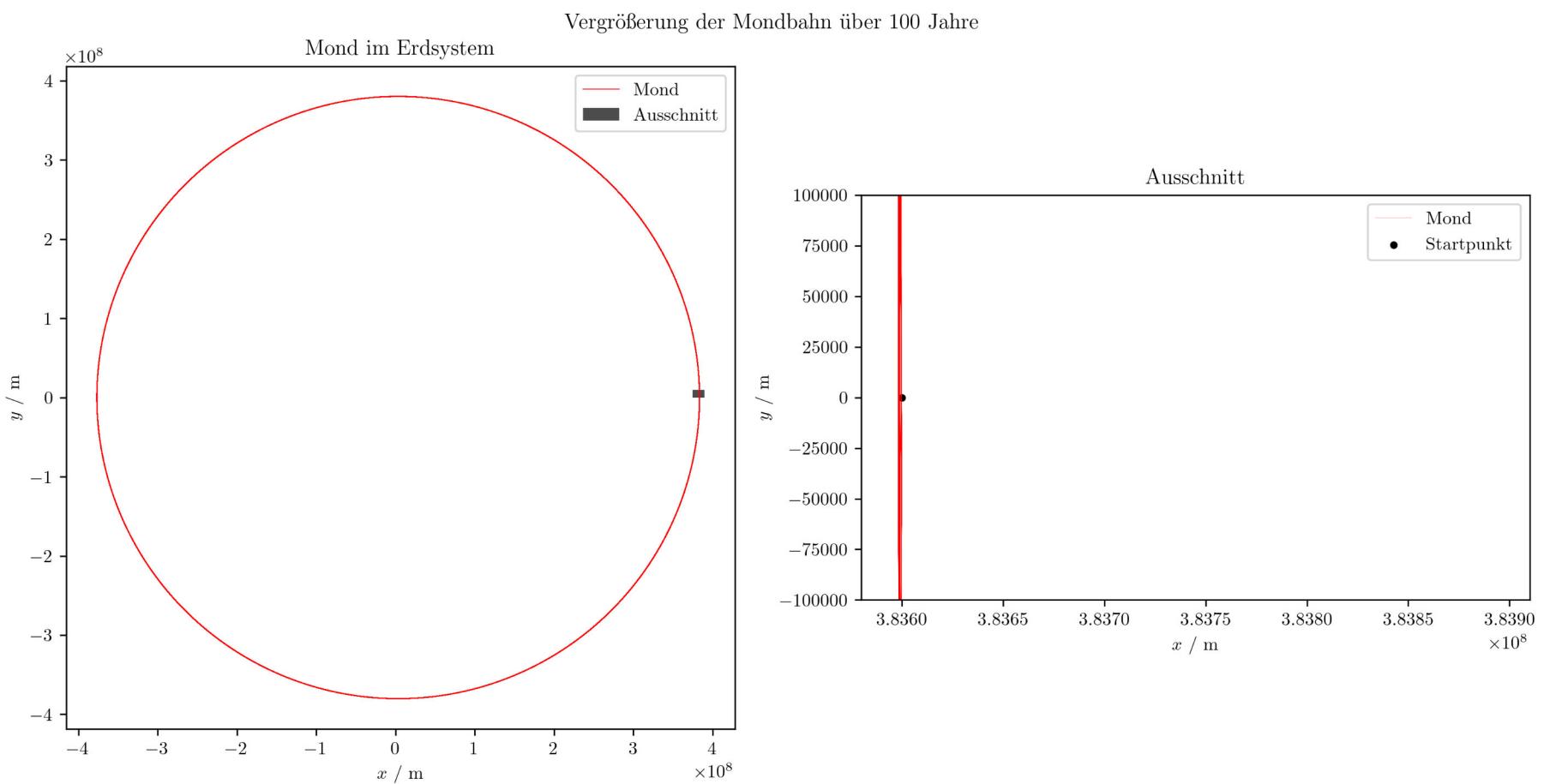
    ax.set_title('Mond im Erdsystem')
    ax.set_xlabel("$x / m$")
    ax.set_ylabel("$y / m$")
    ax.set_aspect('equal')
    ax.plot(x, y, color='r', linewidth=0.5, label='Mond')

    rect = plt.Rectangle((3.75e8, -1e5), 1.5e7, 1e7, facecolor="black", alpha=0.7, label='Ausschnitt')
    ax.add_patch(rect)
    ax.legend()

    ax2.set_title('Ausschnitt')
    ax2.set_xlabel("$x / m$")
    ax2.set_ylabel("$y / m$")
    ax2.set_aspect('equal')
    ax2.plot(x, y, 'r', linewidth=0.1, label='Mond')
    ax2.scatter(x[0], y[0], c='k', s=10, label='Startpunkt')
    ax2.set_xlim([3.8358e8, 3.8391e8])
    ax2.set_ylim([-1e5, 1e5])
    ax2.legend()

    plt.tight_layout()
    plt.savefig("4body_friction_Moon_N=2.png", dpi=300)
    plt.close()

display(Image(filename='4body_friction_Moon_N=2.png', width=1000, height=2000))
```



```
In [ ]: if not fastExecution:
    N = 2
    t, x_E, y_E, x_M, y_M, phi_tide, omega_E, omega_tide = N_body_problem(*iv_stable_orbit_Nbody(), k=k_fit, t_max=5*TMondbahn)

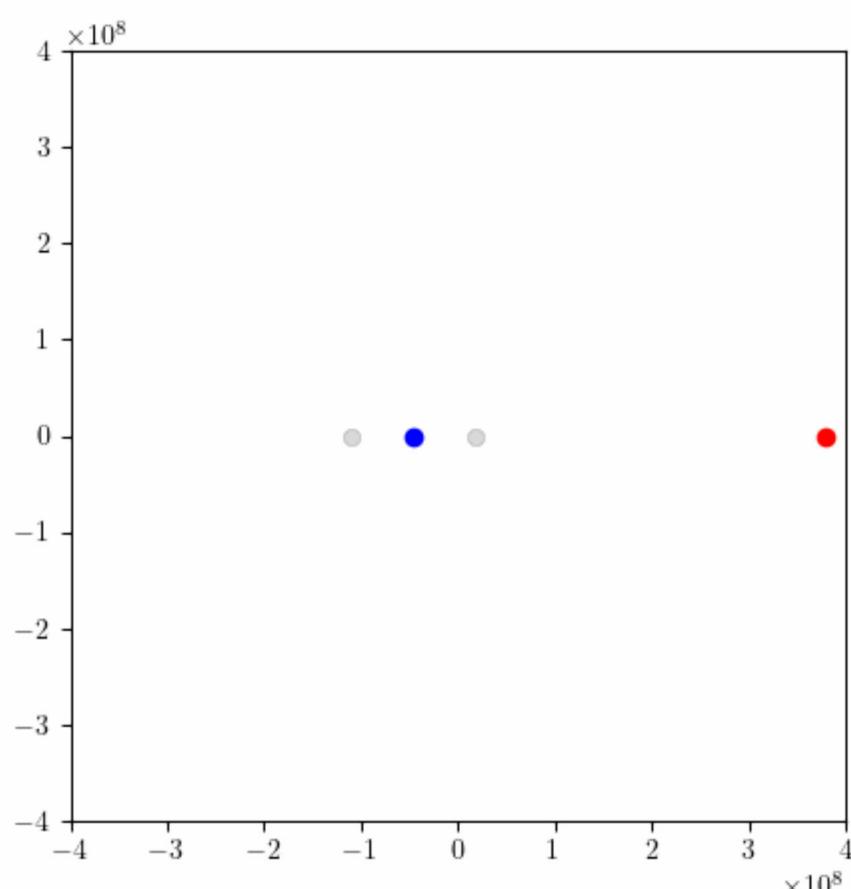
In [ ]: # Cartesian coordinates of the tides:
x_tide, y_tide = [x_E + RErde*np.cos(phi_tide[i]) for i in range(N)], [y_E + RErde*np.sin(phi_tide[i]) for i in range(N)]

# Animation of the N body problem
if not fastExecution: # Don't render if not necessary
    fig = plt.figure(figsize=(5, 5)) # Square figure, so the circles are actually circles
    ax = plt.axes(xlim=(-4e8, 4e8), ylim=(-4e8, 4e8))
    earth_line, = ax.plot([], [], marker='o', lw=0.1, color='blue', label='Erde (*10)')
    moon_line, = ax.plot([], [], marker='o', lw=0.1, color='red', label='Mond')
    tide_lines = [ax.plot([], [], marker='o', lw=0.1, color='black', label='Flutberg {i} (*10')[0] for i in range(N)]
    # ax.legend()

    def animate(i):
        earth_line.set_data([x_E[i]*10], [y_E[i]*10])
        moon_line.set_data([x_M[i]], [y_M[i]])
        for j in range(N):
            tide_lines[j].set_data([x_tide[j][i]*10], [y_tide[j][i]*10])
        return earth_line, moon_line, *tide_lines

    anim = FuncAnimation(fig, animate, init_func=None, frames=t.shape[0], interval=30, blit=True)
    anim.save('Erde_Mond_N=2Flutberge.gif', writer='pillow')
    plt.close()

display(Image(data=open('Erde_Mond_N=2Flutberge.gif','rb').read(), format='png'))
```



```
In [ ]: if not fastExecution:
    #30min
    _, _, _, _, _, omega_E, _ = N_body_problem(*iv_stable_orbit_Nbody(), k=k_fit, t_max=100*365*24*3600)

    # The change in the angular velocity
    angular_velocity_change = (omega_E[0] - omega_E[-1])

    #  $\omega = 2\pi/T \rightarrow T = 2\pi/\omega$ 
    rotation_period_at_start = (2*pi/omega_E[0])
    rotation_period_at_end = (2*pi/omega_E[-1])
    print(f'rotation_period_at_start: {rotation_period_at_start}, rotation_period_at_end: {rotation_period_at_end}')
    print(f'rotation_period_at_start [h]: {rotation_period_at_start/3600}, rotation_period_at_end [h]: {rotation_period_at_end/3600}')

    print(f'tau_lit: {tau}, tau_calc: {rotation_period_at_end - rotation_period_at_start}')
else:
    print(f'tau_lit: {tau}, tau_calc: {0.0021018376137362793}')


tau_lit: 0.0021, tau_calc: 0.0021018376137362793
```

Die größten Unterschiede zur exakten Differentialgleichung sind:

Der Mond entfernt sich nicht von der Erde, da die Gravitation der Flutberge auf den Mond vernachlässigt wird. Auch die Schwerpunktsbewegung unterscheidet sich, was aufgrund der entsprechend gewählten Anfangsbedingungen nicht zu sehen ist.

Die Bewegung der Flutberge bleibt sehr ähnlich, um diese zu betrachten ist die Näherung also gut. Daher bleibt auch die Verlangsamung der Erdrotation fast gleich. Die Winkelgeschwindigkeiten unterscheiden sich nur so stark, weil wir unterschiedliche Anfangswerte für sie gewählt haben. Insgesamt sehen sich die Lösungen sehr ähnlich, und solange man nicht an der Änderung der Mondbahn interessiert ist, ist die Näherung auch gerechtfertigt. Für sehr lange Zeiten ist sie dementsprechend nicht ausreichend.